eclipse

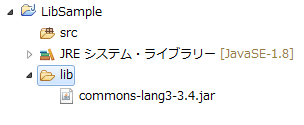
プロジェクトにライブラリを追加する方法

①jarの配置

②クラスパスを通す

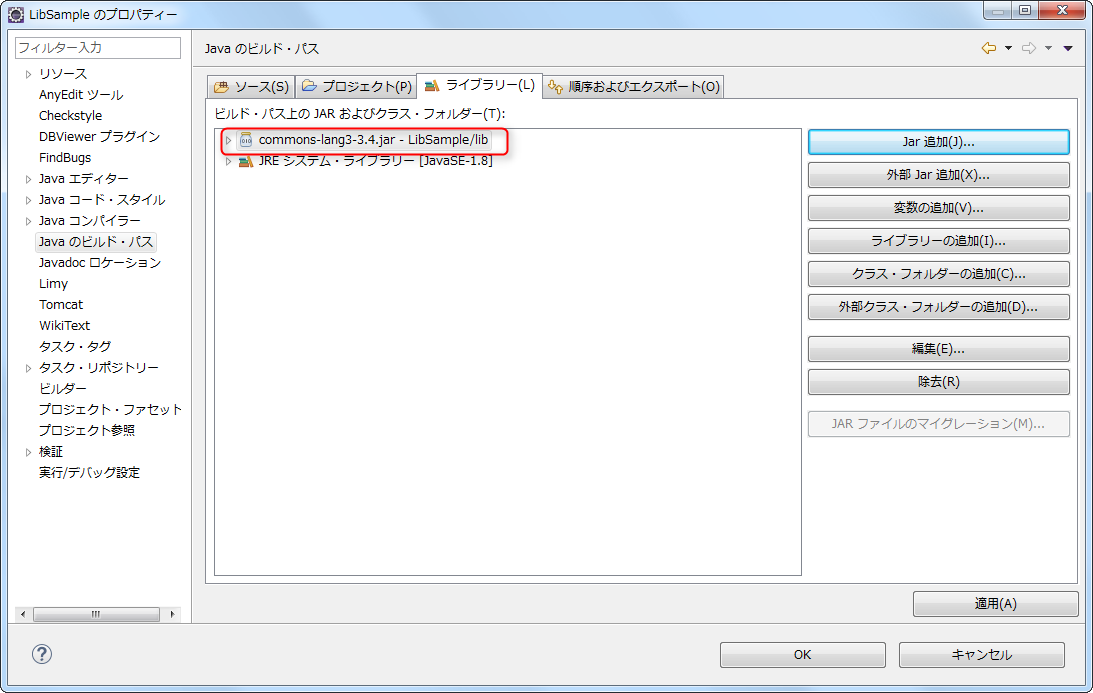
①jarの配置

プロジェクト直下にlibフォルダを作成しそこにjarを置くのが慣例



②クラスパスを通す

「Jarの追加」または「外部Jarの追加」



プロジェクトにクラスパスを通すと「参照ライブラリー」というのが表示される



<https://ittoybox.com/archives/385>

