

Ant と build.xml

配列と演数子

制御文
メソッド

第 1.0 版

作 成 者	NH 中村広二
作 成 日	2019 年 8 月 8 日
最終更新日	2019 年 8 月 8 日

Ant と build.xml

Apache Ant（アパッチ アント）は、ビルドツールソフトウェア。

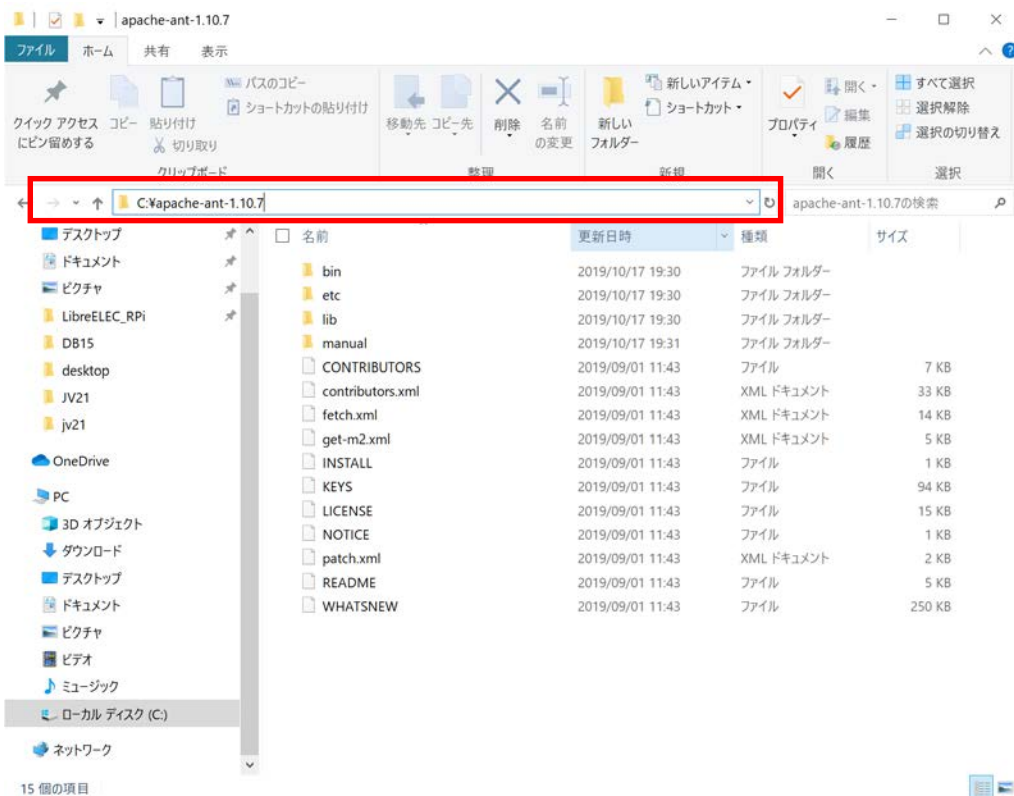
GNU make の Java 版ともいえるものであり、オペレーティングシステム（OS）など特定の環境に依存しにくいビルドツールである。XML 文書でビルド（ソフトウェア構築）のルールを記述することが特徴である。統合開発環境 Eclipse には Ant プラグインが標準で内蔵されている。元々 Apache Tomcat をビルドするために開発されたものである。

Ant はタスクと呼ばれる何種類もの XML 要素をビルドファイル（デフォルトでは build.xml）上に記述してビルドのルールを作る。

Ant を用いて、コンパイルからアプリケーション jar（実行用の圧縮ファイル形式）ができる開発環境を整える。war（Web アプリケーションの圧縮ファイル形式）も Ant でビルド（アプリ作成）できるが、今回は軽く説明する程度とする。総合開発環境 Eclipse でも同様に jar と war を作成できる。Eclipse では、Ant で作成するよりも容易に作成できる。

