hazelcast

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

Table of Contents

Overview

Hazelcast is a distributed IMDG, i.e. in-memory data grid, which is used widely across industries by companies like Nissan, JPMorgan, Tmobile, to name a few.

It offers various rich features including distributed cache to store key-value pairs, constructs to create and use distributed data structure, and a way to distribute your computation and queries among nodes in a cluster.

Hazelcast is a very useful tool in developing applications that require high scalability, performance, and availability.

Audience

This tutorial deep dives into various features that make Hazelcast a very useful tool. It is directed towards software professionals who want to develop highly scalable and performant applications. Post this tutorial, you would have intermediate knowledge of Hazelcast and its usage.

Prerequisite

To make the most of this tutorial, you should have working knowledge of Data Structures, while having some exposure to Java is preferable.

Hazelcast – Introduction

Distributed In-memory Data Grid

A data grid is a superset to distributed cache. Distributed cache is typically used only for storing and retrieving key-value pairs which are spread across caching servers. However, a data grid, apart from supporting storage of key-value pairs, also supports other features, for example,

- It supports other data structures like locks, semaphores, sets, list, and queues.
- It provides a way to query the stored data by rich querying languages, for example, SQL.
- It provides a distributed execution engine which helps to operate on the data in parallel.

Benefits of Hazelcast

- **Support multiple data structures**: Hazelcast supports the usage of multiple data structures along with Map. Some of the examples are Lock, Semaphore, Queue, List, etc.
- **Fast R/W access**: Given that all the data is in-memory, Hazelcast offers very high-speed data read/write access.
- **High availability**: Hazelcast supports the distribution of data across machines along with additional support for backup. This means that the data is not stored on a single machine. So, even if a machine goes down, which occurs frequently in a distributed environment, the data is not lost.
- **High Performance**: Hazelcast provides constructs which can be used to distribute the workload/computation/query among multiple worker machines. This means a computation/query uses resources from multiple machines which reduces the execution time drastically.
- **Easy to use**: Hazelcast implements and extends a lot of `java.util.concurrent` constructs which make it very easy to use and integrate with the code. Configuration to start using Hazelcast on a machine just involves adding the `Hazelcast jar` to our classpath.

Hazelcast vs Other Caches & Key-Value stores

Comparing Hazelcast with other caches like Ehcache, Guava, and Caffeine may not be very useful. It is because, unlike other caches, Hazelcast is a distributed cache, that is, it spreads the data across machines/JVM. Although Hazelcast can work very well on single JVM as well, however, it is more useful is a distributed environment.

Similarly comparing it with Databases like MongoDB is also of not much use. This is because, Hazelcast mostly stores data in memory (although it also supports writing to disk). So, it offers high R/W speed with the limitation that data needs to be stored in memory.

Hazelcast also supports caching/storing complex data types and provides an interface to query them, unlike other data stores.

A comparison, however, can be made with **Redis** which also offers similar features.

Hazelcast vs Redis

In terms of features, both Redis and Hazelcast are very similar. However, following are the points where Hazelcast scores over Redis:

- **Built for Distributed Environment from ground-up**: Unlike Redis, which started as single machine cache, Hazelcast, from the very beginning, has been built for distributed environment.

- **Simple cluster scale in/out**: Maintaining a cluster where nodes are added or removed is very simple in case of Hazelcast, for example, adding a node is a matter of launching the node with the required configuration. Removing a node requires simple shutting down of the node. Hazelcast automatically handles partitioning of data, etc. Having the same setup for Redis and performing the above operation requires more precaution and manual efforts.

- **Less resources needs to support failover**: Redis follows master-slave approach. For failover, Redis requires additional resources to setup **Redis Sentinel**. These Sentinel nodes are responsible to elevate a slave to master if the original master node goes down. In Hazelcast, all nodes are treated equal, failure of a node is detected by other nodes. So, the case of a node going down is handled pretty transparently and that too without any additional set of monitoring servers.

- **Simple Distributed Compute**: Hazelcast, with its **EntryProcessor**, provides a simple interface to send the code to the data for parallel processing. This reduces data transfer over the wire. Redis also supports this, however, achieving this requires one to be aware of **Lua** scripting which adds additional learning curve.

Hazelcast – Setup

Hazelcast requires Java 1.6 or above. Hazelcast can also be used with .NET, C++, or other JVM based languages like Scala and Clojure. However, for this tutorial, we are going to use Java 8.

Before we move on, following is the project setup that we will use for this tutorial.

```
hazelcast/
├── com.example.demo/
│   ├── SingleInstanceHazelcastExample.java
│   ├── MultiInstanceHazelcastExample.java
│   ├── Server.java
│   └── ....
├── pom.xml
├── target/
├── hazelcast.xml
├── hazelcast-multicast.xml
├── ...
```

For now, we can just create the package, i.e., `com.example.demo` inside the hazelcast directory. Then, just `cd` to that directory. We will look at other files in the upcoming sections.

Installing Hazelcast

Installing Hazelcast simply involves adding a JAR file to your build file. POM file or `build.gradle` based on whether you are using Maven or Gradle respectively.

If you are using Gradle, adding the following to `build.gradle` file would be enough:

```
dependencies {
compile "com.hazelcast:hazelcast:3.12.12”
}
```

POM for the tutorial

We will use the following POM for our tutorial:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>1.0.0</modelVersion>
	<groupId>com.example</groupId>
	<artifactId>demo</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>demo</name>
	<description>Demo project for Hazelcast</description>

	<properties>
		<maven.compiler.source>1.8</maven.compiler.source>
		<maven.compiler.target>1.8</maven.compiler.target>
	</properties>

	<dependencies>
		<dependency>
			<groupId>com.hazelcast</groupId>
			<artifactId>hazelcast</artifactId>
			<version>3.12.12</version>
		</dependency>
	</dependencies>

    <!-- Below build plugin is not needed for Hazelcast, it is being used only
to created a shaded JAR so that -->
    <!-- using the output i.e. the JAR becomes simple for testing snippets in
the tutorial-->
	<build>
```

```
            <plugins>
                  <plugin>
                        <!-- Create a shaded JAR and specify the entry point class-->
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-shade-plugin</artifactId>
                        <version>3.2.4</version>
                        <executions>
                              <execution>
                                    <phase>package</phase>
                                    <goals>
                                          <goal>shade</goal>
                                    </goals>
                              </execution>
                        </executions>
                  </plugin>
            </plugins>
      </build>
</project>
```

Hazelcast – First Application

Hazelcast can be run in isolation (single node) or multiple nodes can be run to form a cluster. Let us first try starting a single instance.

Single Instance

Now, let us try creating and using a single instance of Hazelcast cluster. For that, we will create `SingleInstanceHazelcastExample.java` file.

```
package com.example.demo;

import java.util.Map;

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

public class SingleInstanceHazelcastExample {

    public static void main(String... args){

        //initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        System.out.println("Hello world");

        // perform a graceful shutdown
        hazelcast.shutdown();
    }
}
```

Now let's compile the code and execute it:

```
mvn clean install
java -cp target/demo-0.0.1-SNAPSHOT.jar
com.example.demo.SingleInstanceHazelcastExample
```

If you execute above code, the output would be:

```
Hello World
```

However, more importantly, you will also notice log lines from Hazelcast which signifies that Hazelcast has started. Since we are running this code only once, i.e., a single JVM, we would only have one member in our cluster.

```
Jan 30, 2021 10:26:51 AM com.hazelcast.config.XmlConfigLocator
INFO: Loading 'hazelcast-default.xml' from classpath.
Jan 30, 2021 10:26:51 AM com.hazelcast.instance.AddressPicker
INFO: [LOCAL] [dev] [3.12.12] Prefer IPv4 stack is true.
Jan 30, 2021 10:26:52 AM com.hazelcast.instance.AddressPicker
INFO: [LOCAL] [dev] [3.12.12] Picked [localhost]:5701, using socket
ServerSocket[addr=/0:0:0:0:0:0:0:0,localport=5701], bind any local is true
Jan 30, 2021 10:26:52 AM com.hazelcast.system
...

Members {size:1, ver:1} [
      Member [localhost]:5701 - 9b764311-9f74-40e5-8a0a-85193bce227b this
]

Jan 30, 2021 10:26:56 AM com.hazelcast.core.LifecycleService
INFO: [localhost]:5701 [dev] [3.12.12] [localhost]:5701 is STARTED
...

You will also notice log lines from Hazelcast at the end which signifies
Hazelcast was shutdown:
INFO: [localhost]:5701 [dev] [3.12.12] Hazelcast Shutdown is completed in 784 ms.
```

```
Jan 30, 2021 10:26:57 AM com.hazelcast.core.LifecycleService
INFO: [localhost]:5701 [dev] [3.12.12] [localhost]:5701 is SHUTDOWN
```

Cluster: Multi Instance

Now, let's create `MultiInstanceHazelcastExample.java` file (as below) which would be used for multi-instance cluster.

```
package com.example.demo;

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

public class MultiInstanceHazelcastExample {

    public static void main(String... args) throws InterruptedException{

        //initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();


        //print the socket address of this member and also the size of the cluster
        System.out.println(String.format("[%s]: No. of hazelcast members: %s",
                hazelcast.getCluster().getLocalMember().getSocketAddress(),
                hazelcast.getCluster().getMembers().size()));

        // wait for the member to join
        Thread.sleep(30000);

        //perform a graceful shutdown
        hazelcast.shutdown();
    }
}
```

Let's execute the following command on **two different shells**:

```
java -cp .\target\demo-0.0.1-SNAPSHOT.jar
com.example.demo.MultiInstanceHazelcastExample
```

You would notice on the **1st shell** that a Hazelcast instance has been started and a member has been assigned. Note the last line of output which says that there is a **single member using port 5701**.