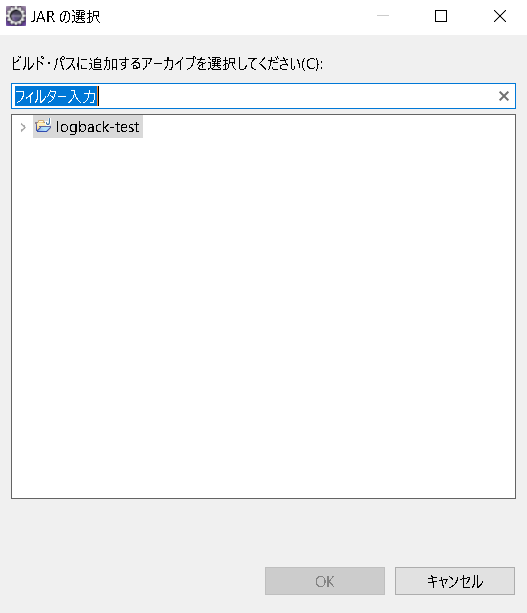
JARの追加、外部JARの追加

ビルドパス（クラスパス）にJARを追加すること



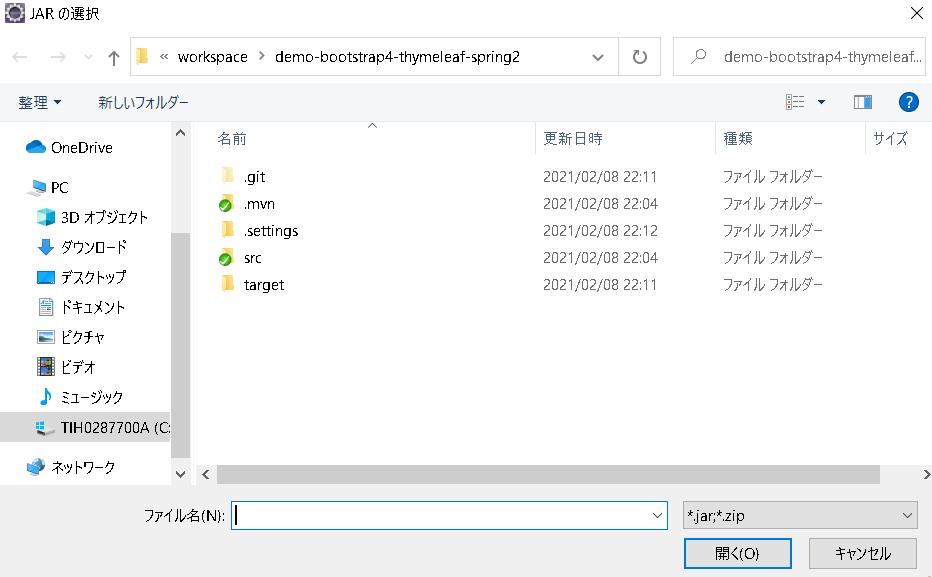
JARの追加

ワークスペース(プロジェクト)からJARを選択



外部JARの追加

ファイルシステムからJARを選択



クラスパス

クラスファイルの場所

フォルダ、zip、jarの3つが指定可能

デフォルトはカレントディレクトリ

Stringなどの組み込みクラスは勝手にロードされる

プロパティファイルなどもクラスパスに置く