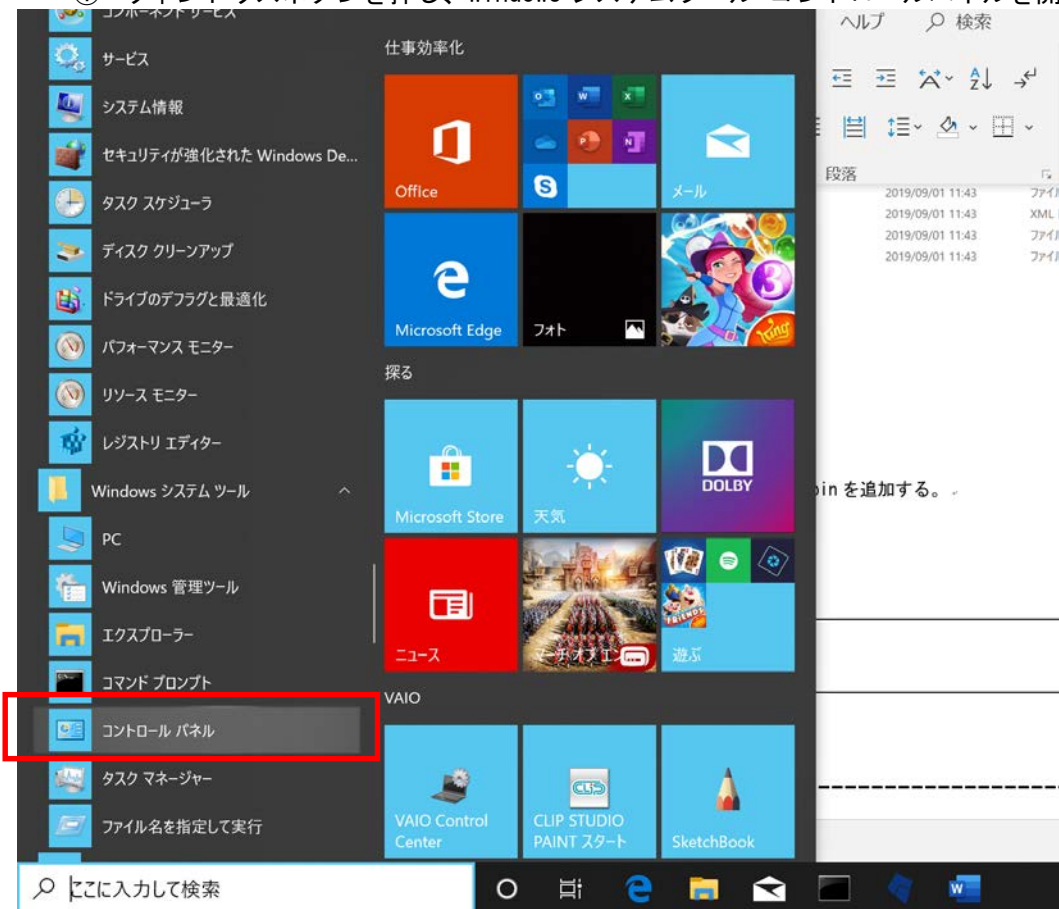
Ant のインストール

(1) apache-ant-1.10.7-bin.zip を C:\%に解凍する。



(2) 環境変数に ANT_HOME と環境変数 path に ANT_HOME/bin を追加する。

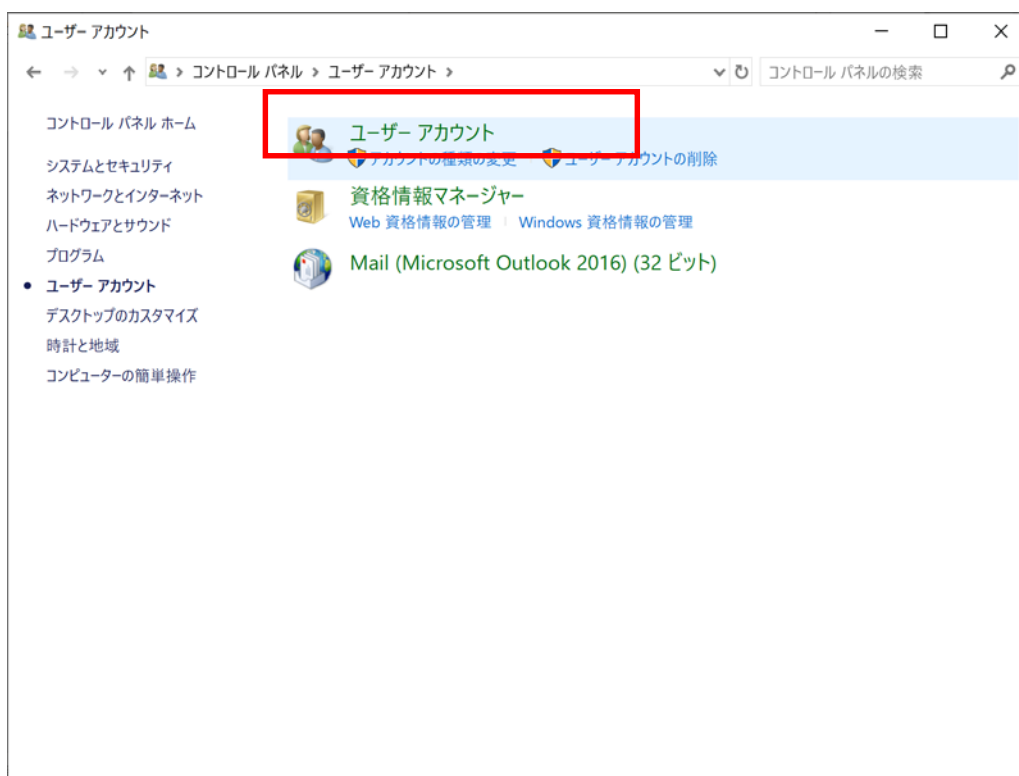
① ウィンドウズボタンを押し、Windows システムツール-コントロールパネルを開く。