```
Jan 30, 2021 12:20:21 PM com.hazelcast.internal.cluster.ClusterService
INFO: [localhost]:5701 [dev] [3.12.12]

Members {size:1, ver:1} [
        Member [localhost]:5701 - b0d5607b-47ab-47a2-b0eb-6c17c031fc2f this
]

Jan 30, 2021 12:20:21 PM com.hazelcast.core.LifecycleService
INFO: [localhost]:5701 [dev] [3.12.12] [localhost]:5701 is STARTED
[/localhost:5701]: No. of hazelcast members: 1
```

You would notice on the **2nd shell** that a Hazelcast instance has joined the 1st instance. Note the last line of the output which says that there are now **two members using port 5702**.

```
INFO: [localhost]:5702 [dev] [3.12.12]

Members {size:2, ver:2} [
        Member [localhost]:5701 - b0d5607b-47ab-47a2-b0eb-6c17c031fc2f
        Member [localhost]:5702 - 037b5fd9-1a1e-46f2-ae59-14c7b9724ec6 this
]

Jan 30, 2021 12:20:46 PM com.hazelcast.core.LifecycleService
INFO: [localhost]:5702 [dev] [3.12.12] [localhost]:5702 is STARTED
[/localhost:5702]: No. of hazelcast members: 2
```

Hazelcast – Configuration

Hazelcast supports programmatic as well as XML-based configuration. However, it is the XML configuration which is heavily used in production, given its ease of use. But XML configuration internally uses the Programmatic configuration.

XML Configuration

The `hazelcast.xml` is where these configurations need to be placed. The file is searched for in the following location (in same order) and is chosen from the first available location:

- Passing the location of the XML to the JVM via the system property - `Dhazelcast.config=/path/to/hazelcast.xml`
- `hazelcast.xml` in the current working directory
- `hazelcast.xml` in the classpath
- default `hazelcast.xml` provided by Hazelcast

Once the XML is found, Hazelcast would load the required configuration from the XML file.

Let's try that out with an example. Create an XML in your current directory with the name `hazelcast.xml`.

```
<hazelcast
      xsi:schemaLocation="http://www.hazelcast.com/schema/config
http://www.hazelcast.com/schema/config/hazelcast-config-3.12.12.xsd"
      xmlns="http://www.hazelcast.com/schema/config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

       <!-- name of the instance -->
      <instance-name>XML_Hazelcast_Instance</instance-name>

</hazelcast>
```

The XML as of now only contains the schema location of the Hazelcast XML which is used for validation. But more importantly, it contains the instance name.

Now create an `XMLConfigLoadExample.java` file with the following content.

```
package com.example.demo;

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

public class XMLConfigLoadExample {

    public static void main(String... args) throws InterruptedException{

        //initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();



        //specified the name written in the XML file
        System.out.println(String.format("Name of the instance:
%s",hazelcast.getName()));



        //perform a graceful shutdown
        hazelcast.shutdown();
    }
}
```

Execute the above Java file with the following command:

```
java  -Dhazelcast.config=hazelcast.xml  -cp .\target\demo-0.0.1-SNAPSHOT.jar
com.example.demo.XMLConfigLoadExample
```

The output for above command would be:

```
Jan 30, 2021 1:21:41 PM com.hazelcast.config.XmlConfigLocator
INFO: Loading configuration hazelcast.xml from System property
'hazelcast.config'
Jan 30, 2021 1:21:41 PM com.hazelcast.config.XmlConfigLocator
INFO: Using configuration file at C:\Users\demo\eclipse-
workspace\hazelcast\hazelcast.xml
...
Members {size:1, ver:1} [
        Member [localhost]:5701 - 3d400aed-ddb9-4e59-9429-3ab7773e7e09 this
]

Name of cluster: XML_Hazelcast_Instance
```

As you see, Hazelcast loaded the configuration and printed the name which was specified in the configuration (last line).

There are a whole lot of configuration options which can be specified in the XML. The complete list can be found at:

https://github.com/hazelcast/hazelcast/blob/master/hazelcast/src/main/resources/hazelcast-full-example.xml

We will see a few of these configurations as we move along the tutorial.

Programmatic Configuration

As stated earlier, XML configuration is ultimately done via programmatic configuration. So, let's try programmatic configuration for the same example which we saw in XML configuration. For that, let's create the `ProgramaticConfigLoadExample.java` file with the following content.

```
package com.example.demo;


import com.hazelcast.config.Config;
import com.hazelcast.core.Hazelcast;
```

```
import com.hazelcast.core.HazelcastInstance;

public class ProgramaticConfigLoadExample {

    public static void main(String... args) throws InterruptedException {
        Config config = new Config();
        config.setInstanceName("Programtic_Hazelcast_Instance");

        // initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance(config);

        // specified the name written in the XML file
        System.out.println(String.format("Name of the instance: %s",
hazelcast.getName()));

        // perform a graceful shutdown
        hazelcast.shutdown();
    }
}
```

Let's execute the code without passing any hazelcast.xml file by:

```
java -cp .\target\demo-0.0.1-SNAPSHOT.jar
com.example.demo.ProgramaticConfigLoadExample
```

The output of the above code is:

```
Name of the instance: Programtic_Hazelcast_Instance
```

Logging

To avoid dependencies, Hazelcast by default uses JDK based logging. But it also supports logging via **slf4j**, **log4j**. For example, if we want to setup logging via for sl4j with logback, we can update the POM to contain the following dependencies:

```
    <!-- contains both sl4j bindings and the logback core -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
    </dependency>
```

Define a configuration logback.xml file and add it to your classpath, for example, `src/main/resources`.

```
<configuration>
     <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
           <encoder>
                 <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
           </encoder>
     </appender>

     <root level="info">
           <appender-ref ref="STDOUT" />
     </root>

     <logger name="com.hazelcast" level="error">
           <appender-ref ref="STDOUT" />
     </logger>
</configuration>
```

Now, when we execute the following command, we notice that all the meta information about the Hazelcast member creation etc. is not printed. And this is because we have set the logging level for Hazelcast to error and asked Hazelcast to use sl4j logger.

```
java  -Dhazelcast.logging.type=slf4j -cp .\target\demo-0.0.1-SNAPSHOT.jar
com.example.demo.SingleInstanceHazelcastExample
```

Output

```
John
```

Variables

Value written to XML configuration files can vary based on the environment. For example, in production, you may use a different username/password for connecting to the Hazelcast cluster compared to the dev environment. Instead of maintaining separate XML files, one can also write variables in the XML files and then pass those variables via command line or programmatically to Hazelcast. Here is an example for choosing the name of the instance from the command line.

So, here is our XML file with the variable `${varname}`

```
<hazelcast
      xsi:schemaLocation="http://www.hazelcast.com/schema/config
http://www.hazelcast.com/schema/config/hazelcast-config-3.12.12.xsd"
      xmlns="http://www.hazelcast.com/schema/config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <instance-name>${instance_name}</instance-name>
</hazelcast>
```

And here is the sample Java code we would use to print the variable value:

```
package com.example.demo;

import java.util.Map;

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

public class XMLConfigLoadWithVariable {

    public static void main(String... args) throws InterruptedException {
        // initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // specified the name written in the XML file
        System.out.println(String.format("Name of the instance: %s",
hazelcast.getName()));

        // perform a graceful shutdown
        hazelcast.shutdown();
    }
}
```

And, following is the command:

```
java  -Dhazelcast.config=others\hazelcast.xml -Dinstance_name=dev_cluster -cp
.\target\demo-0.0.1-SNAPSHOT.jar com.example.demo.XMLConfigLoadWithVariable
```

And the output shows that the variable was replaced by Hazelcast correctly.

```
Name of the instance: dev_cluster
```

Hazelcast – Setting up multi-node instances

Given that Hazelcast is a distributed IMDG and typically is set up on multiple machines, it requires access to the internal/external network. The most important use-case being discovery of Hazelcast nodes within a cluster.

Hazelcast requires the following ports:

- 1 inbound port to receive pings/data from other Hazelcast nodes/clients
- *n* number of outbound ports which are required to send ping/data to other members of the cluster.

This node discovery happens in few ways:

- Multicast
- TCP/IP
- Amazon EC2 auto discovery

Of this, we will look at Multicast and TCP/IP

Multicast

Multicast joining mechanism is enabled by default. https://en.wikipedia.org/wiki/Multicast is a way of communication form in which message is transmitted to all the nodes in a group. And this is what Hazelcast uses to discover other members of the cluster. All the examples that we have looked at earlier use multicast to discover members.

Let's now explicitly turn it on. Save the following in `hazelcast-multicast.xml`

```
<hazelcast
      xsi:schemaLocation="http://www.hazelcast.com/schema/config
http://www.hazelcast.com/schema/config/hazelcast-config-3.12.12.xsd"
      xmlns="http://www.hazelcast.com/schema/config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <network>
            <join>
```

```
                <multicast enabled="true" />
            </join>
      </network>
</hazelcast>
```

And then, let us execute the following:

```
java  -Dhazelcast.config=hazelcast-multicast.xml -cp .\target\demo-0.0.1-
SNAPSHOT.jar com.example.demo.XMLConfigLoadExample
```

In the output, we notice the following lines from Hazelcast which effectively means that multicast joiner is used to discover the members

```
Jan 30, 2021 5:26:15 PM com.hazelcast.instance.Node
INFO: [localhost]:5701 [dev] [3.12.12] Creating MulticastJoiner
```

Multicast, by default, accepts communication from all the machines in the multicast group. This may be a security concern and that is why typically, on-premise, multicast communication is firewalled. So, while multicast is good for development work, in production, it is best to use TCP/IP based discovery.

TCP/IP

Due to the drawbacks stated for Multicast, TCP/IP is the preferred way for communication. In case of TCP/IP, a member can connect to only known/listed members.

Let's use TCP/IP for discovery mechanisms. Save the following in hazelcast-tcp.xml

```
<hazelcast
      xsi:schemaLocation="http://www.hazelcast.com/schema/config
    http://www.hazelcast.com/schema/config/hazelcast-config-3.12.12.xsd"
      xmlns="http://www.hazelcast.com/schema/config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <network>
            <join>
                  <multicast enabled="false" />
                  <tcp-ip enabled="true">
                        <members>localhost</members>
                  </tcp-ip>
            </join>
      </network>
</hazelcast>
```

And then, let's execute the following command:

```
java  -Dhazelcast.config=hazelcast-tcp.xml -cp .\target\demo-0.0.1-SNAPSHOT.jar
com.example.demo.XMLConfigLoadExample
```

The **output** is following:

```
INFO: [localhost]:5701 [dev] [3.12.12] Creating TcpIpJoiner
Jan 30, 2021 8:09:29 PM
com.hazelcast.spi.impl.operationexecutor.impl.OperationExecutorImpl
```

The above output shows that TCP/IP joiner was use to join two members

And if you execute following command on two different shells:

```
java '-Dhazelcast.config=hazelcast-tcp.xml' -cp .\target\demo-0.0.1-SNAPSHOT.jar
com.example.demo.MultiInstanceHazelcastExample
```

We see the following output:

```
Members {size:2, ver:2} [
        Member [localhost]:5701 - 62eedeae-2701-4df0-843c-7c3655e16b0f
        Member [localhost]:5702 - 859c1b46-06e6-495a-8565-7320f7738dd1 this
]
```

The above output means that the nodes were able to join using TCP/IP and both are using localhost as the IP address.

Note that we can specify more IPs or the machine names (which would be resolved by DNS) in the XML configuration file.