JAVA\_HOME

環境変数

他のアプリケーションで必要とする場合がある

JDKまたはJREフォルダを指定

Maven

Super POM

<https://maven.apache.org/ref/3.0.4/maven-model-builder/super-pom.html>

・全てのPOMが継承してる

・jarのダウンロード先が記載されてる



Central Repository

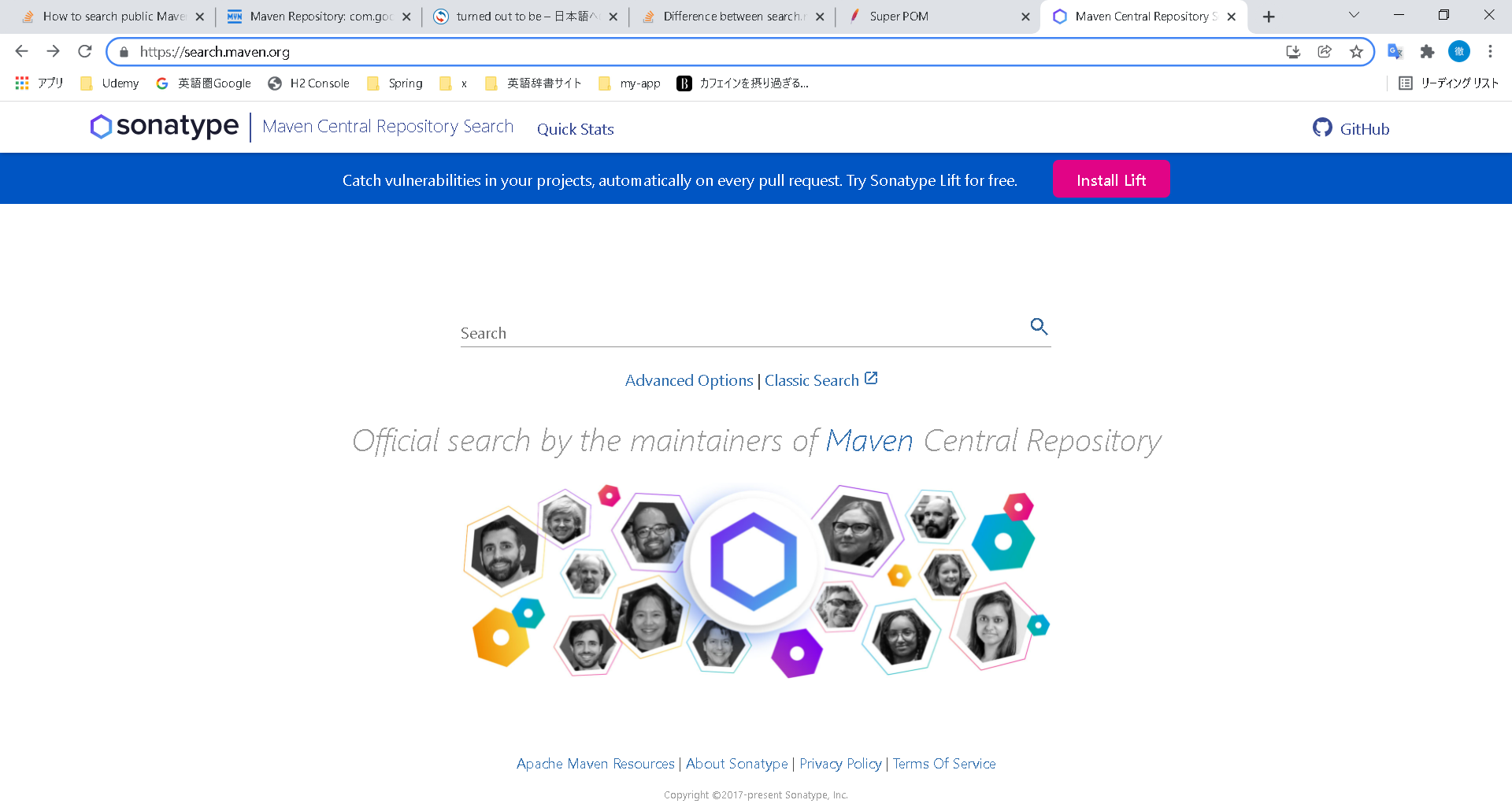
・<https://repo.maven.apache.org/maven2/>