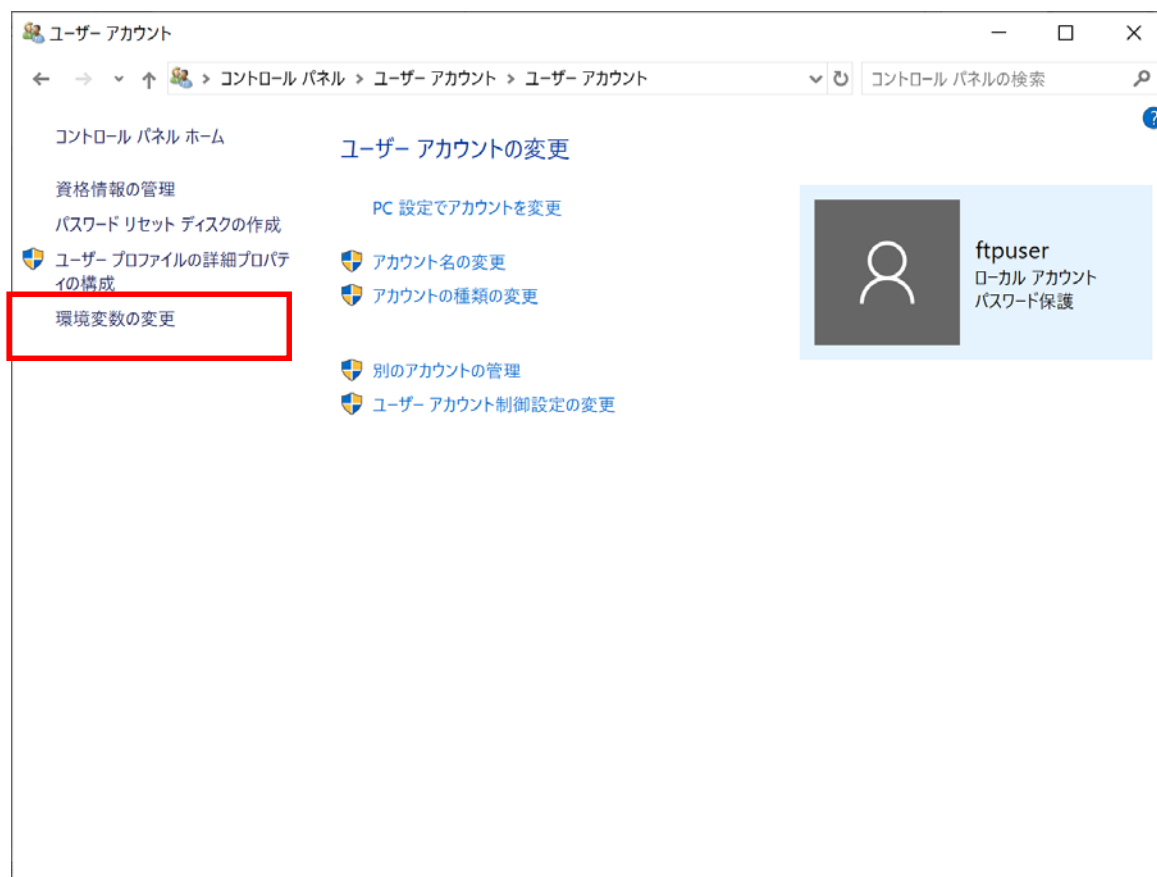
② ユーザーアカウントをクリックする。



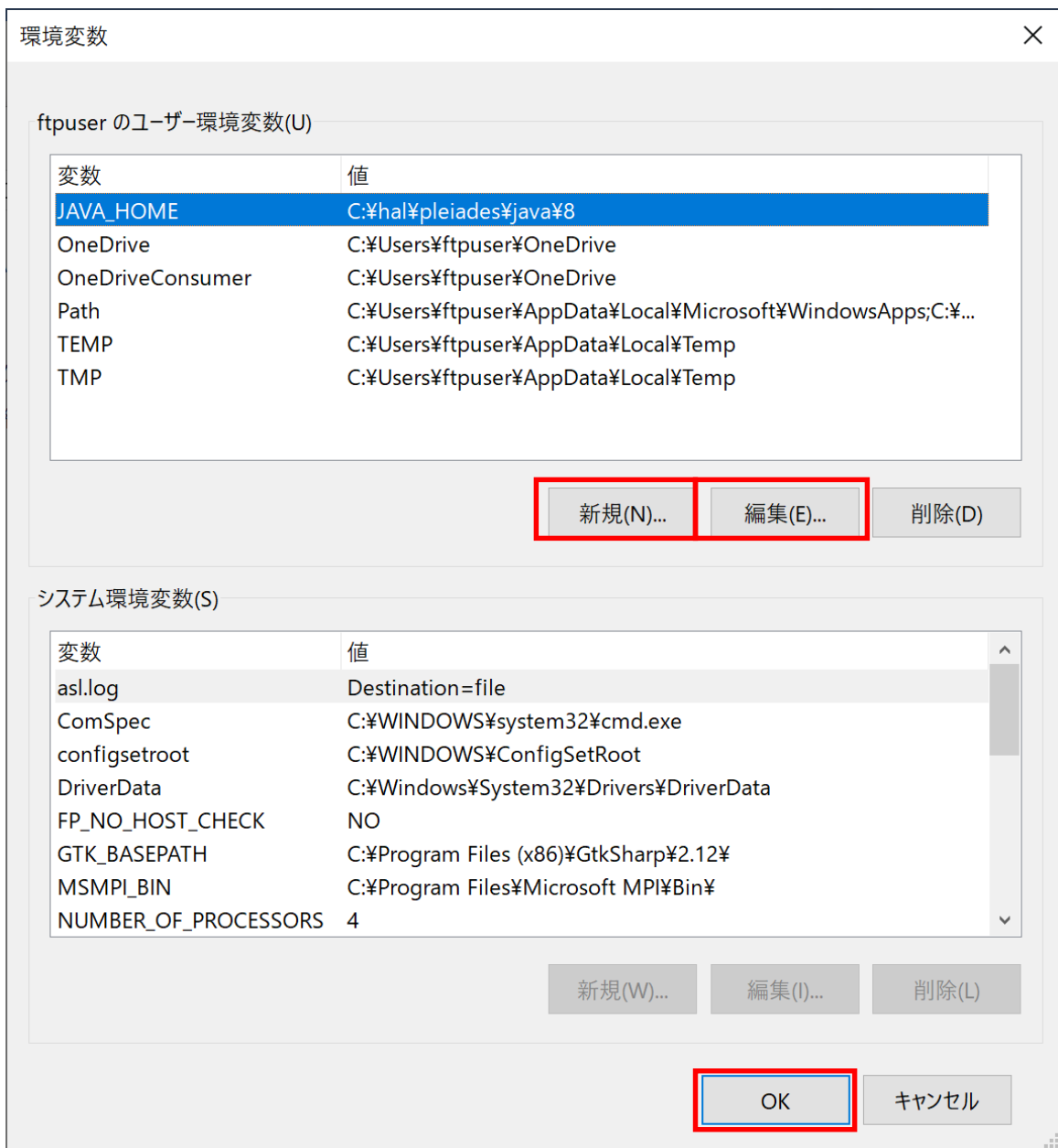
③ ユーザーアカウントをさらにクリックする。



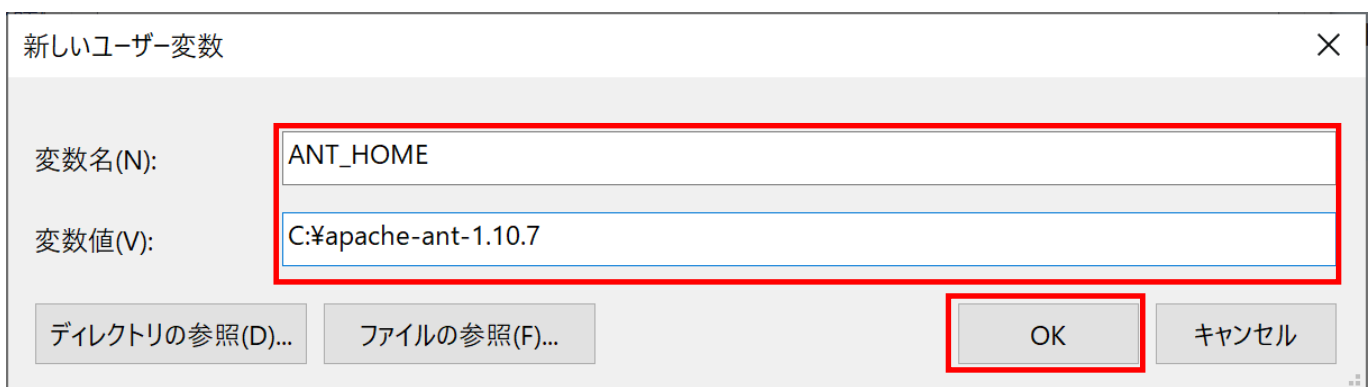
- ④ 「環境変数の変更」をクリックする。



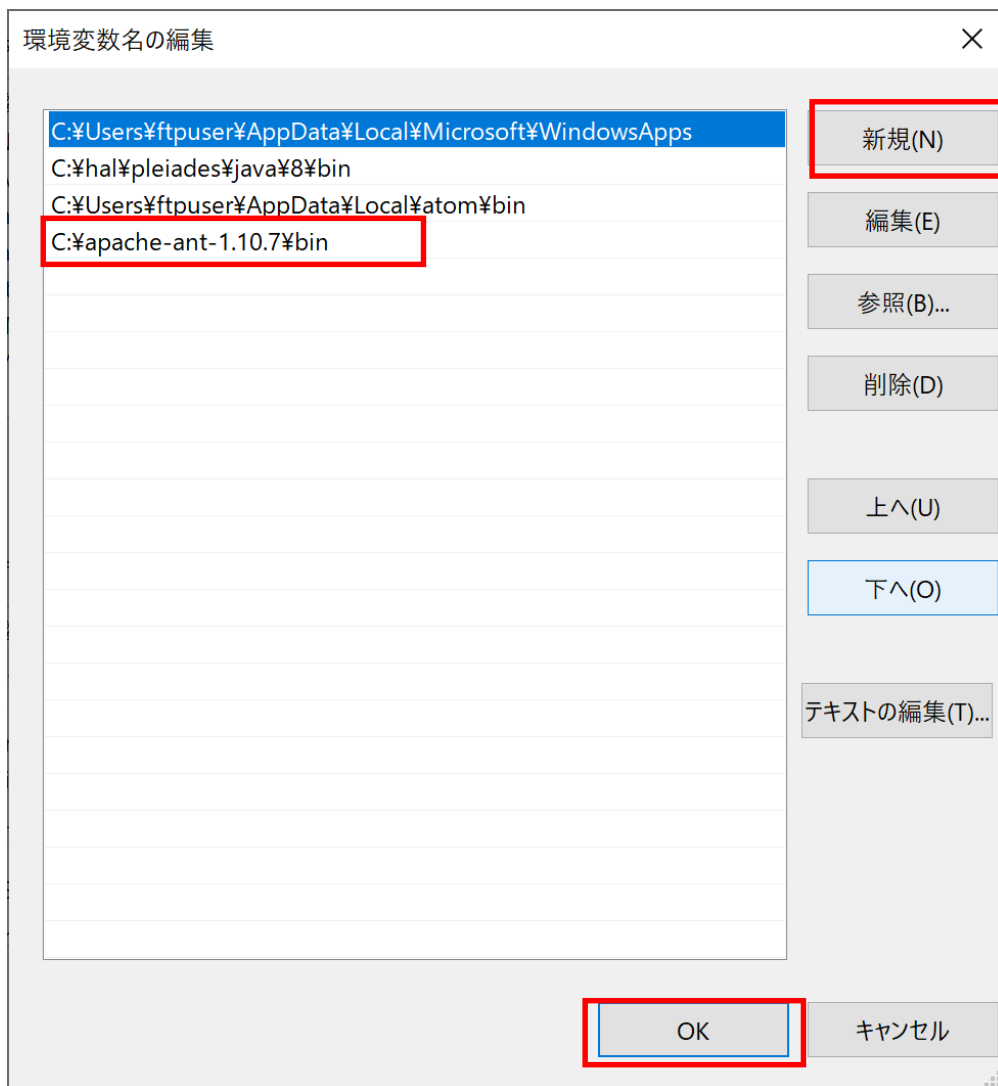
- ⑤ 「ユーザー環境変数」の新規ボタンをクリックする。



- ⑥ 「ANT_HOME」を新規設定する。

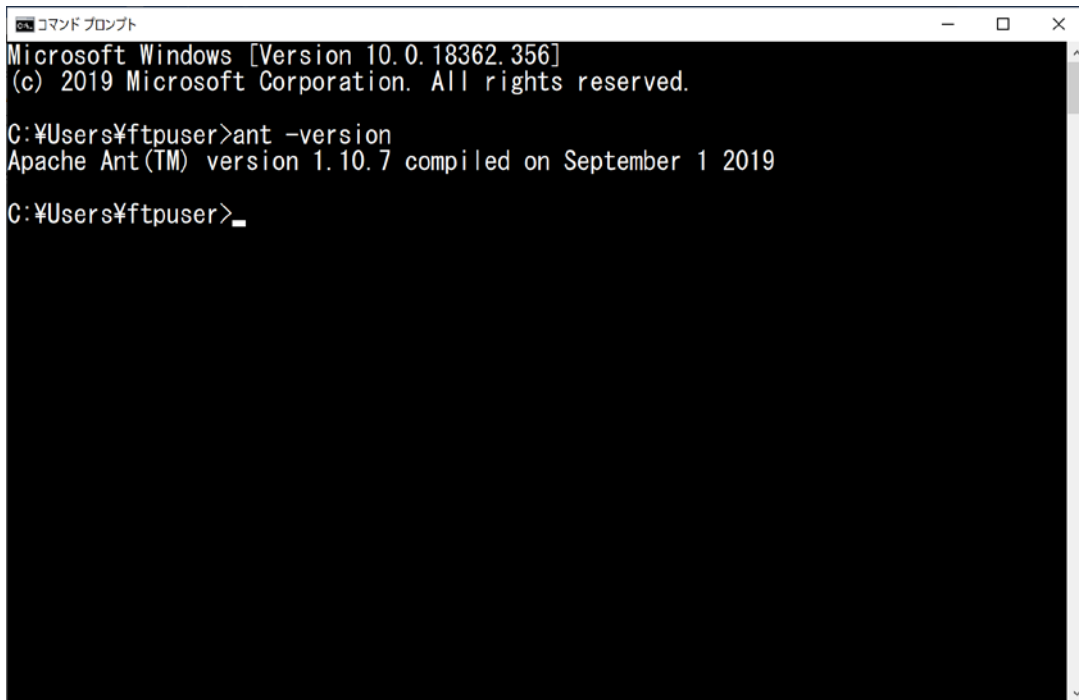


- ⑦ 「ユーザー環境変数」の編集ボタンをクリックし path に「C:¥apache-ant-1.10.7¥bin」を新規追加する



- ⑧ コマンドプロンプトを起動し、次のコマンドを実行する。Ant のバージョンが表示すれば Ant のインストールは成功。

>ant -version



```
コマンド プロンプト
Microsoft Windows [Version 10.0.18362.356]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\¥ftpuser>ant -version
Apache Ant(TM) version 1.10.7 compiled on September 1 2019

C:\Users\¥ftpuser>_
```


アプリケーション(jar)の開発環境を設定する

(1) jar プロジェクトを用意する。

まずは次のようなフォルダ構成を、c:\¥直下などに作成する。

フォルダ名とファイル名の大文字小文字は指示通り作成する。

```
PrintForm
├──src
│   └──main
│       Main.java           ※中身が空の java ファイルを作成する
├──resources
│   logback.xml             ※中身が空の xml ファイルを作成する
└──lib
    logback-classic-1.2.3.jar ※ファイルを指定の場所からコピーする
    logback-core-1.2.3.jar   ※ファイルを指定の場所からコピーする
    slf4j-api-1.7.25.jar     ※ファイルを指定の場所からコピーする
build.xml                   ※中身が空の xml ファイルを PrintForm フォルダ直下に作成する
```

(2) logback.xml を作成する。

文字コードを必ず「UTF-8」で保存する。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <!-- コンソール出力用 appender -->
  <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>

  <!-- ログファイル用 appender -->
  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <!-- ファイルは日替わりでローテーション -->
      <fileNamePattern>logs/log%d{yyyy-MM-dd}.log</fileNamePattern>
      <maxHistory>3</maxHistory>
    </rollingPolicy>
    <encoder>
      <!-- ファイル出力形式 -->
      <pattern>%d{yyyy-MM-dd' T' HH:mm:ss' Z'} - %m%n</pattern>
    </encoder>
  </appender>

  <!-- ロガー定義 -->
  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

(3) main.java を実装する。

文字コードを必ず「UTF-8」で保存する。

```
package main;

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

/**
 * Ant で HelloWorld の main 処理を持つクラス。<br>
 * <br>
 * @author tarosa0001
 */
public class Main {
    /** ロガー */
    private static Logger logger = LoggerFactory.getLogger(Main.class);

    /**
     * [main 処理]<br>
     * Ant で HelloWorld の main 処理。<br>
     * <br>
     * @param args コマンド引数
     */
    public static void main(String[] args) {
        logger.info("処理開始");

        logger.info("Ant で Hello World!!");

        logger.info("処理終了");
    }
}
```

(4) build.xml を定義する。

文字コードを必ず「UTF-8」で保存する。