```
<hazelcast
     xsi:schemaLocation="http://www.hazelcast.com/schema/config
    http://www.hazelcast.com/schema/config/hazelcast-config-3.12.12.xsd"
     xmlns="http://www.hazelcast.com/schema/config"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

     <network>
           <join>
                 <multicast enabled="false" />
                 <tcp-ip enabled="true">
```

```
                        <members>machine1, machine2....</members>
                    </tcp-ip>
                </join>
            </network>
</hazelcast>
```

Hazelcast – Data Structures

`java.util.concurrent` package provides data structures such as `AtomicLong`, `CountDownLatch`, `ConcurrentHashMap`, etc. which are useful when you have more than one thread reading/writing data to the data structure. But to provide thread safety, all of these threads are expected to be on a single JVM/machine.

Hazelcast provides a way to distribute your data structure across JVMs/machines.

There are two major benefits of distributing data structure:

- **Better Performance**: If more than one machine has access to the data, all of them can work in parallel and complete the work in a lesser timespan.
- **Data Backup**: If a JVM/machine goes down, we have another JVMs/machines holding the data

IAtomicLong

The Atomic Long data structure in Java provides a thread safe way for using Long.

Similarly, `IAtomicLong` is more of a distributed version of `AtomicLong`. It provides similar functions of which following are useful ones: `set`, `get`, `getAndSet`, `incrementAndGet`. One important point to note here is that the performance of the above functions may not be similar as the data structure is distributed across machines.

AtomicLong has one synchronous backup which means if we have a setup where we have, say, 5 JVMs running, only two JVMs will hold this variable.

Let's look at some of the useful functions.

Initializing and Setting value to IAtomicLong

```
public class Application {

    public static void main(String... args) throws IOException {

        //initialize hazelcast instance and the counter variable
```

```
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();
        IAtomicLong counter = hazelcast.getAtomicLong("counter");
        System.out.println(counter.get());

        counter.set(2);
        System.out.println(counter.get());

        System.exit(0);
    }
}
```

When the above code is executed, it will produce the following output:

```
0
2
```

Synchronization across JVMs

Atomic Long provides concurrency control across JVMs. So, methods like `incrementAndGet`, `compareAndSet` can be used to atomically update the counter

Let's execute code below simultaneously, on two JVMs

```
public class AtomicLong2 {

    public static void main(String... args) throws IOException,
InterruptedException {

        // initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();
        IAtomicLong counter = hazelcast.getAtomicLong("counter");

        for(int i = 0; i < 1000; i++) {
            counter.incrementAndGet();
        }
        System.exit(0);
    }
}
```

The 2nd line of the output of the above code would always be:

```
2000
```

If `incrementAndGet()` would not have been thread safe, the above code may have not given 2000 as the output all the time. It would probably be less than that, as the writes one thread may have gotten overwritten by another.

Useful Methods

Function Name	Description
`get()`	Return the current value
`set(long newValue)`	Set the value to `newValue`
`addAndGet(long value)`	Atomically add the value and return the updated value
`decrementAndGet(long value)`	Atomically subtract the value and return the updated value
`getAndAdd(long value)`	Atomically return the current value and store the sum of current value and the value
`getAndDecrement(long value)`	Atomically return the current value and store the subtraction of value from the current value
`compareAndSet(long expected, long newValue)`	Atomically set value to `newValue` if the `oldValue` is equal to expected value
`decrementAndGet(long value)`	Atomically subtract the value and return the updated value

ILock

The `java.util.concurrent.locks.Lock` provides an interface which can be implemented and used for locking critical sections when working in a multithreaded environment in a JVM.

Similarly, `ILock` extends the interface to provide a distributed version of `Java Lock`. It provides similar functions: `lock`, `unlock`, `tryLock`

But a major difference between `ILock` and `Java Lock` is that while `Java Lock` provides protection of critical section from threads in a single JVM, `ILock` provides synchronization for threads in a single JVM as well as multiple JVMs.

`ILock` has one synchronous backup meaning that if we have a setup where we have, say, 5 JVMs running, only two JVMs will hold this variable.

Let's look at an example of the useful functions.

Acquiring and Releasing Lock

Let's say we execute the following code on two JVMs.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a lock
        ILock hzLock = hazelcast.getLock("lock_1");
        IAtomicLong counter = hazelcast.getAtomicLong("counter");

        // acquire lock
        hzLock.lock();

        System.out.println("Acquiring Lock");

        try{
            Thread.sleep(5000);
            System.out.println("Incrementing Counter");
            counter.incrementAndGet();
            System.out.println("Counter: " + counter.get());
        }
        finally {
            // release lock
            System.out.println("Lock Released");
            hzLock.unlock();
        }
        System.exit(0);
    }
```

The output of the above function shows that the second JVM was able to acquire lock only after first JVM released the lock.

```
Acquired Lock
Incrementing Counter
Counter: 1
Lock Released
Acquired Lock
Incrementing Counter
Counter: 2
Lock Released
```

Using tryLock instead of Lock

To reduce the chances of deadlock, it is recommended to use `tryLock(timeout, unit)` method instead `lock()`. By default, the `lock()` has a timeout of 5 mins and throws `OperationTimeoutException` exception if lock is not acquired in that timespan. `tryLock` instead returns a Boolean based on whether the lock is acquired or not in the provided timespan.

Let's execute the following code on **two JVMs**.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a lock
        ILock hzLock = hazelcast.getLock("lock_1");

        // acquire lock
        if(hzLock.tryLock(2000, TimeUnit.SECONDS)) {
            System.out.println("Acquired Lock");
            Thread.sleep(5000);
            System.out.println("Lock Released");
            hzLock.unlock();
        }
        else
            System.out.println("Couldn't acquire lock");
```

```
        System.exit(0);
    }
```

The **output** for the code would be:

```
Acquired Lock
Couldn't acquire lock
Lock Released
```

Good practices and know-hows

- While acquiring locks can be very useful, it is recommended to keep the critical section as short as possible. This ensures that the performance does not degrade and it also reduces the chances of a deadlock.
- If a member (which has acquired) goes down, the lock is automatically released and is up for grabs for other members.
- The lock is rentrant; it ensures that the same thread can acquire a lock multiple times without causing a deadlock.

Useful Methods

Function Name	Description
`lock()`	Acquire the provided lock instance so that no other thread can acquire it. If unavailable, it waits indefinitely till lock is acquired.
`unlock()`	Release the acquire lock
`tryLock(long time, TimeUnit unit)`	Try to acquire lock in the given time window. Return true if the lock is acquired, else false.
`isLocked()`	Check if the lock is already acquired by some other thread

ISemaphore

The `java.util.concurrent.Semaphore` supports synchronization by providing limited access when working in a multithreaded environment in a JVM.

Sounds a lot like a lock, right? But there are two major differences between lock and semaphore:

- Semaphore does not have ownership. It can be acquired by a thread and released by another thread. Locks are tied to a thread. It needs to be released and acquired by the same thread.

- Semaphore supports entry of 1 of one or more threads into the critical section based on **req** based on available permits.

Similarly, ISemaphore provides a distributed version of Java Semaphore. ISemaphore provides a distributed version of Java Semaphore. It provides similar functions: `acquire`, `release`.

But a major difference between ISemaphore and Java Semaphore is that while the Java Semaphore provides protection of critical section from threads in a single JVM, ISemaphore provides synchronization for threads in a single JVM as well as multiple JVMs.

ISemaphore has one synchronous backup which means, if we have a setup where we have, say, 5 JVMs running, only two JVMs will hold this semaphore.

Acquiring Permit, Releasing Permit

Let's execute the following code on **three JVMs**. The code is supposed to print the number of threads that have acquired semaphore. And we have permit of 2 which means, at a time, only two threads should be permitted to enter the `if` block

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a lock
        ISemaphore hzSemaphore = hazelcast.getSemaphore("semaphore_1");
        IAtomicLong activeThreads = hazelcast.getAtomicLong("threads");

        hzSemaphore.init(2);

        for(int i=0; i< 10; i++) {
```

```
            if(hzSemaphore.tryAcquire(2000, TimeUnit.MILLISECONDS));
            {
                System.out.println("Thread count: " +
activeThreads.incrementAndGet());
                Thread.sleep(2000);
                hzSemaphore.release();
                activeThreads.decrementAndGet();
            }
        }

        System.exit(0);
    }
```

The output for the code shows that we have 1 or 2 threads active which is what we expect given the permit being set to 2.

Good Practices

- If a member which has permit goes down, the permit is released automatically, making it available for other threads to acquire.

- Avoid using `acquire`() of semaphore, as it is a blocking call which may lead to deadlock. It's better to use `tryAcquire`() with a timeout to avoid blocking.