・jarのダウンロード先

Central Repositoryの検索エンジン

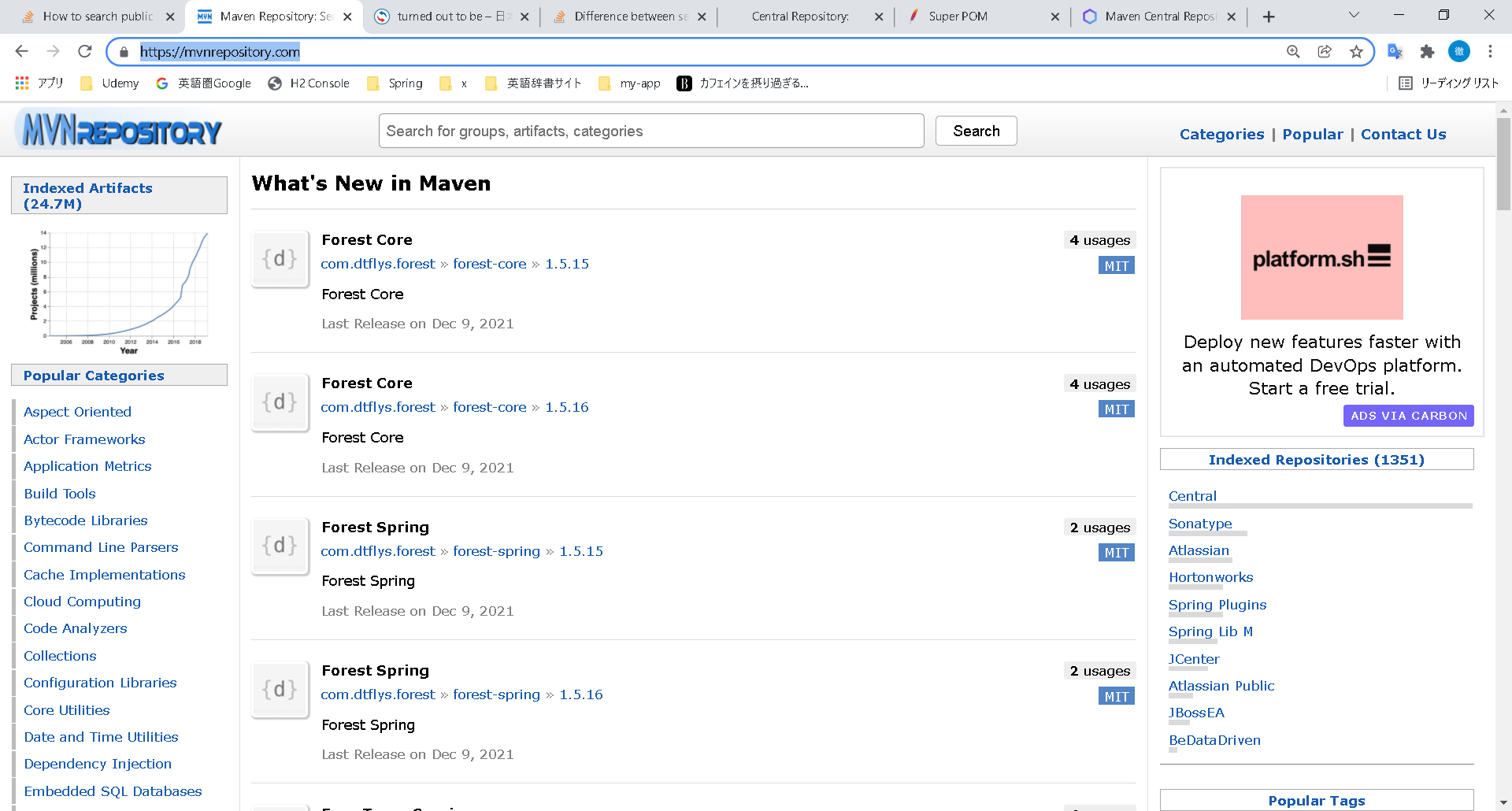
・https://search.maven.org/　公式

　classでの検索ができる



・https://mvnrepository.com/

　GroupId、ArtifactIdなどでの検索しかできない



JREとJDK

JRE

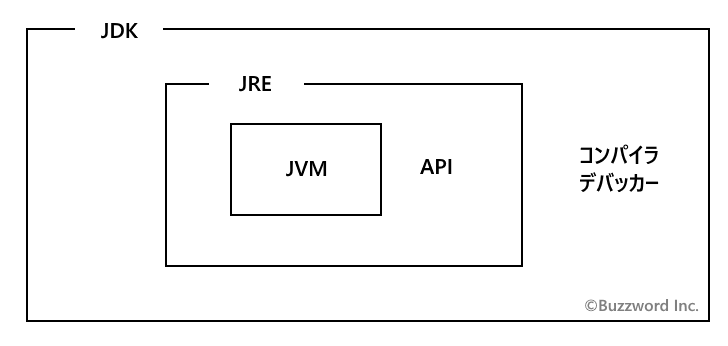
Java Runtime Environment

Java実行環境

JDK

Java Development Kit

JRE　＋　コンパイラ（javac.exe）やデバッガなどの開発ツール



javaコマンド

java {実行クラス}

java -cp {クラスパス} {実行クラス}

{クラスパス}

・-cpまたは-classpathで指定

・ディレクトリ、jar、zipの3つが指定可能

・複数指定する場合はwindowsでは「;」、linuxでは「：」で区切る

・パッケージ宣言のあるクラスを実行する場合はルートパッケージの1階層上のディレクトリを指定

・クラスパスを指定しないとデフォルトはカレントディレクトリ

・クラスパスを指定するとデフォルトのカレントディレクトリが除外されてしまうため必要がある場合は「.」で指定

・ワイルドカード「\*」が指定可能

{実行クラス}

・メインメソッドを持つクラスを指定

・FQCNで指定

・.classは記述しない

メインクラス

C:\work\logback-test\bin\chapters\introduction\HelloWorld1.class

FQCN：chapters.introduction.HelloWorld1

ライブラリ

C:\work\logback-test\lib\logback-core-1.2.3.jar

C:\work\logback-test\lib\slf4j-api-1.7.30.jar

実行例

C:\>java -cp C:\work\logback-test\bin;C:\work\logback-test\lib\\* chapters.introduction.HelloWorld1

C:\work\logback-test>java -cp bin;lib\\* chapters.introduction.HelloWorld1

C:\work\logback-test\bin>java -cp .;../lib\logback-classic-1.2.3.jar;../lib\logback-core-1.2.3.jar;../lib\slf4j-api-1.7.30.jar; chapters.introduction.HelloWorld1

Java

logback

・logback-classic

・ロガーの取得

・ログレベル

・{}

・logback.xml

・変換指定子

logback-classic

依存関係にはlogback-classic を追加する。

以下のjarをダウンロードしてくる。

・logback-classic.jar

・logback-core.jar

・slf4j-api.jar