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="printform" default="onejar">

    <property name="path.home"      location="." />
    <property name="path.lib"        location="${path.home}/lib" />
    <property name="path.src"        location="${path.home}/src" />
    <property name="path.bin"        location="${path.home}/bin" />
    <property name="path.release"    location="${path.home}/release" />
    <property name="jarName"         value="printform.jar" />
    <property name="mainClass"       value="main.Main" />
    <property name="title"           value="this title" />
    <property name="vendor"          value="this vendor" />

    <!-- implementationVersion にビルド日時を用いる例 -->
    <tstamp>
        <format property="version" pattern="yyyyMMddHHmmss" />
    </tstamp>

    <!--=====-->
    <!-- ライブラリのクラスも含めた jar ファイルを作成する -->
    <!--=====-->
    <target name="onejar" depends="clean, compile, copyresource, unjarlib">
        <mkdir dir="${path.release}" />
        <jar basedir="${path.bin}" destfile="${path.release}/${jarName}">
            <manifest>
                <attribute name="Main-Class" value="${mainClass}" />
                <attribute name="Implementation-Title" value="${title}" />
                <attribute name="Implementation-Vendor" value="${vendor}" />
                <attribute name="Implementation-Version" value="${version}" />
            </manifest>
        </jar>
    </target>

    <!--=====-->
    <!-- 作成したファイルを削除する -->
    <!--=====-->
    <target name="clean">
        <delete dir="${path.bin}" />
        <delete dir="${path.release}" />
    </target>

    <!--=====-->
    <!-- path.src 配下の java ファイルをコンパイルする -->
    <!--=====-->
    <target name="compile">
        <!--echo message"start compile..." -->
        <mkdir dir="${path.bin}" />
        <path id="classpath">
            <fileset dir="${path.lib}" includes="*.jar" />
            <!--<pathelement location="${path.bin}" />-->
        </path>
        <javac
            srcdir="${path.src}"
```

```

        destdir="${path.bin}"
        encoding="UTF8"
        includeantruntime="false"
        debug="on"
        debuglevel="lines, vars, source"
        classpathref="classpath" />
    </target>

    <!--=====-->
    <!-- path.lib 配下の jar ファイルをクラスファイルに展開する -->
    <!--=====-->
    <target name="unjarlib">
        <mkdir dir="${path.bin}" />
        <unjar dest="${path.bin}">
            <fileset dir="${path.lib}" includes="*.jar" />
        </unjar>
    </target>