Useful Methods

Function Name	Description
`acquire()`	Acquire the permit if available. If unavailable, it waits indefinitely till the permit is available.
`release()`	Release the acquired permit.
`tryAcquire(long time, TimeUnit unit)`	Try to acquire a permit in the given time window. Return true if the permit is acquired, else false.
`availablePermits()`	Return the number of permits which are available with this ISemaphore instance

ICountDownLatch

The `java.util.concurrent.CountDownLatch` provides a way for threads to wait, while other threads complete a set of operations in a multithreaded environment in a JVM.

Similarly, `ICountDownLatch` provides a distributed version of `Java CountDownLatch`. It provides similar functions: `setCount`, `countDown`, `await,` etc.

A major difference between `ICountDownLatch` and `Java CountDownLatch` is that while `Java CountDownLatch` provides protection of critical section from threads in a single JVM, `ICountDownLatch` provides synchronization for threads in a single JVM as well as multiple JVMs.

`ICountDownLatch` has one synchronous backup which means if we have a setup where we have, say, 5 JVMs running, only two JVMs will hold this latch.

Setting Latch & Awaiting Latch

Let's execute the following code on **two JVMs**. The master code on one and worker code on other. The code is supposed to make the worker thread wait till the master thread completes.

The first piece is the master code which creates a latch and counts it down.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a lock
        ICountDownLatch countDownLatch =
hazelcast.getCountDownLatch("count_down_1");
        System.out.println("Setting counter");
        countDownLatch.trySetCount(2);
        Thread.sleep(2000);
        System.out.println("Counting down");
        countDownLatch.countDown();
        Thread.sleep(2000);
        System.out.println("Counting down");
        countDownLatch.countDown();

        System.exit(0);
    }
```

The second piece is of worker code which creates a latch and counts it down.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a lock
        ICountDownLatch countDownLatch =
hazelcast.getCountDownLatch("count_down_1");
        countDownLatch.await(5000, TimeUnit.MILLISECONDS);
        System.out.println("Worker successful");

        System.exit(0);
    }
```

The output for the code shows that the worker prints only after the countdown was completed to 0.

```
Setting counter
Counting down
Counting down
Worker successful
```

Useful Methods

Function Name	Description
`await()`	Wait for the latch's count to reach to zero before proceeding
`countDown()`	Decrement the countdown latch
`trySetCount(int count)`	Set the count of the latch
`getCount()`	Get the current count of the latch

ISet

The `java.util.Set` provides an interface for holding collections of objects which are unique. The ordering of elements does not matter.

Similarly, `ISet` implements a distributed version of `Java Set`. It provides similar functions: `add`, `forEach`, etc.

One important point to note about `ISet` is that, unlike other collection data, it is not partitioned. All the data is stored/present on a single JVM. Data is still accessible to all JVMs, but the set cannot be scaled beyond a single machine/JVM.

The set supports synchronous backup as well as asynchronous backup. Synchronous backup ensures that even if the JVM holding the set goes down, all elements would be preserved and available from the backup.

Let's look at an example of the useful functions.

Adding elements and reading elements

Let's execute the following code on 2 JVMs. The producer code on one and consumer code on other.

The first piece is the producer code which creates a set and adds item to it.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a set
        ISet<String> hzFruits = hazelcast.getSet("fruits");

        hzFruits.add("Mango");
        hzFruits.add("Apple");
        hzFruits.add("Banana");

        // adding an existing fruit
        System.out.println(hzFruits.add("Apple"));

        System.out.println("Size of set:" + hzFruits.size());

        System.exit(0);
    }
```

The second piece is of consumer code which reads set elements.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a set
        ISet<String> hzFruits = hazelcast.getSet("fruits");
        Thread.sleep(2000);
        hzFruits.forEach(System.out::println);

        System.exit(0);
    }
```

The output for the code for the producer shows that it is not able to add an existing element.

```
false
3
```

The output for the code for the consumer prints set size and the fruits which are can be in a different order.

```
3
Banana
Mango
Apple
```

Useful Methods

Function Name	Description
`add(Type element)`	Add element to the set if not already present
`remove(Type element)`	Remove element from the set
`size()`	Return the count of elements in the set
`contains(Type element)`	Return if the element is present

`getPartitionKey()`	Return the partition key which hold the set
`addItemListener(ItemListener<Type> listener, value)`	Notifies the subscriber of an element being removed/added/modified in the set.

IList

The `java.util.List` provides an interface for holding collections of objects that do not necessarily need to be unique. The ordering of elements does not matter.

Similarly, `IList` implements a distributed version of Java List. It provides similar functions: `add`, `forEach`, etc.

All the data which is present in `IList` is stored/present on a single JVM. Data is still accessible to all the JVMs, but the list cannot be scaled beyond a single machine/JVM.

The list supports synchronous backup as well as asynchronous backup. Synchronous backup ensures that even if the JVM holding the list goes down, all the elements would be preserved and available from the backup.

Let's look at an example of the useful functions.

Adding elements and reading elements

Let's execute the following code on 2 JVMs. The producer code on one and consumer code on other.

The first piece is the producer code which creates a list and adds item to it.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a list
        IList<String> hzFruits = hazelcast.getList("fruits");

        hzFruits.add("Mango");
        hzFruits.add("Apple");
        hzFruits.add("Banana");
```

```
        // adding an existing fruit
        System.out.println(hzFruits.add("Apple"));

        System.out.println("Size of list:" + hzFruits.size());

        System.exit(0);
    }
```

The second piece is of consumer code which reads the list elements.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a list
        IList<String> hzFruits = hazelcast.getList("fruits");
        Thread.sleep(2000);
        hzFruits.forEach(System.out::println);

        System.exit(0);
    }
```

The output for the code for the producer shows that it is not able to add an existing element.

```
true
4
```

The output for the code for the consumer prints the list size and the fruits are in expected order.

```
4
Mango
Apple
Banana
Apple
```

Useful Methods

Function Name	Description
`add(Type element)`	Add element to the list
`remove(Type element)`	Remove element from the list
`size()`	Return the count of elements in the list
`contains(Type element)`	Return if the element is present
`getPartitionKey()`	Return the partition key which holds the list
`addItemListener(ItemListener<Type> listener, value)`	Notifies the subscriber of an element being removed/added/modified in the list.

IQueue

The `java.util.concurrent.BlockingQueue` provides an interface which supports threads in a JVM to produce and consume messages at different rates. The producer blocks based on available capacity and the consumer blocks for the element to be available in the queue.

Similarly, `IQueue` extends the `BlockingQueue` and makes it a distributed version of it. It provides similar functions: `put`, `take`, etc.

One important point to note about `IQueue` is that, unlike other collections, data is not partitioned. All the data is stored/present on a single JVM. Data is still accessible to all the JVMs, but the queue cannot be scaled beyond a single machine/JVM. If the number of elements increases beyond available memory, an `OutOfMemoryException` is thrown.

The queue supports synchronous backup as well as asynchronous backup. Synchronous backup ensures that even if the JVM holding the queue goes down, all the elements would be preserved and available from the backup.

Let's look at an example of the useful functions.

Adding elements and reading elements

Let’s execute the following code on 3 JVMs. The producer code on one and 2 consumers code on others.

The first piece is the producer code which creates a queue and adds item to it.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a queue
        IQueue<String> hzFruits = hazelcast.getQueue("fruits");

        String[] fruits = {"Mango", "Apple", "Banana", "Watermelon"};

        for (String fruit : fruits) {
            System.out.println("Producing: " + fruit);
            Thread.sleep(1000);
        }

        System.exit(0);
    }
```

The second piece is of consumer code which reads the elements.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        IQueue<String> hzFruits = hazelcast.getQueue("fruits");

        while(!hzFruits.isEmpty()) {
            System.out.println("Consuming: " + hzFruits.take());
            Thread.sleep(2000);
        }

        System.exit(0);
    }
```

The output for the code for the producer shows that it is not able to add an existing element.

```
Producing Mango
Producing Apple
Producing Banana
Producing Watermelon
```

The output for the code for the first consumer shows that it consumes some part of the data.

```
Consuming Mango
Consuming Banana
```

The output for the code for the second consumer shows that it consumes the other part of the data:

```
Consuming Apple
Consuming Watermelon
```

Useful Methods

Function Name	Description
`add(Type element)`	Add an element to the list
`remove(Type element)`	Remove an element from the list
`poll()`	Return the head of the queue or returns NULL if the queue is empty
`take()`	Return the head of the queue or wait till the element becomes available
`size()`	Return the count of elements in the list
`contains(Type element)`	Return if the element is present
`getPartitionKey()`	Return the partition key which holds the list
`addItemListener(ItemListener<Type> listener, value)`	Notifies the subscriber of an element being removed/added/modified in the list.

IMap

The `java.util.concurrent.Map` provides an interface which supports storing key value pair in a single JVM. While `java.util.concurrent.ConcurrentMap` extends this to support thread safety in a single JVM with multiple threads.

Similarly, `IMap` extends the `ConcurrentHashMap` and provides an interface which makes the map thread safe across JVMs. It provides similar functions: put, get etc.

The `IMap` supports synchronous backup as well as asynchronous backup. Synchronous backup ensures that even if the JVM holding the queue goes down, all elements would be preserved and available from the backup.

Let's look at an example of the useful functions.

Creation & Read/Write

Adding elements and reading elements. Let's execute the following code on two JVMs. The producer code on one and one consumer code on the other.