ロガーの取得

LoggerFactory#getLogger()

引数にクラスを指定するとFQCN を名前にしたロガーを取得できる。

ログレベル

trace

debug

info

warn

error

{}

logger.debug("The new entry is {}.", entry);

logback.xml

logbackの設定ファイル

クラスパス直下に置く

置かなかったらデフォルトの設定が自動で適用される

<configuration>

<!--変数-->

<property name="LOG\_DIR" value="C:/logs" />

<!--アペンダー-->

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<!--フォーマット-->

<encoder>

<pattern>%d{yyyy-MMM-dd HH:mm:ss.SSS} %-5level [%thread] %logger{35} - %msg %n</pattern>

</encoder>

</appender>

<appender name="FILE" class="ch.qos.logback.core.FileAppender">

<file>${LOG\_DIR}/spring-boot-sample-logger-example.log</file>

<encoder>

<pattern>%d{yyyy-MMM-dd HH:mm:ss.SSS} %-5level [%thread] %logger{35} - %msg %n</pattern>

<charset>UTF-8</charset>

</encoder>

</appender>

<!--ロガー-->

<logger name="com.example.actor" level="DEBUG" />

<logger name="org.hibernate" level="ERROR" />

<logger name="org.springframework" level="INFO" />

<logger name="org.thymeleaf" level="INFO" />

<!--ルートロガー-->

<root>

<!--アペンダーの割り当てー-->

<appender-ref ref="STDOUT" />

<appender-ref ref="FILE" />

</root>

</configuration>

ロガー

・アペンダーを割り当てる

・ログレベルを設定する

アペンダー

・ログの出力先

ロガーの親子関係

・全てのロガーは、ルートロガーを最上位とした親子関係になっている

・親子はドット (.)で区切られる

・アペンダーとログレベルが直近の親から引き継がれる

・ログレベルはオーバーライドできる

・アペンダーの設定は積み重ねられる

変換指定子

%<変換指定子>{オプション}

|  |  |
| --- | --- |
| %d,%date | 日時 |
| %level | ログレベル |
| %thred | スレッド |
| %logger | ロガーの名前 |
| %msg | メッセージ |
| %n | 改行 |