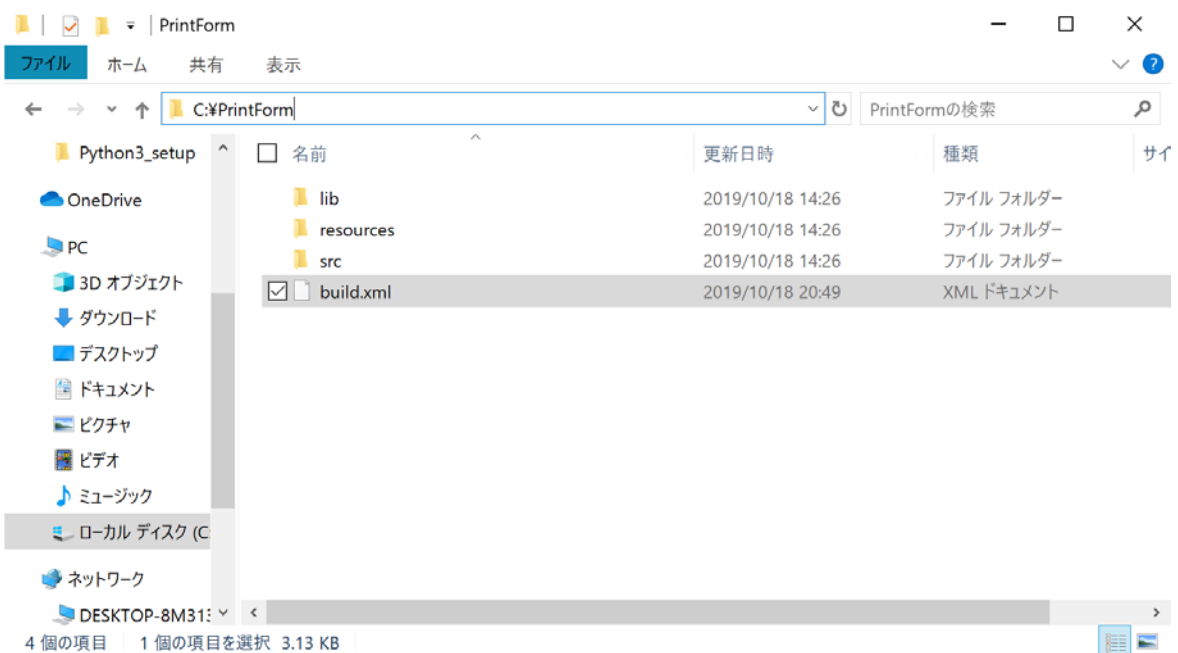
    <!--=====-->
    <!-- path.res、path.src 配下のリソースファイルを path.bin にコピーする -->
    <!--=====-->
    <target name="copyresource">
        <copy todir="${path.bin}">
            <fileset dir="${path.src}" excludes="*.java" />
        </copy>
    </target>

</project>

```

(5) Ant を実行する。

(ant 実行前)



```
> cd C:¥PrintForm
```

```
> ant -version
```

```
> ant
```

```
Microsoft Windows [Version 10.0.18362.356]
(c) 2019 Microsoft Corporation. All rights reserved.

C:¥Users¥ftpuser>cd C:¥PrintForm

C:¥PrintForm>ant -version
Apache Ant(TM) version 1.10.7 compiled on September 1 2019

C:¥PrintForm>ant
Buildfile: C:¥PrintForm¥build.xml

clean:

compile:
  [mkdir] Created dir: C:¥PrintForm¥bin
  [javac] Compiling 1 source file to C:¥PrintForm¥bin

copyresource:
  [copy] Copying 1 file to C:¥PrintForm¥bin

unjarlib:
  [unjar] Expanding: C:¥PrintForm¥lib¥logback-classic-1.2.3.jar into C:¥PrintForm¥bin
  [unjar] Expanding: C:¥PrintForm¥lib¥logback-core-1.2.3.jar into C:¥PrintForm¥bin
```

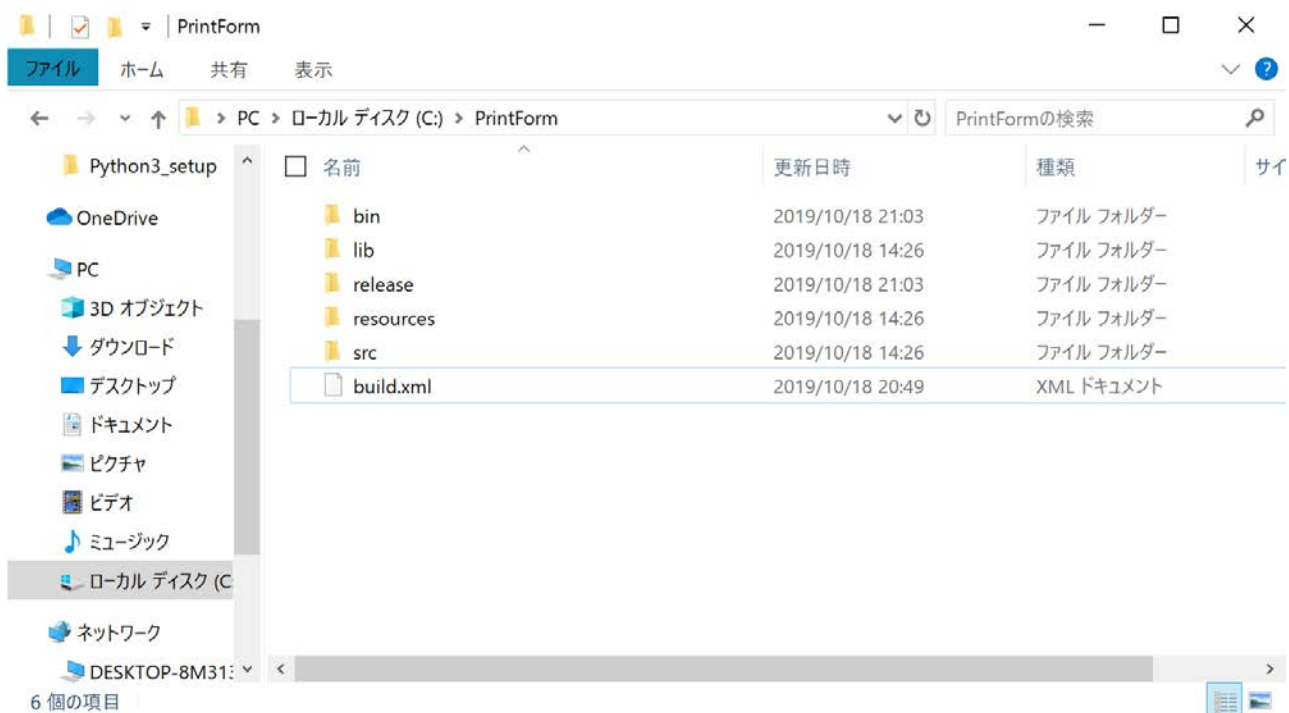
```
  [unjar] Expanding: C:¥PrintForm¥lib¥slf4j-api-1.7.25.jar into C:¥PrintForm¥bin

onejar:
  [mkdir] Created dir: C:¥PrintForm¥release
  [jar] Building jar: C:¥PrintForm¥release¥printform.jar