The first piece is the producer code which creates a map and adds item to it.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a map
        IMap<String, String> hzStock = hazelcast.getMap("stock");

        hzStock.put("Mango", "4");
        hzStock.put("Apple", "1");
        hzStock.put("Banana", "7");
        hzStock.put("Watermelon", "10");

        Thread.sleep(5000);

        System.exit(0);
    }
```

The second piece is of consumer code which reads the elements.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a map
        IMap<String, String> hzStock = hazelcast.getMap("stock");

        for(Map.Entry<String, String> entry: hzStock.entrySet()){
            System.out.println(entry.getKey() + ":" + entry.getValue());
        }

        Thread.sleep(5000);

        System.exit(0);
    }
```

The output for the code for the consumer:

```
Mango:4
Apple:1
Banana:7
Watermelon:10
```

Useful Methods

Function Name	Description
`put(K key, V value)`	Add an element to the map
`remove(K key)`	Remove an element from the map
`keySet()`	Return a copy of all the keys in the map
`localKeySet()`	Return a copy of all keys which are present in the local partition
`values()`	Return a copy of all the values in the map

`size()`	Return the count of elements in the map
`containsKey(K key)`	Return true if the key is present
`executeOnEnteries(EntryProcessor processor)`	Applies the processor on all the map's keys and returns the output of this application. We will look at an example for the same in the upcoming section.
`addEntryListener(EntryListener listener, value)`	Notifies the subscriber of an element being removed/added/modified in the map.
`addLocalEntryListener(EntryListener listener, value)`	Notifies the subscriber of an element being removed/added/modified in the local partitions

Eviction

By default, keys in Hazelcast stay indefinitely in the IMap. If we have a very large set of keys, then we need to ensure that the keys which are heavily used are stored in the IMap as compared to the ones which are used less often, in order to have better performance and efficient memory usage.

For this purpose, one can manually delete keys via `remove()`/`evict()` functions for the keys which are not used that often. However, Hazelcast also provides automatic eviction of keys based on various eviction algorithms.

This policy can be set by XML or programmatically. Let's look at an example for the same:

```
<map name="stock">
    <max-size policy="FREE_HEAP_PERCENTAGE">30</max-size>
    <eviction-policy>LFU</eviction-policy>
</map>
```

There are two attributes in the above configuration.

- **`Max-size`**: Policy which is used to communicate to Hazelcast the limit at which we claim that max size of the map "stock" has reached.
- **`Eviction-policy`**: Once the above max-size policy is hit, what algorithm to use to remove/evict the key.

Here are some of the useful max_size policy.

Max Size Policy	Description
PER_NODE	Max number of entries per JVM for the map which is the default policy.
FREE_HEAP	Minimum free heap memory to be kept aside (in MBytes) in the JVM
FREE_HEAP_PERCENTAGE	Minimum free heap memory to be kept aside (in percent) in the JVM
USED_HEAP	Maximum allowed heap memory used in the JVM (in MBytes)
USED_HEAP_PERCENTAGE	Maximum allowed heap memory used in the JVM (in percent)

Here are some of the useful eviction policy:

Eviction Policy	Description
NONE	No eviction will be made which is the default policy
LFU	Least frequently used would be evicted
LRU	Least recently used key would be evicted

Another useful parameter for eviction is also **time-to-live-seconds**, i.e., TTL. With this, we can ask Hazelcast to remove any key which is older than X seconds. This ensures that we are proactive in removing older keys before max-size policy is hit.

Partitioned data and High Availability

One important point to note about IMap is that unlike other collections, the data is partitioned across JVMs. All the data doesn't need to be stored/present on a single JVM. Complete data is still accessible to all JVMs. This gives Hazelcast a way to scale linearly across available JVMs and not be constrained by memory of a single JVM.

The IMap instances are divided into multiple partitions. By default, the map is divided into 271 partitions. And these partitions are distributed across Hazelcast members available. Each entry in which is added to the map is stored in a single partition.

Let's execute this code on 2 JVMs.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a map
        IMap<String, String> hzStock = hazelcast.getMap("stock");

        hzStock.put("Mango", "4");
        hzStock.put("Apple", "1");
        hzStock.put("Banana", "7");
        hzStock.put("Watermelon", "10");

        Thread.sleep(5000);

        // print the keys which are local to these instance
        hzStock.localKeySet().forEach(System.out::println);

        System.exit(0);
    }
```

As seen in the following output, the consumer 1 prints its own partition which contains 2 keys:

```
Mango
Watermelon
```

Consumer 2 owns the partition which has the other 2 keys:

```
Banana
Apple
```

By default, IMap has one synchronous backup, which means that even if one node/member goes down, the data would not get lost. There are two types of back up.

- **Synchronous**: The `map.put(key, value)` would not succeed till the key is also backed up on another node/member. Sync backups are blocking and thus impact the performance of the put call.

- **Async**: The backup of the stored key is performed eventually. Async backups are non-blocking and fast but they do not guarantee existence of the data if a member were to go down.

The value can be configured using XML configuration. For example, let's do it for out stock map:

```
<map name="stock">
<backup-count>1</backup-count>
<async-backup-count>1<async-backup-count>
</map>
```

Hashcode and Equals

In Java-based HashMap, key comparison happens by checking equality of the `hashCode()` and `equals()` method. For example, a vehicle may have `serialId` and the `model` to keep it simple.

```
public class Vehicle implements Serializable{
    private static final long serialVersionUID = 1L;
    private int serialId;
    private String model;

    public Vehicle(int serialId, String model) {
        super();
        this.serialId = serialId;
        this.model = model;
    }

    public int getId() {
        return serialId;
    }

    public String getModel() {
        return model;
    }

    @Override
    public int hashCode() {
```

```
        final int prime = 31;
        int result = 1;
        result = prime * result + serialId;
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Vehicle other = (Vehicle) obj;
        if (serialId != other.serialId)
            return false;
        return true;
    }
}
```

When we try using the above class as the key for HashMap and IMap, we see the difference in comparison.

```
    public static void main(String... args) throws IOException,
InterruptedException {

        // create a Java based hash map
        Map<Vehicle, String> vehicleOwner = new HashMap<>();
        Vehicle v1 = new Vehicle(123, "Honda");
        vehicleOwner.put(v1, "John");

        Vehicle v2 = new Vehicle(123, null);
        System.out.println(vehicleOwner.containsKey(v2));
```

```
      // create a hazelcast map
       HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();
       IMap<Vehicle, String> hzVehicleOwner = hazelcast.getMap("owner");

       hzVehicleOwner.put(v1, "John");
       System.out.println(hzVehicleOwner.containsKey(v2));

       System.exit(0);
   }
```

Now, why does Hazelcast give the answer as false?

Hazelcast serializes the key and stores it as a byte array in binary format. As these keys are serialized, the comparison cannot be made based on `equals`() and `hashcode`().

Serializing and Deserializing are required in case of Hazelcast because the function `get`(), `containsKey`(), etc. may be invoked on the node which does not own the key, so remote call is required.

Serializing and Deserializng are expensive operations and so, instead of using `equals`() method, Hazelcast compares byte arrays.

What this means is that all the attributes of the `Vehicle` class should match not just id. So, let's execute the following code:

```
    public static void main(String... args) throws IOException,
InterruptedException {

       Vehicle v1 = new Vehicle(123, "Honda");

       // create a hazelcast map
       HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();
       IMap<Vehicle, String> hzVehicleOwner = hazelcast.getMap("owner");

       Vehicle v3 = new Vehicle(123, "Honda");
       System.out.println(hzVehicleOwner.containsKey(v3));

       System.exit(0);
```

```
    }
```

The **output** of the above code is:

```
true
```

This output means all the attributes of `Vehicle` should match for equality.

EntryProcessor

EntryProcessor is a construct which supports sending of code to the data instead of bringing data to the code. It supports serializing, transferring, and the execution of function on the node which owns the IMap keys instead of bringing in the data to the node which initiates the execution of the function.

Let's understand this with an example. Let's say we create an IMap of `Vehicle -> Owner`. And now, we want to store lowercase for the owner. So, how do we do that?

```
    public static void main(String... args) throws IOException,
InterruptedException {

        // create a hazelcast map
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();
        IMap<Vehicle, String> hzVehicleOwner = hazelcast.getMap("owner");

        hzVehicleOwner.put(new Vehicle(123, "Honda"), "John");
        hzVehicleOwner.put(new Vehicle(23, "Hyundai"), "Betty");
        hzVehicleOwner.put(new Vehicle(103, "Mercedes"), "Jane");

        for(Map.Entry<Vehicle, String> entry: hzVehicleOwner.entrySet())
            hzVehicleOwner.put(entry.getKey(), entry.getValue().toLowerCase());

        for(Map.Entry<Vehicle, String> entry: hzVehicleOwner.entrySet())
            System.out.println(entry.getValue());

        System.exit(0);
    }
```

The **output** of the above code is:

```
john
jane
```

```
betty
```

While this code seems simple, it has a major drawback in terms of scale if there are high number of keys:

- Processing would happen on the single/caller node instead of being distributed across nodes.
- More time as well as memory would be needed to get the key information on the caller node.

That is where the EntryProcessor helps. We send the function of converting to lowercase to each node which holds the key. This makes the processing parallel and keeps the memory requirements in check.

```
   public static void main(String... args) throws IOException,
InterruptedException {

        // create a hazelcast map
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();
        IMap<Vehicle, String> hzVehicleOwner = hazelcast.getMap("owner");

        hzVehicleOwner.put(new Vehicle(123, "Honda"), "John");
        hzVehicleOwner.put(new Vehicle(23, "Hyundai"), "Betty");
        hzVehicleOwner.put(new Vehicle(103, "Mercedes"), "Jane");

        hzVehicleOwner.executeOnEntries(new OwnerToLowerCaseEntryProcessor());

        for(Map.Entry<Vehicle, String> entry: hzVehicleOwner.entrySet())
            System.out.println(entry.getValue());

        System.exit(0);
    }

    static class OwnerToLowerCaseEntryProcessor extends
AbstractEntryProcessor<Vehicle, String> {
        @Override
        public Object process(Map.Entry<Vehicle, String> entry) {
            String ownerName = entry.getValue();
            entry.setValue(ownerName.toLowerCase());
            return null;
        }
    }
```

The **output** of the above code is:

```
john
jane
betty
```

Hazelcast – Client

Hazelcast clients are the lightweight clients to Hazelcast members. Hazelcast members are responsible to store data and the partitions. They act like the server in the traditional client-server model.

Hazelcast clients are created only for accessing data stored with Hazelcast members of the cluster. They are not responsible to store data and do not take any ownership to store data.

The clients have their own life cycle and do not affect the Hazelcast member instances.

Let's first create `Server.java` and run it.