変数定義

<property name="LOG\_DIR" value="C:/logs"/>

参考

・Logback 使い方メモ

<https://qiita.com/opengl-8080/items/49719f2d35171f017aa9>

・[第3章 logbackの設定](http://logback.qos.ch/manual/configuration_ja.html)

<http://logback.qos.ch/manual/configuration_ja.html>

プロパティファイル

・javaではプロパティファイルの文字コードは基本的にISO-8859-1で記述する。（springを使うときはこの文字コードでなければならない）

・ISO-8859-1は日本語などのマルチバイトは表現できないため、日本語を表現するときはユニコード・エスケープ形式（\xxxx)を使用する。

JVM

JVMの終了

/\* 何らかのデータを読み込む処理 \*/

if(/\* データが正しくない場合 \*/) {

System.out.println("データが壊れています。異常終了します。");

System.exit(1);

}

System.out.println("正常終了しました。");

・System.exit()メソッドでJVMを終了できる

・引数にはint型の終了コードを渡す。

　0なら正常終了。0以外なら異常終了。

JVMシステムプロパティ

・システムプロパティ = JVMのシステムプロパティ = JVMシステムプロパティ

・主なJVMシステムプロパティ

|  |  |
| --- | --- |
| java.version | JREのバージョン |
| java.home | JREのインストール先ディレクトリ |
| os.name | OSの名前 |
| line.separator | 改行コード |
| user.name | ユーザ名 |
| user.home | ユーザのホームディレクトリ |
| user.dir | ユーザの現在の作業ディレクトリ |

JVMシステムプロパティの取得

// 個別に表示

System.out.println("user.dir" + "=" + System.getProperty("user.dir"));

// 全て表示

Properties properties = System.getProperties();

for(Object key : properties.keySet()) {

System.out.println(key + "=" + System.getProperty((String)key));