BUILD SUCCESSFUL
Total time: 9 seconds

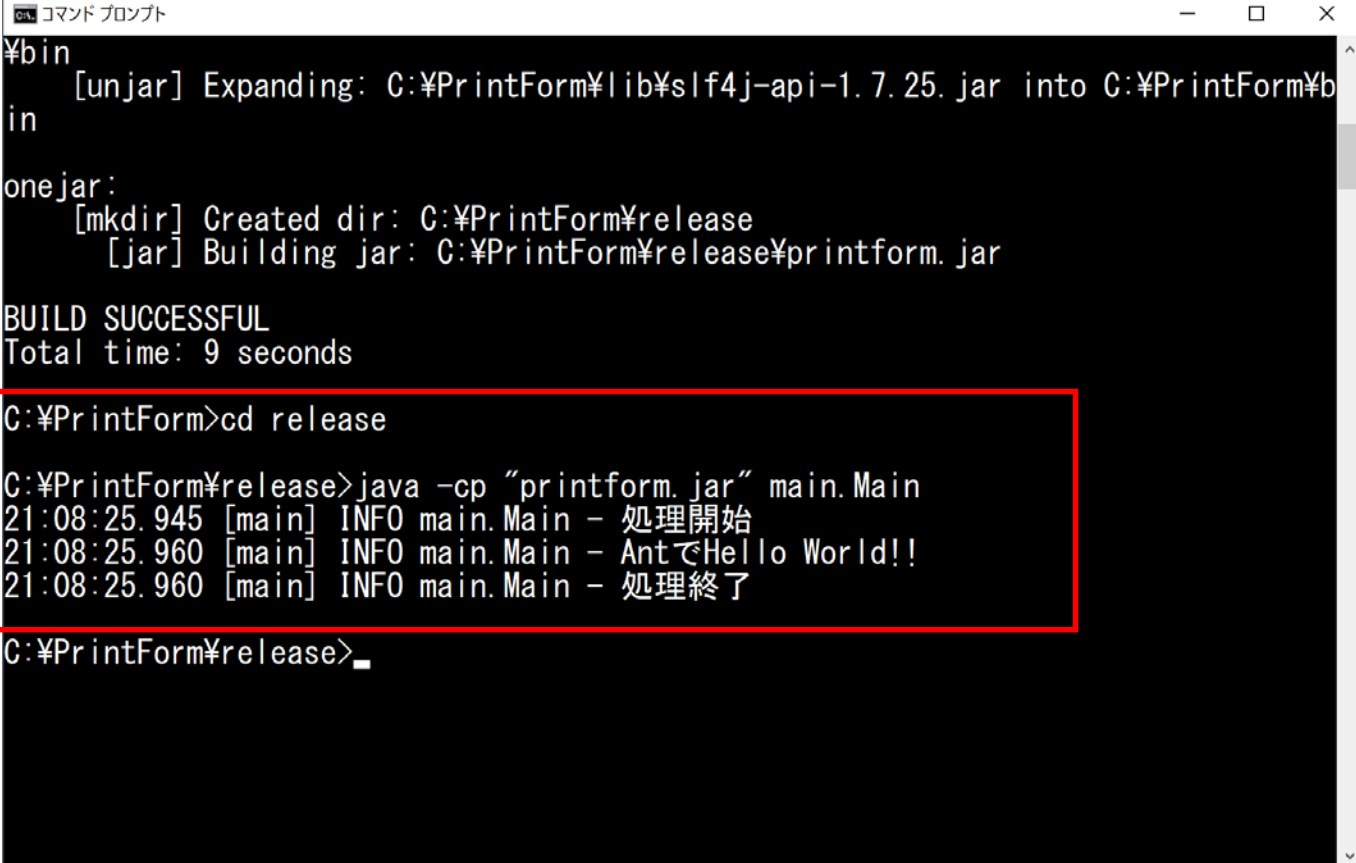
C:¥PrintForm>cd release
C:¥PrintForm¥release>_
```

(ant 実行後)



```
>cd release
```

```
>java -cp "printform.jar" main.Main
```



```
コマンド プロンプト
C:\¥PrintForm¥bin
[unjar] Expanding: C:\¥PrintForm¥lib¥slf4j-api-1.7.25.jar into C:\¥PrintForm¥bin
onejar:
  [mkdir] Created dir: C:\¥PrintForm¥release
  [jar] Building jar: C:\¥PrintForm¥release¥printform.jar

BUILD SUCCESSFUL
Total time: 9 seconds

C:\¥PrintForm>cd release

C:\¥PrintForm¥release>java -cp "printform.jar" main.Main
21:08:25.945 [main] INFO main.Main - 処理開始
21:08:25.960 [main] INFO main.Main - AntでHello World!!
21:08:25.960 [main] INFO main.Main - 処理終了

C:\¥PrintForm¥release>
```

上記のように、表示すれば成功。

WEB アプリケーション(war)の開発環境を設定する

(1) war プロジェクトを用意する。

```

RegisterFormWeb
├── src
│   ├── jp
│   │   ├── ac
│   │   │   ├── hal
│   │   │   │   ├── filter
│   │   │   │   │   ├── EncodingFilter.java ※ファイルを指定の場所からコピーする
│   │   │   │   │   └── ctrl
│   │   │   │   │       └── HelloWorldCtrl.java ※中身が空の java ファイルを作成する
│   └── WebContent
│       ├── jsp
│       │   ├── result.jsp ※中身が空の jsp ファイルを作成する
│       ├── META-INF
│       ├── WEB-INF
│       │   ├── lib
│       │   │   ├── mybatis-3.5.1.jar ※ファイルを指定の場所からコピーする
│       │   │   ├── mysql-connector-java-8.0.15.jar ※ファイルを指定の場所からコピーする
│       │   │   ├── jsp-api.jar ※ファイルを指定の場所からコピーする
│       │   │   └── servlet-api.jar ※ファイルを指定の場所からコピーする
│       ├── web.xml ※中身が空の xml ファイルを WEB-INF フォルダ直下に作成する
│       └── index.jsp ※中身が空の jsp ファイルを作成する
└── build.xml ※中身が空の xml ファイルを RegisterFormWeb フォルダ直下に作成する
  
```

(2) web.xml を作成する。

文字コードを必ず「UTF-8」で保存する。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>jQuery_Ajax_Request</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <servlet>
    <description>HelloWorld</description>
    <display-name>HelloWorldCtrl</display-name>
    <servlet-name>HelloWorldCtrl</servlet-name>
    <servlet-class>jp.ac.hal.ctrl.HelloWorldCtrl</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloWorldCtrl</servlet-name>
    <url-pattern>/HelloWorldCtrl</url-pattern>
  </servlet-mapping>

  <filter>
    <filter-name>Encoding</filter-name>
    <filter-class>jp.ac.hal.filter.EncodingFilter</filter-class>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>utf-8</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>Encoding</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

</web-app>
```

(3) HelloWorldCtrl.java を実装する。

文字コードを必ず「UTF-8」で保存する。

```
package jp.ac.hal.ctrl;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorldCtrl extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public HelloWorldCtrl() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // 呼び出し元 Jsp からデータ受け取り
        request.setCharacterEncoding("UTF8");
        String inputVal = request.getParameter("val");

        System.out.println("*****HelloWorldCtrl.doPost***** inputVal = " +
inputVal);

        // 呼び出し先 Jsp に渡すデータセット
        request.setAttribute("calcVal", inputVal + " + サブレットで追加");

        // result.jsp にページ遷移 → redirect の場合、データは Jsp に渡せない
        // response.sendRedirect("jsp/result.jsp");

        // result.jsp にページ遷移
        RequestDispatcher dispatch = request.getRequestDispatcher("jsp/result.jsp");
        dispatch.forward(request, response);
    }
}
```

(4) EncodingFilter.java の解説。

文字化けを防ぐために Filter を実装した。今回はコピーで OK。
web.xml に設定することによって、Web アプリ起動時に最初に起動する。

```
package jp.ac.hal.filter;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class EncodingFilter implements Filter {
    private String encoding = null;

    public void init(FilterConfig config) throws ServletException {
        encoding = config.getInitParameter("encoding");
    }

    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws ServletException, IOException {
        req.setCharacterEncoding(encoding);
        res.setContentType("text/html; charset=UTF-8");
        chain.doFilter(req, res);
    }

    public void destroy() {
        encoding = null;
    }
}
```

(5) index.jsp を実装する。

文字コードを必ず「UTF-8」で保存する。

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>入力画面</title>
</head>
<body>
    <form action="<%=request.getContextPath()%>/HelloWorldCtrl" method="post">
        入力してください。
        <br />
        <input type="text" name="val" value="" />
        <br />
        <input type="submit" value="送信する" />
    </form>
</body>
</html>
```

(6) result.jsp を実装する。

文字コードを必ず「UTF-8」で保存する。