```
import java.util.Map;

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

public class Server {

    public static void main(String... args){

        //initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        //create a simple map
        Map<String, String> vehicleOwners = hazelcast.getMap("vehicleOwnerMap");

        // add key-value to map
        vehicleOwners.put("John", "Honda-9235");

        // do not shutdown, let the server run
        //hazelcast.shutdown();
    }
}
```

Now, run the above class.

```
java  -cp .\target\demo-0.0.1-SNAPSHOT.jar com.example.demo.Server
```

For setting up a client, we also need to add client jar.

```
    <dependency>
        <groupId>com.hazelcast</groupId>
        <artifactId>hazelcast-client</artifactId>
        <version>3.12.12</version>
    </dependency>
```

Let's now create `Client.java`. Note that similar to Hazelcast members, clients can also be configured programmatically or via XML configuration (i.e., via `-Dhazelcast.client.config` or `hazelcast-client.xml`).

Let's use the default configuration which means our client would be able to connect to local instances.

```
import java.util.Map;

import com.hazelcast.client.HazelcastClient;
import com.hazelcast.core.HazelcastInstance;

public class Client {

    public static void main(String... args){

        //initialize hazelcast client
        HazelcastInstance hzClient = HazelcastClient.newHazelcastClient();

        //read from map
        Map<String, String> vehicleOwners = hzClient.getMap("vehicleOwnerMap");
        System.out.println(vehicleOwners.get("John"));

        System.out.println("Member of cluster: " +
hzClient.getCluster().getMembers());

        // perform shutdown
        hzClient.getLifecycleService().shutdown();
    }
}
```

Now, run the above class.

```
java  -cp .\target\demo-0.0.1-SNAPSHOT.jar com.example.demo.Client
```

It will produce the following output:

```
Honda-9235
Member of cluster: [Member [localhost]:5701 - a47ec375-3105-42cd-96c7-fc5eb382e1b0]
```

As seen from the output:

- The cluster only contains 1 member which is from Server.java.
- The client is able to access the map which is stored inside the server.

Load Balancing

Hazelcast Client supports load balancing using various algorithms. Load balancing ensures that the load is shared across members and no single member of the cluster is overloaded. The default load balancing mechanism is set to round-robin. The same can be changed by using the `loadBalancer` tag in the `config`.

We can specify the type of load balancer using the load-balancer tag in the configuration. Here is a sample for choosing a strategy that randomly picks up a node.

```
<hazelcast-client xmlns="http://www.hazelcast.com/schema/client-config"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://www.hazelcast.com/schema/client-config
                  http://www.hazelcast.com/schema/client-config/hazelcast-
client-config-4.2.xsd">

     <load-balancer type="random"/>

</hazelcast-client>
```

Failover

In a distributed environment, members can fail arbitrarily. For supporting failover, it is recommended that address to multiple members is provided. If the client gets access to any one

member, that is sufficient for it to get addressed to other members. The parameters `addressList` can be specified in the client configuration.

For example, if we use the following configuration:

```
<hazelcast-client xmlns="http://www.hazelcast.com/schema/client-config"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://www.hazelcast.com/schema/client-config
                  http://www.hazelcast.com/schema/client-config/hazelcast-
client-config-4.2.xsd">

     <address-list>machine1, machine2</address-list>

</hazelcast-client>
```

Even if, say, machine1 goes down, clients can use machine2 to get access to other members of the cluster.

Hazelcast – Serialization

Hazelcast is ideally used in an environment where data/query are distributed across machines. This requires data to be serialized from our Java objects to a byte array which can be transferred over the network.

Hazelcast supports various types of Serialization. However, let's look at some commonly used ones, i.e., `Java Serialization` and `Java Externalizable`.

Java Serialization

First let's look at Java Serialization. Let's say, we define an `Employee` class with `Serializable` interface implemented.

```
public class Employee implements Serializable{
    private static final long serialVersionUID = 1L;
    private String name;
    private String department;

    public Employee(String name, String department) {
        super();
        this.name = name;
        this.department = department;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDepartment() {
        return department;
    }

    public void setDepartment(String department) {
        this.department = department;
    }
```

```
    @Override
    public String toString() {
        return "Employee [name=" + name + ", department=" + department + "]";
    }
}
```

Let's now write code to add Employee object to the Hazelcast map.

```
public class EmployeeExample {

    public static void main(String... args){

        //initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        //create a set to track employees
        Map<Employee, String> employeeOwners =
hazelcast.getMap("employeeVehicleMap");

        Employee emp1 = new Employee("John Smith", "Computer Science");

        // add employee to set
        System.out.println("Serializing key-value and add to map");
        employeeOwners.put(emp1, "Honda");

        // check if emp1 is present in the set
        System.out.println("Serializing key for searching and Deserializing
value got out of map");
        System.out.println(employeeOwners.get(emp1));

        // perform a graceful shutdown
        hazelcast.shutdown();
    }
}
```

It will produce the following output:

```
Serializing key-value and add to map
Serializing key for searching and Deserializing value got out of map
Honda
```

A very important aspect here is that simply by implementing a Serializable interface, we can make Hazelcast use Java Serialization. Also note that Hazelcast stores serialized data for key and value

instead of storing it in-memory like HashMap. So, Hazelcast does the heavy-lifting of Serialization and Deserialization.

However, there is a pitfall here. In the above case, what if the department of the employee changes? The person is still the same.

```
public class EmployeeExampleFailing {

    public static void main(String... args){

        //initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        //create a set to track employees
        Map<Employee, String> employeeOwners =
hazelcast.getMap("employeeVehicleMap");

        Employee emp1 = new Employee("John Smith", "Computer Science");

        // add employee to map
        System.out.println("Serializing key-value and add to map");
        employeeOwners.put(emp1, "Honda");


        Employee empDeptChange = new Employee("John Smith", "Electronics");

        // check if emp1 is present in the set
        System.out.println("Checking if employee with John Smith is present");
        System.out.println(employeeOwners.containsKey(empDeptChange));

        Employee empSameDept = new Employee("John Smith", "Computer Science");

        System.out.println("Checking if employee with John Smith is present");
        System.out.println(employeeOwners.containsKey(empSameDept));

        // perform a graceful shutdown
        hazelcast.shutdown();
    }
}
```

Output of the above code is:

```
Serializing key-value and add to map
Checking if employee with name John Smith is present
false
Checking if employee with name John Smith is present
true
```

It is because Hazelcast does not deserialize the key, i.e., `Employee` while comparison. It directly compares the bytecode of the serialized key. So, an object with the same value to all the attributes would be treated the same. But if the value to those attributes changes, for example, `department` in the above scenario, those two keys are treated as unique.

Java Externalizable

What if, in the above example, we don't care about the value of the department while performing serialization/deserialization of keys. Hazelcast also supports Java Externalizable which gives us control over what tags are used for serialization and deserialization.

Let's modify our `Employee` class accordingly:

```
public class EmplyoeeExternalizable implements Externalizable {
    private static final long serialVersionUID = 1L;
    private String name;
    private String department;

    public EmplyoeeExternalizable(String name, String department) {
        super();
        this.name = name;
        this.department = department;
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
        System.out.println("Deserializaing....");
        this.name = in.readUTF();
    }

    @Override
```

```
    public void writeExternal(ObjectOutput out) throws IOException {
        System.out.println("Serializing....");
        out.writeUTF(name);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDepartment() {
        return department;
    }

    public void setDepartment(String department) {
        this.department = department;
    }

    @Override
    public String toString() {
        return "Employee [name=" + name + ", department=" + department + "]";
    }
}
```

So, as you can see from the code, we have added `readExternal`/`writeExternal` methods which are responsible for serialization/deserialization. Given that we are not interested in the department while serialization/deserialization, we exclude those in `readExternal`/`writeExternal` methods.

Now, if we execute the following code:

```
public class EmployeeExamplePassing {

    public static void main(String... args){

        //initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        //create a set to track employees
```

```
        Map<EmplyoeeExternalizable, String> employeeOwners =
hazelcast.getMap("employeeVehicleMap");

        EmplyoeeExternalizable emp1 = new EmplyoeeExternalizable("John Smith",
"Computer Science");

        // add employee to map
        employeeOwners.put(emp1, "Honda");


        EmplyoeeExternalizable empDeptChange = new EmplyoeeExternalizable("John
Smith", "Electronics");

        // check if emp1 is present in the set
        System.out.println("Checking if employee with John Smith is present");
        System.out.println(employeeOwners.containsKey(empDeptChange));

        EmplyoeeExternalizable empSameDept = new EmplyoeeExternalizable("John
Smith", "Computer Science");

        System.out.println("Checking if employee with John Smith is present");
        System.out.println(employeeOwners.containsKey(empSameDept));

        // perform a graceful shutdown
        hazelcast.shutdown();
    }
}
```

The **output** we get is:

```
Serializing....
Checking if employee with John Smith is present
Serializing....
true
Checking if employee with John Smith is present
Serializing....
true
```

As the output shows, using `Externalizable` interface, we can provide Hazelcast with serialized data for only the name of the employee.

Also note that Hazelcast serializes our key twice:

- Once while storing the key,
- And, second for searching the given key in the map. As stated earlier, this is because Hazelcast uses serialized byte arrays for key comparison.

Overall, using Externalizable has more benefits as compared to Serializable if we want to have more control over what attributes are to be serialized and how we want to handle them.

Hazelcast – Advanced

Hazelcast – Spring Integration

Hazelcast supports an easy way to integrate with Spring Boot application. Let's try to understand that via an example.

We will create a simple API application which provides an API to get employee information for a company. For this purpose, we will use Spring Boot driven RESTController along with Hazelcast for caching data.

Note that to integrate Hazelcast in Spring Boot, we will need two things:

- Add Hazelcast as a dependency to our project.
- Define a configuration (static or programmatic) and make it available to Hazelcast

Let's first define the POM. Note that we have to specify Hazelcast JAR to use it in the Spring Boot project.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.example</groupId>
	<artifactId>hazelcast</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>demo</name>
	<description>Demo project to explain Hazelcast integration with Spring
Boot</description>

	<properties>
		<maven.compiler.target>1.8</maven.compiler.target>
		<maven.compiler.source>1.8</maven.compiler.source>
	</properties>


	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.4.0</version>
```

```
	</parent>

	<dependencies>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-cache</artifactId>
		</dependency>

		<dependency>
			<groupId>com.hazelcast</groupId>
			<artifactId>hazelcast-all</artifactId>
			<version>4.0.2</version>
		</dependency>

	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

Also add `hazelcast.xml` to `src/main/resources`:

```
<hazelcast
	xsi:schemaLocation="http://www.hazelcast.com/schema/config
http://www.hazelcast.com/schema/config/hazelcast-config-3.12.12.xsd"
	xmlns="http://www.hazelcast.com/schema/config"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

	<instance-name>XML_Hazelcast_Instance</instance-name>
</hazelcast>
```

Define an entry point file for Spring Boot to use. Ensure that we have `@EnableCaching` specified:

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@EnableCaching
@SpringBootApplication
public class CompanyApplication {
      public static void main(String[] args) {
            SpringApplication.run(CompanyApplication.class, args);
      }
}
```

Let us define our employee POJO:

```
package com.example.demo;

import java.io.Serializable;

public class Employee implements Serializable{
    private static final long serialVersionUID = 1L;
    private int empId;
    private String name;
    private String department;

    public Employee(Integer id, String name, String department) {
        super();
        this.empId = id;
        this.name = name;
        this.department = department;
    }

    public int getEmpId() {
        return empId;
    }

    public void setEmpId(int empId) {
        this.empId = empId;
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDepartment() {
        return department;
    }

    public void setDepartment(String department) {
        this.department = department;
    }

    @Override
    public String toString() {
        return "Employee [empId=" + empId + ", name=" + name + ", department=" +
department + "]";
    }
}
```

And ultimately, let us define a basic REST controller to access employee:

```
package com.example.demo;

import org.springframework.cache.annotation.Cacheable;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/v1/")
class CompanyApplicationController{

    @Cacheable(value = "employee")
    @GetMapping("employee/{id}")
    public Employee getSubscriber(@PathVariable("id") int id) throws
InterruptedException {
        System.out.println("Finding employee information with id " + id + "
...");
        Thread.sleep(5000);
```

```
        return new Employee(id, "John Smith", "CS");
    }
}
```

Now let us execute the above application, by running the command:

```
mvn clean install
mvn spring-boot:run
```

You will notice that the output of the command would contain Hazelcast member information which mean Hazelcast Instance is automatically configured for us using `hazelcast.xml` configuration.

```
Members {size:1, ver:1} [
        Member [localhost]:5701 - 91b3df1d-a226-428a-bb74-6eec0a6abb14 this
]
```

Now let us execute via **curl** or use browser to access API:

```
curl -X GET http://localhost:8080/v1/employee/5
```

The output of the API would be our sample employee.

```
{
  "empId": 5,
  "name": "John Smith",
  "department": "CS"
}
```

In the server logs (i.e. where Spring Boot application running), we see the following line:

```
Finding employee information with id 5 ...
```

However, note that it takes almost 5 secs (because of sleep we added) to access the information. But If we call the API again, the output of the API is immediate. This is because we have specified @Cacheable notation. The data of our first API call has been cached using Hazelcast as a backend.

Hazelcast – Monitoring

Hazelcast provides multiple ways to monitor the cluster. We will look into how to monitor via REST API and via JMX. Let's first look into REST API.

Monitoring Hazelcast via REST API

To monitor health of the cluster or member state via REST API, one has to enable REST API based communication to the members. This can be done by configuration and also programmatically.

Let us enable REST based monitoring via XML configuration in `hazelcast-monitoring.xml`:

```
<hazelcast
      xsi:schemaLocation="http://www.hazelcast.com/schema/config
http://www.hazelcast.com/schema/config/hazelcast-config-3.12.12.xsd"
      xmlns="http://www.hazelcast.com/schema/config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <instance-name>XML_Hazelcast_Instance</instance-name>

      <network>
            <rest-api enabled="true">
                  <endpoint-group name="CLUSTER_READ" enabled="true" />
                  <endpoint-group name="HEALTH_CHECK" enabled="true" />
            </rest-api>
      </network>
</hazelcast>
```

Let us create a Hazelcast instance which runs indefinitely in `Server.java` file:

```
public class Server {

    public static void main(String... args){

        //initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();
```

```
        // do not shutdown, let the server run
        //hazelcast.shutdown();
    }
}
```

And now let us execute start the cluster:

```
java '-Dhazelcast.config=hazelcast-monitoring.xml' -cp .\target\demo-0.0.1-
SNAPSHOT.jar com.example.demo.Server
```

Once started, the health of the cluster can be found out by calling the API like:

```
http://localhost:5701/hazelcast/health
```

The **output** of the above API call:

```
Hazelcast::NodeState=ACTIVE
Hazelcast::ClusterState=ACTIVE
Hazelcast::ClusterSafe=TRUE
Hazelcast::MigrationQueueSize=0
Hazelcast::ClusterSize=1
```

This displays that there is 1 member in our cluster and it is Active.

More detailed information about the nodes, for example, `IP`, `port`, `name` can be found using:

```
http://localhost:5701/hazelcast/rest/cluster
```

The **output** of the above API:

```
Members {size:1, ver:1} [
     Member [localhost]:5701 - e6afefcb-6b7c-48b3-9ccb-63b4f147d79d this
]

ConnectionCount: 1
AllConnectionCount: 2
```

JMX monitoring

Hazelcast also supports JMX monitoring of the data structures embedded inside it, for example, IMap, Iqueue, and so on.

To enable JMX monitoring, we first need to enable JVM based JMX agents. This can be done by passing "`-Dcom.sun.management.jmxremote`" to the JVM. For using different ports or use authentication, we can use `-Dcom.sun.management.jmxremote.port`, `-Dcom.sun.management.jmxremote.authenticate`, respectively.

Apart from this, we have to enable JMX for Hazelcast MBeans. Let us enable JMX based monitoring via XML configuration in `hazelcast-monitoring.xml`:

```
<hazelcast
      xsi:schemaLocation="http://www.hazelcast.com/schema/config
http://www.hazelcast.com/schema/config/hazelcast-config-3.12.12.xsd"
      xmlns="http://www.hazelcast.com/schema/config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <instance-name>XML_Hazelcast_Instance</instance-name>

      <properties>
            <property name="hazelcast.jmx">true</property>
      </properties>
</hazelcast>
```

Let us create a Hazelcast instance which runs indefinitely in `Server.java` file and add a map:

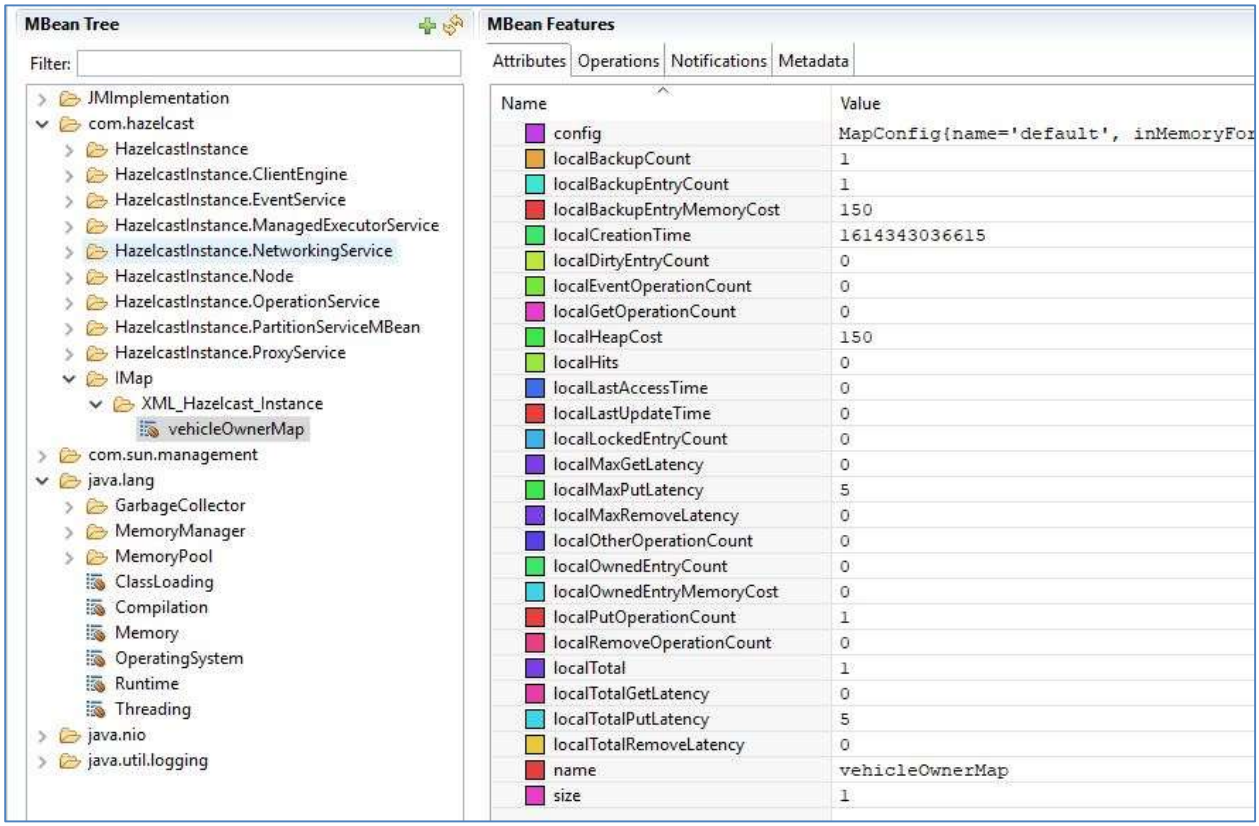
```
public class Server {

    public static void main(String... args){

        //initialize hazelcast server/instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        //create a simple map
        Map<String, String> vehicleOwners = hazelcast.getMap("vehicleOwnerMap");

        // add key-value to map
        vehicleOwners.put("John", "Honda-9235");

        // do not shutdown, let the server run
        //hazelcast.shutdown();
    }
}
```

Now we can execute the following command to enable JMX:

```
java  '-Dcom.sun.management.jmxremote' '-Dhazelcast.config=others\hazelcast-monitoring.xml' -cp .\target\demo-0.0.1-SNAPSHOT.jar com.example.demo.Server
```

The JMX ports can now be connected by JMX clients like `jConsole`, `VisualVM`, etc.