}

・System.getProperty()メソッドで引数にString型のキー名を指定する

・System.getProperties()メソッドで全てのJVMシステムプロパティを取得できる。

JVMシステムプロパティの設定

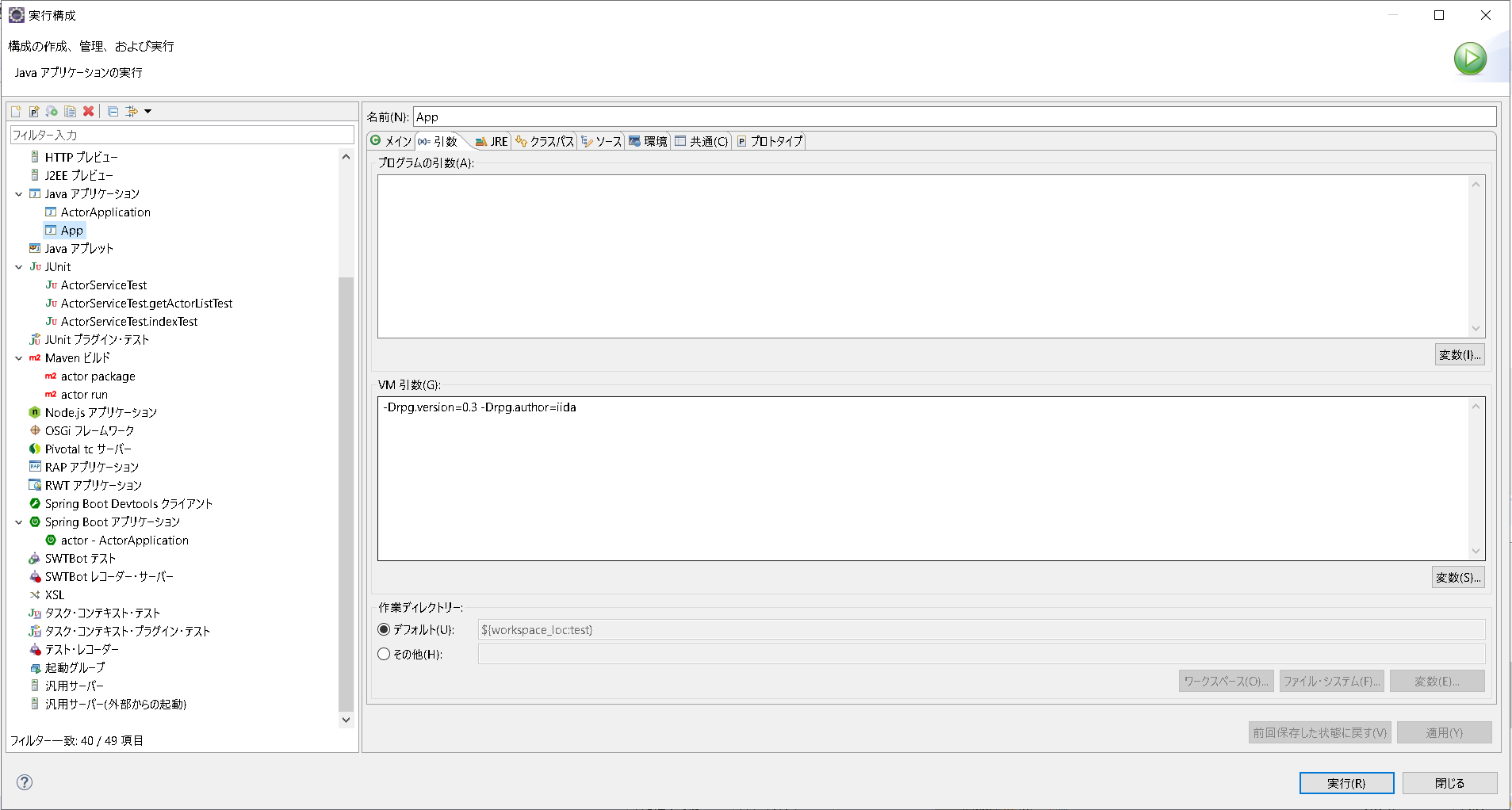
・プログラムから

System.setProperty("rpg.version", "0.3");

・javaコマンド実行時

> java -Drpg.version=0.3 -Drpg.author=湊 Main

・eclipseでjavaプログラムを実行時



[実行]→[実行の構成] →[引数]タブ→[VM 引数]に設定

OSの環境変数の取得

//全て表示

Map<String, String> enviroment = System.getenv();

for(String key : enviroment.keySet()) {

System.out.println(key + "=" + enviroment.get(key));

}

・System.getenv()メソッドを使う

Junit4

参考

・ドキュメント

<https://junit.org/junit4/javadoc/latest/>

・qiita

<https://qiita.com/tsukakei/items/b892409cf982f1951933>

@Test

・テストメソッドであることを示す

・public voidなメソッドに付ける

@Test

public void methodTest() {

org.junit.Assert.assertTrue( new ArrayList().isEmpty() );

}

・expectedオプションで期待する例外を指定できる

@Test(expected=IndexOutOfBoundsException.class)

public void outOfBounds() {

new ArrayList<Object>().get(1);

}

・timeoutオプションで制限時間を指定できる

@Test(timeout=100)

public void infinity() {

while(true);

}

@Ignore

・一時的に実施しないことを示す

・テストメソッドに付与する

@Ignore

@Test

public void something() { ..