```
<%@ page language="java" contentType="text/html; charset=UTF8"
    pageEncoding="UTF8"%>
<%
    // Servlet のデータ受け取り
    request.setCharacterEncoding("UTF8");
    String strCalc = (String) request.getAttribute("calcVal");
    System.out.println("#####result.jsp strCalc = " + strCalc);
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF8">
<title>PAGE</title>
</head>
<body>
    Servlet で処理した結果を表示 :
    <%=strCalc%>
</body>
</html>
```

(6) build.xml を定義する。

文字コードを必ず「UTF-8」で保存する。

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="RegisterFormWeb" default="war">

    <path id="compile.classpath">
        <fileset dir="WebContent/WEB-INF/lib">
            <include name="*.jar"/>
        </fileset>
    </path>

    <!--=====-->
    <!-- build、dist フォルダを作成する -->
    <!--=====-->
    <target name="init">
        <mkdir dir="build/classes"/>
        <mkdir dir="dist" />
    </target>

    <!--=====-->
    <!-- src 配下の java ファイルをコンパイルする -->
    <!--=====-->
    <target name="compile" depends="init" >
        <javac destdir="build/classes" debug="true" srcdir="src" includeantruntime="false">
            <classpath refid="compile.classpath"/>
        </javac>
    </target>

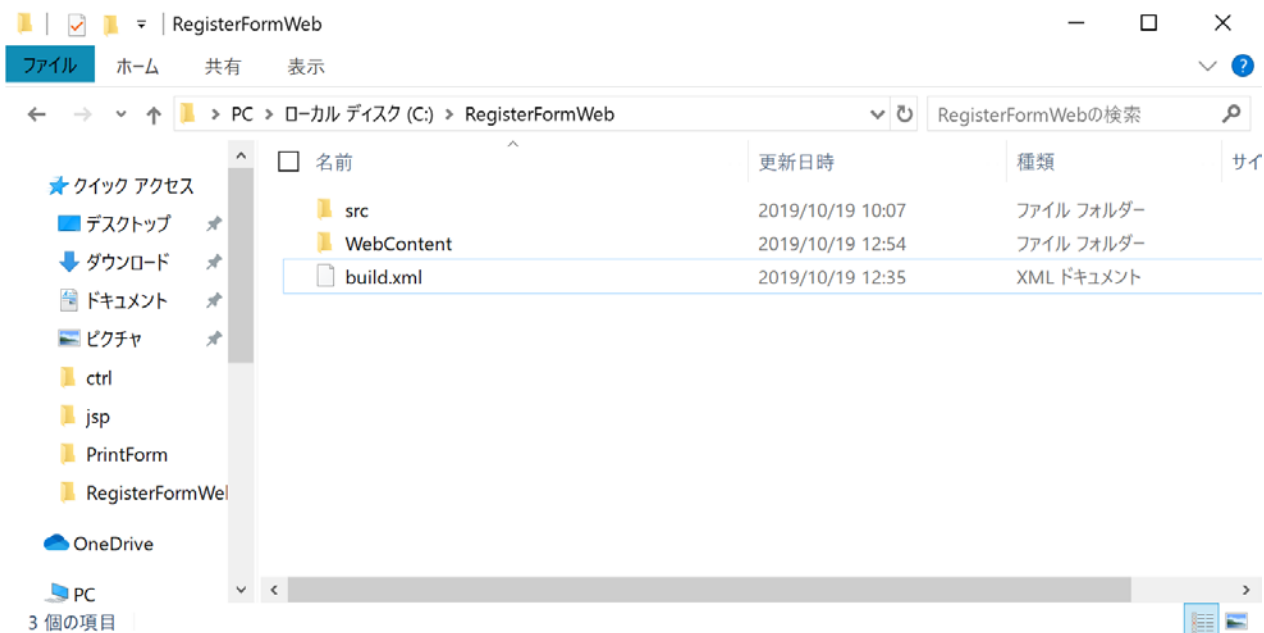
    <!--=====-->
    <!-- ライブラリのクラスも含めた war ファイルを作成する -->
    <!--=====-->
    <target name="war" depends="compile">
        <war destfile="dist/registerform.war" webxml="WebContent/WEB-INF/web.xml">
            <fileset dir="WebContent"/>
            <lib dir="WebContent/WEB-INF/lib"/>
            <classes dir="build/classes"/>
        </war>
    </target>

    <!--=====-->
    <!-- 作成したファイルを削除する -->
    <!--=====-->
    <target name="clean">
        <delete dir="dist" />
        <delete dir="build" />
    </target>

</project>
```

