Singleton Pattern

A Singleton is a design pattern which allows the creation of an object in memory only once in an application and can be shared across multiple classes. It can be useful for services in an application, or other resources like a connection or thread pool.

To make a class into a Singleton, use:

1. `private static` variable of the class' type
2. `private` constructor - to prevent arbitrary object creation
3. `public static getInstance()` method, which will either instantiate the object or return the instance in memory

```
public class Singleton {
    // Private static variable of the class' type
    private static Singleton instance;

    // Private Constructor
    private Singleton() {}

    // Public static getInstance method
    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

Factory Pattern

Factory is a design pattern which creates objects in which the precise type may not be known until runtime. To make a factory, use:

1. Abstract data type
2. Classes that inherit the abstract data type (the objects whose instantiation details may not be known until runtime)
3. Static method that returns a concrete instance, whose return type is the abstract data type in (1)

```
    // Abstract Data Type
    public interface Dessert {}

    // Classes that inherit the abstract data type
    public class Cake implements Dessert {}

    public class Cookie implements Dessert {}

    public class IceCream implements Dessert {}

    // Good practice to throw an exception if the desired concrete instance is not fo
und
    public class DessertNotFoundException extends RuntimeException {}

    // Factory class that returns the concrete instance
    public class DessertFactory {
        public static Dessert getDessert(String dessertType) {
            switch(dessertType) {
                case "cake":
                    return new Cake();
                case "cookie":
                    return new Cookie();
                case "ice cream":
                    return new IceCream();
                default:
                    throw new DessertNotFoundException(dessertType + " not found!");
            }
        }

        public class Demo {
            public static void main(String[] args) {
                Dessert d1 = DessertFactory.getDessert("cake");
                Dessert d2 = DessertFactory.getDessert("cookie");
                Dessert d3 = DessertFactory.getDessert("ice cream");
```

```
                Dessert d4 = DessertFactory.getDessert("candy");    // Throws Dessert
NotFoundException
            }
        }
    }
```

POM - Project Object Model

Maven is a dependency manager and build automation tool for Java programs. Maven project configuration and dependencies are handled via the Project Object Model, defined in the `pom.xml` file. This file contains information about the project used to build the project, including project dependencies and plugins.

Some other important tags within the `pom.xml` file include:

- `<project>` - this is the root tag of the file
- `<modelVersion>` - defining which version of the page object model to be used
- `<name>` - name of the project
- `<properties>` - project-specific settings
- `<dependencies>` - this is where you put your Java dependencies you want to use. Each one needs a `<dependency>`, which has:
    - `<groupId>` , `<artifactId>`, `<version>` - project coordinates
- `<plugins>` - for 3rd party plugins that work with Maven

Here's an example:

```
<project>
  <modelVersion>4.0.0</modelVersion>


  <groupId>com.revature.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>


  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-artifact</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
```

```
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-core</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
  </dependencies>

</project>
```

Maven Build Lifecycle

When Maven builds your project, it goes through several steps called **phases**. The default maven build lifecycle goes through the following phases:

1. Validate => project is correct and all necessary information is available
2. Compile => compiles project source code
3. Test => tests all compiled code
4. Package => packages all compiled code to WAR/JAR file
5. Integration => performs all integration tests on WAR/JAR
6. Verify => runs checks on the results of integration tests
7. Install => installs WAR/JAR to local repository
8. Deploy => copies final WAR/JAR to the remote repository

Each phase in turn is composed of plugin goals that are bound to zero or more build phases. A "goal" represents a specific task which contributes to the building or managing of the project.

For more information, see the Maven documentation.

Using the `mvn` command

To use the Maven CLI (command-line interface), first test that you have Maven installed:

```
mvn --version
```

Now, once you are in your project directory, you can run any phase in the default build lifecycle. Maven will look for the `pom.xml` file and use that to run the phase.

```
cd /path/to/myproject/
mvn package
```

To execute a specific Maven goal, use the `plugin:goal` syntax:

```
mvn dependency:copy-dependencies
```

Multiple phases or goals can be run sequentially. Again, see the Maven documentation for more information.

Maven Repositories

When Maven "builds" a Java project, it must first search for any dependencies declared in the `pom.xml` file. Maven dependencies are stored both locally and in a central repository. The local repository is in the `$HOME/.m2/repository` folder (can be changed in `$MAVEN_HOME/conf/settings.xml`), while the central repository is accessible at https://mvnrepository.com. If Maven cannot find a given dependency locally, it searches the central repository for the artifact and then downloads it to the local repository.

A Maven "build" means to take the project source code, along with any dependencies like libraries or frameworks, compile it, and bundle it all together into an artifact - this could be a `.war` file, a `.jar` file, or an `.ear` file. WAR stands for "web archive", JAR stands for "Java archive", and EAR stands for "Enterprise Application archive". This artifact can then be either directly run or deployed onto a web container (in the case of a web application).

Maven Project Coordinates

Maven Project coordinates identify uniquely a project, a dependency, or a plugins defined in the `pom.xml` file - these are:

- `group-id` - The group, company, team, organization, project, or other group. for example: "com.revature"
- `artifact-id` - A unique identifier under `groupId` that represents a single project. for example: "myproject"
- `version` - A specific release of a project. Projects that have been released have a fixed version identifier that refers to a specific version of the project. Projects undergoing active development can use a special identifier that marks a version as a `SNAPSHOT`. for example: "0.0.1-SNAPSHOT"
- `packaging`- The type of project, defaulting to `jar`, describing the packaged output produced by a project. A project with packaging `jar` produces a JAR archive; a project with packaging `war` produces a web application.

Overview of Logging

Log4j

Log4j is a reliable, fast, and flexible logging framework for Java supported by Apache. It's commonly used to record application events, log granular debugging information for developers, and write exception events to files.

Why do we need logging?

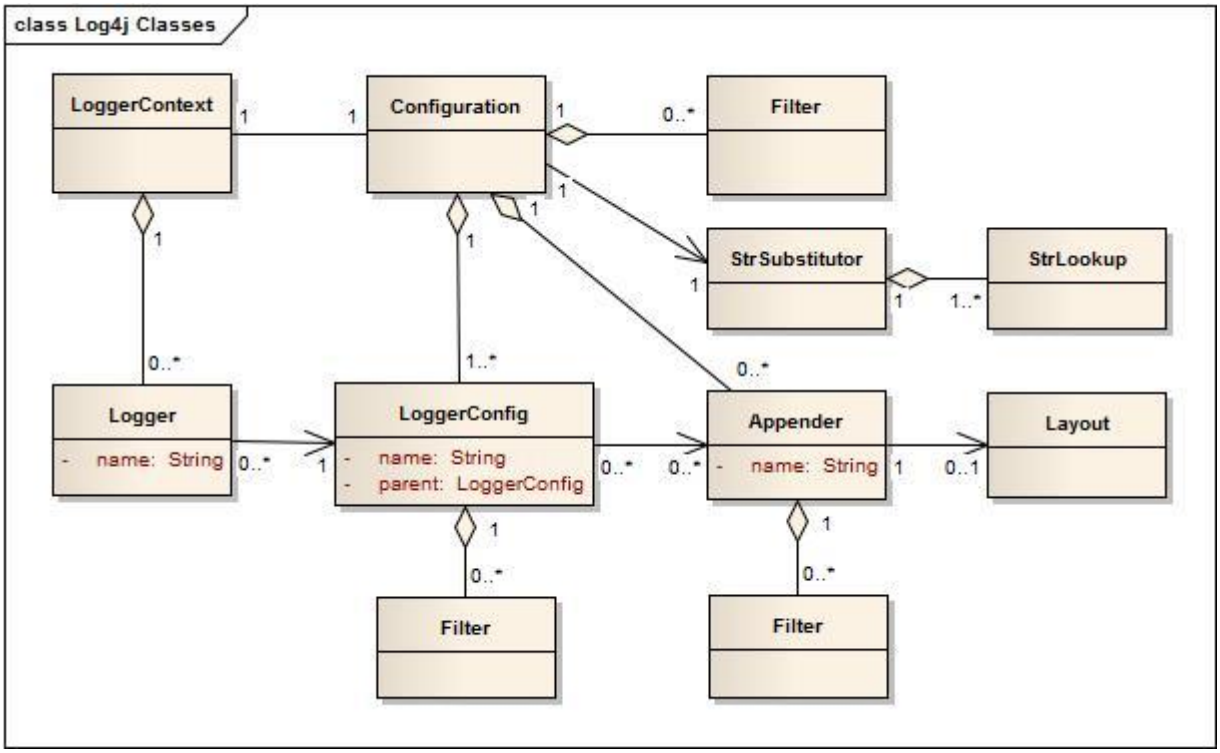
Logging records events that occur during software execution. As users execute programs on the client side, the system accumulates log entries for support teams. In general, it allows for developers to access information about applications to which we do not have direct access. Without logs, we would have no idea of knowing what went wrong when an application crashes, or track and monitor application performance.

Also, a logging framework like Log4j is critical because it allows us to use various logging levels and configure a threshold to determine which messages will be recorded in the application log files.

Logging Levels

Log4j Usage

Log4j has a simple class architecture as shown below:



The main components of the hierarchy are:

- `Logger` - logs the messages

- `Appender` - publishes logs to destination(s)
- `Layout` - formats logging information
- `Configuration` - stores settings
- `Filter` - used to filter out logs that do not meet the threshold

Log4j Logging Levels

Below are the Log4j log levels, in order of least to most restrictive:

1. **ALL** => all levels
2. **DEBUG** => designates fine-grained informational events that are most useful to debug an application
3. **INFO** => informational messages that highlight the progress of the application at the coarse grained level
4. **WARN** => designates potentially harmful situations
5. **ERROR** => designates error events that might still allow the application to continue running
6. **FATAL** => severe error events that presumably lead the application to abort
7. **OFF** => highest possible level, intended to turn off logging

How do logging levels work?

A log request of level *x* in a logger with level *y* is enabled with *x* >= *y*. For the standard levels, we have that ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF

Configuration

Log4j (specifically Log4j2) can be configured using XML, JSON, YML, or Properties files, as described in their documentation.

Simple Example

To use Log4j2, first include the library as a dependency. If you're using Maven, simply add the following to your `pom.xml` file:

```
<!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.11.2</version>
</dependency>
```

Assuming you've setup some configuration file, you can go ahead and use loggers in your code:

```
public class Foo {
```

```
  private static Logger log = LogManager.getLogger(Foo.class);

  public static void main(String[] args) {
    log.info("Hello world!");
        log.warn("Uh oh");
        log.error("This is not good!");
  }

}
```

Setting Logging Threshold

A logger can be assigned a threshold level. All logging requests with level lower than this threshold will be ignored.

For example, setting logger threshold to `INFO` means that logging requests with levels `TRACE` and `DEBUG` will not be logged by this logger.

An example of setting the root logger threshold to `INFO`:

```
<configuration xmlns="http://logging.apache.org/log4php/">
    <appender name="default" class="LoggerAppenderConsole" />
    <root>
        <level value="info" />
        <appender_ref ref="default" />
    </root>
</configuration>
```

If not explicitly configured, loggers will have their threshold level set to `DEBUG` by default.