Here is a snapshot of what we will get if we connect using `jConsole` and see the attributes for `VehicleMap`. As we can see, the name of the map as `vehicleOwnerMap` and the size of map being 1.



Hazelcast – Map Reduce & Aggregations

MapReduce is a computation model which is useful for data processing when you have lots of data and you need multiple machines, i.e., a distributed environment to calculate data. It involves 'map'ing of data into key-value pairs and then 'reducing', i.e., grouping these keys and performing operation on the value.

Given the fact that Hazelcast is designed keeping a distributed environment in mind, implementing Map-Reduce Frameworks comes naturally to it.

Let's see how to do it with an example.

For example, let's suppose we have data about a car (brand & car number) and the owner of that car.

```
Honda-9235, John
Hyundai-235, Alice
Honda-935, Bob
Mercedes-235, Janice
Honda-925, Catnis
Hyundai-1925, Jane
```

And now, we have to figure out the number of cars for each brand, i.e., Hyundai, Honda, etc.

Let's try to find that out using MapReduce:

```
package com.example.demo;

import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Map;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.atomic.AtomicInteger;

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.ICompletableFuture;
import com.hazelcast.core.IMap;
```

```
import com.hazelcast.mapreduce.Context;
import com.hazelcast.mapreduce.Job;
import com.hazelcast.mapreduce.JobTracker;
import com.hazelcast.mapreduce.KeyValueSource;
import com.hazelcast.mapreduce.Mapper;
import com.hazelcast.mapreduce.Reducer;
import com.hazelcast.mapreduce.ReducerFactory;

public class MapReduce {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        try {
            // create two Hazelcast instances
            HazelcastInstance hzMember = Hazelcast.newHazelcastInstance();
            Hazelcast.newHazelcastInstance();

            IMap<String, String> vehicleOwnerMap =
hzMember.getMap("vehicleOwnerMap");

            vehicleOwnerMap.put("Honda-9235", "John");
            vehicleOwnerMap.putc"Hyundai-235", "Alice");
            vehicleOwnerMap.put("Honda-935", "Bob");
            vehicleOwnerMap.put("Mercedes-235", "Janice");
            vehicleOwnerMap.put("Honda-925", "Catnis");
            vehicleOwnerMap.put("Hyundai-1925", "Jane");

            KeyValueSource<String, String> kvs =
KeyValueSource.fromMap(vehicleOwnerMap);

            JobTracker tracker = hzMember.getJobTracker("vehicleBrandJob");
            Job<String, String> job = tracker.newJob(kvs);

            ICompletableFuture<Map<String, Integer>> myMapReduceFuture =
job.mapper(new BrandMapper())
                    .reducer(new BrandReducerFactory()).submit();

            Map<String, Integer> result = myMapReduceFuture.get();
            System.out.println("Final output: " + result);
        } finally {
            Hazelcast.shutdownAll();
        }
    }
```

```
    private static class BrandMapper implements Mapper<String, String, String,
Integer> {
        @Override
        public void map(String key, String value, Context<String, Integer>
context) {
            context.emit(key.split("-", 0)[0], 1);
        }
    }

    private static class BrandReducerFactory implements ReducerFactory<String,
Integer, Integer> {
        @Override
        public Reducer<Integer, Integer> newReducer(String key) {
            return new BrandReducer();
        }
    }

    private static class BrandReducer extends Reducer<Integer, Integer> {
        private AtomicInteger count = new AtomicInteger(0);

        @Override
        public void reduce(Integer value) {
            count.addAndGet(value);
        }

        @Override
        public Integer finalizeReduce() {
            return count.get();
        }
    }
}
```

Let's try to understand this code:

1. We create Hazelcast members. In the example, we have a single member, but there can well be multiple members.

2. We create a map using dummy data and create a Key-Value store out of it.

3. We create a Map-Reduce job and ask it to use the Key-Value store as the data.

4. We then submit the job to cluster and wait for completion.

5. The mapper creates a key, i.e., extracts brand information from the original key and sets the value to 1 and then emits that information as K-V to the reducer.

6. The reducer simply sums the value, grouping the data, based on key, i.e., brand name.

The **output** of the code:

```
Final output: {Mercedes=1, Hyundai=2, Honda=3}
```

Hazelcast – Collection Listener

Hazelcast supports addition of listeners when a given collection, for example, queue, set, list, etc. is updated. Typical events include entry added and entry removed.

Let's see how to implement a set listener via an example. So, let's say we want to implement a listener which tracks the number of elements in a set.

So, let's first implement the Producer:

```
public class SetTimedProducer{

    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        Thread.sleep(5000);

        // create a set
        ISet<String> hzFruits = hazelcast.getSet("fruits");

        hzFruits.add("Mango");

        Thread.sleep(2000);
        hzFruits.add("Apple");

        Thread.sleep(2000);
        hzFruits.add("Banana");

        System.exit(0);
    }
}
```

Now let's implement the listener:

```
package com.example.demo;

import java.io.IOException;

import com.hazelcast.core.ISet;
import com.hazelcast.core.ItemEvent;
import com.hazelcast.core.ItemListener;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

public class SetListener{

    public static void main(String... args) throws IOException,
InterruptedException {

        //initialize hazelcast instance
        HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

        // create a set
        ISet<String> hzFruits = hazelcast.getSet("fruits");

        ItemListener<String> listener = new FruitListener<String>();
        hzFruits.addItemListener(listener, true);

        System.exit(0);
    }

    private static class FruitListener<String> implements ItemListener<String> {
        private int count = 0;

        @Override
        public void itemAdded(ItemEvent<String> item) {
           System.out.println("item added" + item);
           count ++;
           System.out.println("Total elements" + count);
        }

        @Override
        public void itemRemoved(ItemEvent<String> item) {
            count --;
        }
    }
}
```

We will first run the producer:

```
java -cp .\target\demo-0.0.1-SNAPSHOT.jar com.example.demo.SetTimedProducer
```

And then, we run the listeners and let it run indefinitely:

```
java -cp .\target\demo-0.0.1-SNAPSHOT.jar com.example.demo.SetListener
```

The **output** from the Listener is as follows:

```
item added: ItemEvent{event=ADDED, item=Mango, member=Member [localhost]:5701 -
c28a60b7-3259-44bf-8793-54063d244394 this}
Total elements: 1
item added: ItemEvent{event=ADDED, item=Apple, member=Member [localhost]:5701 -
c28a60b7-3259-44bf-8793-54063d244394 this}
Total elements: 2
item added: ItemEvent{event=ADDED, item=Banana, member=Member [localhost]:5701 -
c28a60b7-3259-44bf-8793-54063d244394 this}
Total elements: 3
```

The call with `hzFruits.addItemListener(listener, true)` tells Hazelcast to provide member information. If set to false, we will just be notified that an entry was added/removed. This helps in avoiding the need to serialize and deserialize the entry to make it accessible to the listener.

Hazelcast Queue on single machine

Hazelcast queues are stored on a single member (along with a backup on different machines). This effectively means the queue can hold as many items which can be accommodated on a single machine. So, the queue capacity does not scale by adding more members. Loading more data than what a machine can handle in a queue can cause the machine to crash.

Using Map's set method instead of put

If we use IMap's `put(key, newValue)`, Hazelcast returns the `oldValue`. This means, extra computation and time is spent in deserialization. This also includes more data sent from the network. Instead, if we are not interested in the `oldValue`, we can use `set(key, value)` which returns `void`.

Let's see how to store and inject references to Hazelcast structures. The following code creates a map of the name "stock" and adds `Mango` at one place and `Apple` at another.

```
//initialize hazelcast instance
HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance();

// create a map
IMap<String, String> hzStockTemp = hazelcast.getMap("stock");
hzStock.put("Mango", "4");

IMap<String, String> hzStockTemp2 = hazelcast.getMap("stock");
hzStock.put("Apple", "3");
```

However, the problem here is that we are using `getMap("stock")` twice. Although this call seems harmless in a single node environment, it creates slowness in a clustered environment. The function call `getMap()` involves network round trips to other members of the cluster.

So, it is recommended that we store the reference to the map locally and use the referencing while operating on the map. For example:

```
// create a map
IMap<String, String> hzStock = hazelcast.getMap("stock");
hzStock.put("Mango", "4");
hzStock.put("Apple", "3");
```

Hazelcast uses serialized data for object comparison

As we have seen in the earlier examples, it is very critical to note that Hazelcast does not use deserialize objects while comparing keys. So, it does not have access to the code written in our equals/hashCode method. According to Hazelcast, keys are equal if the value to all the attributes of two Java objects is the same.

Use monitoring

In a large-scale distributed system, monitoring plays a very important role. Using REST API and JMX for monitoring is very important for taking proactive measures instead of being reactive.

Homogeneous cluster

Hazelcast assumes all the machines are equal, i.e., all the machines have same resources. But if our cluster contains a less powerful machine, for example, less memory, lesser CPU power, etc., then it can create slowness if the computation happens on that machine. Worst, the weaker machine can run out of resources causing cascading failures. So, it is necessary that Hazelcast members have equal resource power.