Logback

It is the successor of the **Log4j** logging library and is based on similar concepts.

Logback is fast in both synchronous and asynchronous logging and offers many useful features that make it a good choice for a project of any scale.

The most important difference between using a Logback logger and simply printing a message to System.out is that each logger has context. The logger context allows enabling or disabling certain log messages and is responsible for creating logger instances and arranging their hierarchy.

Adding Logback to a project

Installing Logback is very easy – simply add the dependencies to Maven or Gradle.
To get started with Logback, you will need to add the logback-classic dependency.
If you are using Maven, open the *pom.xml* file and add these lines:

```
<dependencies>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
    </dependency>
</dependencies>
```

If you use Gradle as your build tool, then add this line to your *build.gradle* file:

```
dependencies {
    implementation 'ch.qos.logback:logback-classic:1.2.11'
}
```

This library will transitively pull two other dependencies, slf4j-api and logback-core.

SLF4J (Simple Logging Facade for Java) is a facade or abstraction for various logging libraries, including Logback. It provides a simple logging API, and Logback implements it natively. You can invoke SLF4J logger with Logback as its underlying implementation without any overhead.
The logback-core library lays the foundation for Logback and provides a few ready-made classes for use. They are:

- ConsoleAppender, which adds log events to System.out or System.err;
- FileAppender, which adds log events to a file;
- RollingFileAppender, which adds log events to a file and can change its logging target to another file when a certain condition is met.

The logback-classic library provides classes that make it possible to send data to external systems. They are:

- SMTPAppender, which collects data in packets and sends the contents of the packet to a user-defined email after the occurrence of an event specified by the user;
- DBAppender, which adds data to database tables.

Basic logging
Let's create a class and declare a few Logger objects.

```
package com.example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Example {
   private static final Logger LOG_1 = LoggerFactory.getLogger(Example.class);
   private static final Logger LOG_2 =
LoggerFactory.getLogger("com.example.Example");

   public static void main(String[] args) {
      LOG_1.info("Information from LOG_1");
      LOG_2.warn("Warning from LOG_2");

      LOG_1.info("Are the loggers the same? {}", LOG_1 == LOG_2);
   }
}
```

As you can see, we don't refer to any Logback class directly.

Instead, we invoke SLF4J classes and interfaces, and SLF4J, in turn, delegates logging operations to Logback.

When creating a logger, we use the getLogger method of
the LoggerFactory that accepts either Class or String as an argument.

In both cases, the argument is used as the name of the logger.

If a logger with the same name already exists, the method returns the same logger, and if there is no logger with the same name, a new one is created.

A Logger object has a number of methods, namely trace, debug, info, warn, and error, for outputting a message of the corresponding log request level.

If we run the code, we'll get this output:

```
22:23:52.247 [main] INFO com.example.Example - Information from LOG_1
22:23:52.248 [main] WARN com.example.Example - Warning from LOG_2
22:23:52.248 [main] INFO com.example.Example - Are the loggers the same?
true
```

By default, each log line has the following elements: timestamp, thread name, log request level, logger name, and log message.

Message parameterization

Logback has the ability to add objects or variables to log messages. You can use it to know what kind of error has occurred and why.
Here is an example:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Example {
    private static final Logger LOG = LoggerFactory.getLogger(Example.class);

    public static void main(String[] args) {
        LOG.info("My name is {}. {} {}.", "Bond", "James", "Bond");
    }
}
```

By adding the {} brackets to logger messages, we can indicate where we want to put our variables.

```
22:43:49.141 [main] INFO com.example.Example - My name is Bond. James Bond.
```

We can also use this Logback function to output an error along with relevant information:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Example {
    private static final Logger LOG = LoggerFactory.getLogger(Example.class);

    public static void main(String[] args) {
        int number = 1;
        int divisor = 0;

        try {
            int result = number / divisor;
        } catch (ArithmeticException e) {
            LOG.error("Something went wrong with divisor {}", divisor, e);
        }
    }
}
```

Logback can extract the stack trace, and, as a result, below we can see that, due to the fact that we added an exception object to the message, it was displayed in the log.

You don't need to add a placeholder to the log message to display an exception object, but, in such cases, it must be the last argument passed to the log request method.

```
22:45:44.113 [main] ERROR com.example.Example - Something went wrong with divisor 0
java.lang.ArithmeticException: / by zero
        at com.example.Example.main(Example.java:11)
```

Fine-tuning the logger configuration

To configure loggers in Logback, you can use an *xml* file or a Groovy file. We will configure our logger with an *xml* file. You will need to create a *logback.xml* file and place it in the *resource* folder. This is what the basic settings file will look like:

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

    <root level="info">
        <appender-ref ref="console" />
    </root>
</configuration>
```

All logger settings are recorded between the <configuration> tags.
Next comes the tag <appender>. An **appender** is a tool that allows for configuring where and how the logs will be recorded. The name parameter specifies the name of the appender, and the class parameter specifies the class that will implement the appender.
The <encoder> tag specifies the format in which the logs will be recorded. A log message in the format we defined above would look like this:

```
01:15:54 INFO  com.example.Example - Customized message format
```

The <root> tag refers to the pre-defined root logger. Here we have specified the log level level="info", and the appender <appender-ref ref="console" /> was tied to it.
From the top of the settings, you can notice that we used a ConsoleAppender . This appender makes it possible to output logs to the console.
Another standard appender is FileAppender . As you can see by the name, this appender allows you to write logs to a file.

```
<configuration>
    <appender name="file" class="ch.qos.logback.core.FileAppender">
        <file>${user.dir}/logs/example.log</file>
        <encoder>
            <pattern>%d{HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</
```

```
pattern>
        </encoder>
    </appender>

    <root level="info">
        <appender-ref ref="file" />
    </root>
</configuration>
```

This is the most primitive setup for a FileAppender. A new addition to the settings is the <file> tag, in which we specify where the file is located and where we want to record logs. Don't worry if this file is not there — it will be created when the application is launched. Thanks to ${user.dir}, the log file will appear in the main project folder.

Logger levels

Logback allows you to configure the log level for individual packages and classes. To do this, you will need to go to the *logback.xml* file and specify the name and level parameters. In the name parameter, specify the path to the package or class in your project, and in the level parameter, specify the logging level that you need. Here is an example of how you can configure a logger level:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Example {
    private static final Logger LOG = LoggerFactory.getLogger(Example.class);

    public static void main(String[] args) {
        LoggerLevelClass.log();
        LOG.warn("WARN level message");
        LOG.info("INFO level message");
    }
}

class LoggerLevelClass {
    private static final Logger LOG =
LoggerFactory.getLogger(LoggerLevelClass.class);

    public static void log() {
        LOG.debug("DEBUG level message");
        LOG.info("INFO level message");
    }
}
```

For the classes above, we set up these logger levels:

```
<configuration>
```

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

<logger name="com.example" level="warn"/>
<logger name="com.example.LoggerLevelClass" level="info"/>

<root level="info">
   <appender-ref ref="console" />
</root>
</configuration>
```

Each logger will output only the messages whose log request level is higher than or equal to the level of the logger. There is a hierarchy in Logback that is based on names. That is, if we have a logger named com.logback.first, it will be the parent of a logger called com.logback.first.second, which is the parent of the com.logback logger.first.second.third. This means that you can define logger levels for entire packages. On top of every hierarchy is the root logger. If we run the code, we will see only two log messages:

```
23:34:02.575 INFO  com.example.LoggerLevelClass - INFO level message
23:34:02.576 WARN  com.example.Example - WARN level message
```