・メッセージをつけることも可能

@Ignore("not ready yet")

@Test

public void something() { ...

・テストクラスに付与することも可能

@Ignore

public class IgnoreMe {

@Test

public void test1() { ... }

@Test

public void test2() { ... }

}

@Before

・各テストメソッドを実行する前に都度実行するメソッド（setUpメソッド）

・public voidなメソッドに付与

public class Example {

List empty;

@Before

public void initialize() {

empty= new ArrayList();

}

@Test

public void size() {

...

}

@Test

public void remove() {

...

}

}

@BeforeClass

・各テストメソッドを実行する前に一度だけ実行するメソッド（setUpメソッド）

・public voidなメソッドに付与

public class Example {

@BeforeClass

public static void onlyOnce() {

...

}

@Test

public void one() {

...

}

@Test

public void two() {

...

}

}

@After

・各テストメソッドを実行した後に都度実行するメソッド（tearDownメソッド）

・public voidなメソッドに付与

public class Example {

File output;

@Before

public void createOutputFile() {

output= new File(...);

}

@Test

public void something() {

...

}

@After

public void deleteOutputFile() {

output.delete();

}

}

@AfterClass

・各テストメソッドを実行した後に1度だけ実行するメソッド（tearDownメソッド）

・pubic void なメソッドに付与

public class Example {

DatabaseConnection database;

@BeforeClass

public static void login() {

database= ...;

}

@Test

public void something() {

...

}

@Test

public void somethingElse() {

...

}

@AfterClass

public static void logout() {

database.logout();

}

}

Mockito

参考

・公式サイト

[https://site.mockito.org](https://site.mockito.org/)

・ドキュメント

<https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html>

・qiita

<https://qiita.com/mima_ita/items/3574a03b3379fb5f3c3c>

概要

・クラスやインターフェイスをモックできる

・以下はモックできない

staticメソッド

privateメソッド

finalメソッド

コンストラクタ

・モックしたインスタンスのメソッドはデフォルトではfalsyな値を返す

依存関係

mockito-coreのjarを追加

モックの生成

・mockメソッド

・@Mockアノテーション

@Mock,@Spy,@InjectMocksなどの初期化

・initMocks

・@Runwith

・@Rule

Argument matchers

・any〜など

・Argument matchersを使うときは全ての引数がArgument matchersでなければならない

・リテラルを使う場合はeqを使う

メソッドのモック

・when & then〜

・do〜 & when

voidメソッド、spyインスタンスをモックするとき

do〜 & whenを使う

Spyの方法

・spyメソッド

・@Spyアノテーション

@InjectMocks

モック・Spyをインジェクトする

powermock

・ドキュメント

<https://github.com/powermock/powermock/wiki/Mockito>

できること

staticメソッドのモック

privateメソッドのモック

finalメソッドのモック

コンストラクタのモック

staticイニシャライザの除去

依存関係

ドキュメント参照

powermockを使う準備

テストクラスに以下のアノテーションを付与

@RunWith(PowerMockRunner.class)

@PrepareForTest( { Target.class })

@RunWithは@Ruleでも代替可能

https://github.com/powermock/powermock/wiki/PowerMockRule

プライベートなフィールドやメソッドにアクセス

Whitebox.setInternalState(..)

Whitebox.getInternalState(..)

Whitebox.invokeMethod(..)

Whitebox.invokeConstructor(..)

<https://github.com/powermock/powermock/wiki/Bypass-Encapsulation>

java.langやjava.netなどのシステムクラスのメソッドをモックする場合はラップする

<https://github.com/powermock/powermock/wiki/Mock-System>

staticメソッドをモック

@PrepareForTest(Static.class)

PowerMockito.mockStatic()

コンストラクタをモック

ドキュメント参照

プライベートメソッドをモック

Spyを使う