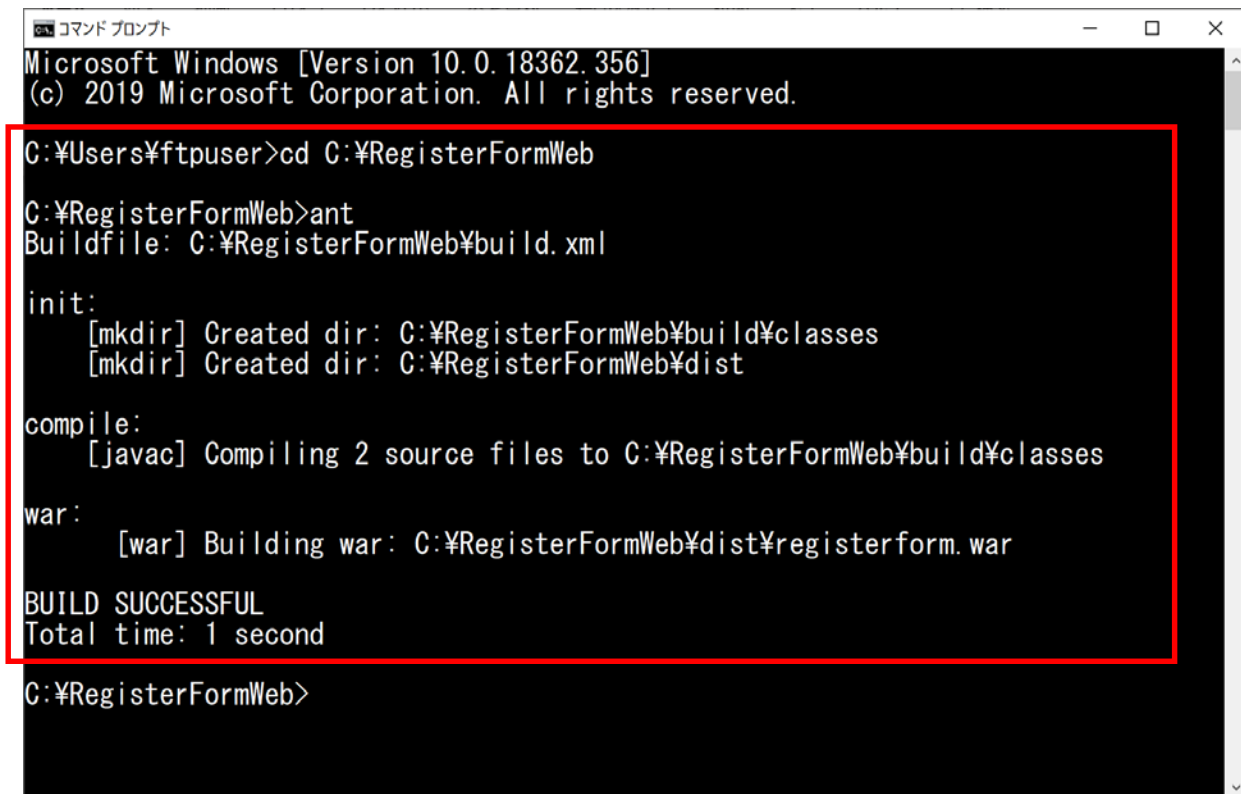
(7) Ant を実行する。

(ant 実行前)



```
> cd C:¥RegisterFormWeb
```

```
> ant
```

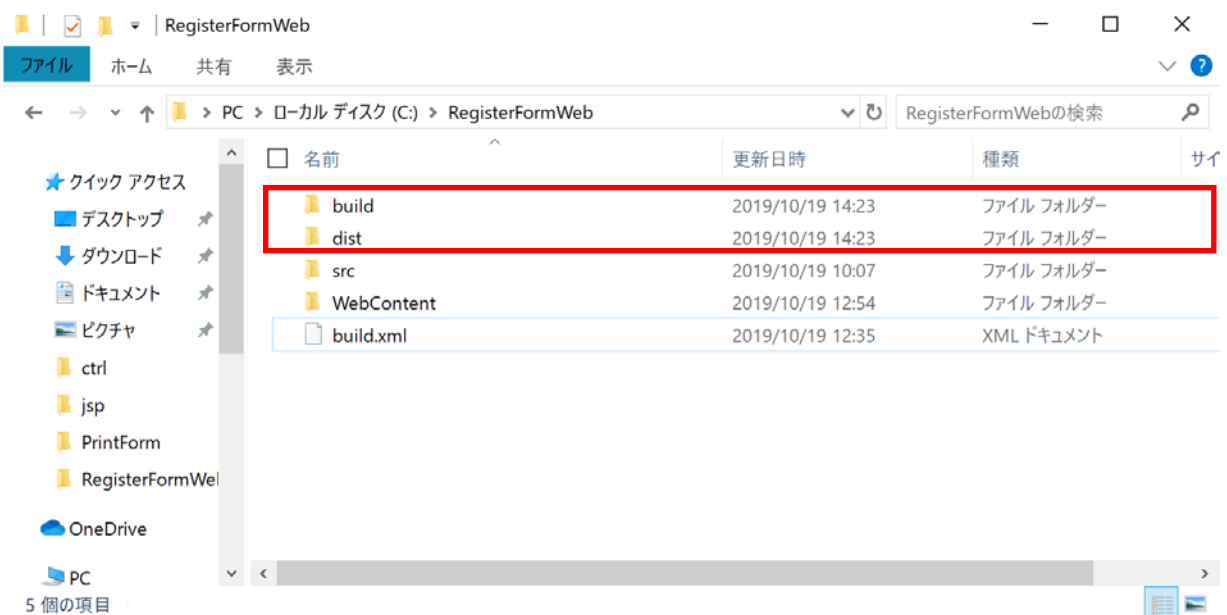


コマンド プロンプト

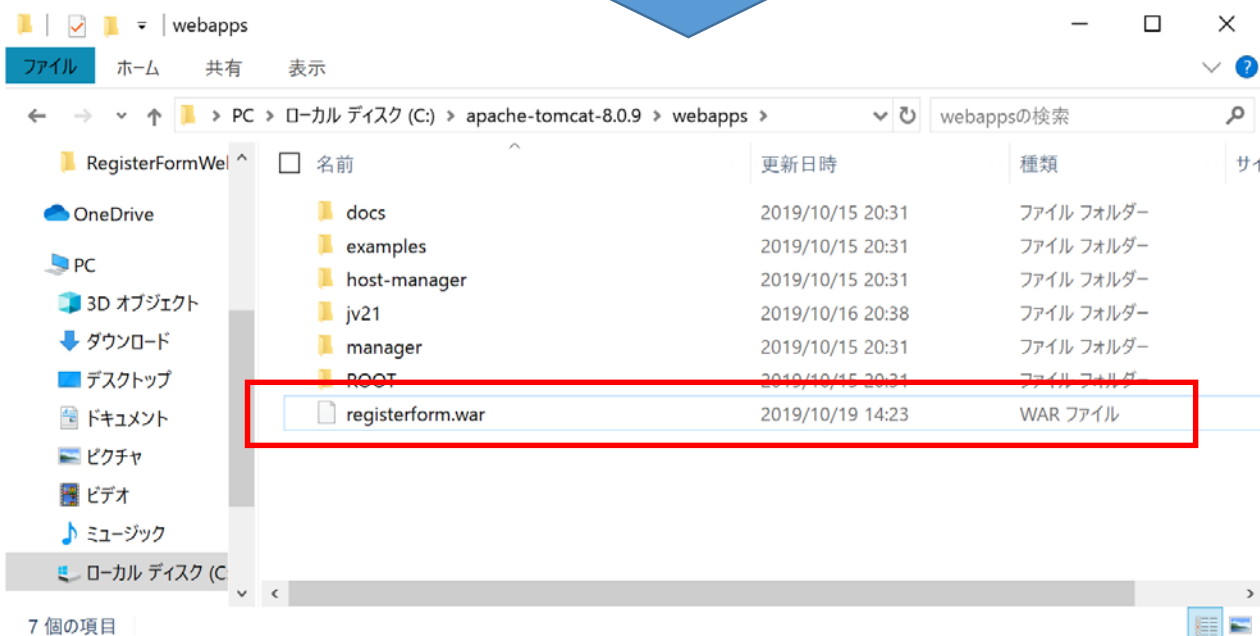
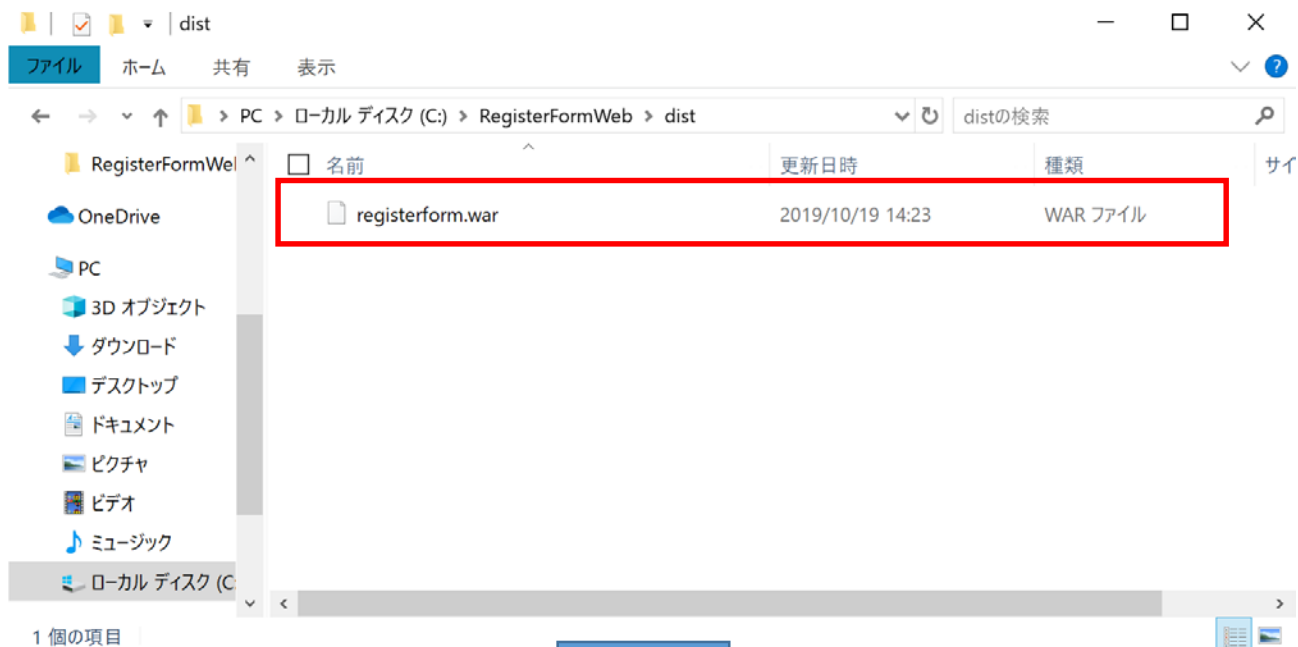
Microsoft Windows [Version 10.0.18362.356]
(c) 2019 Microsoft Corporation. All rights reserved.

```
C:¥Users¥ftpuser>cd C:¥RegisterFormWeb  
C:¥RegisterFormWeb>ant  
Buildfile: C:¥RegisterFormWeb¥build.xml  
  
init:  
[mkdir] Created dir: C:¥RegisterFormWeb¥build¥classes  
[mkdir] Created dir: C:¥RegisterFormWeb¥dist  
  
compile:  
[javac] Compiling 2 source files to C:¥RegisterFormWeb¥build¥classes  
  
war:  
[war] Building war: C:¥RegisterFormWeb¥dist¥registerform.war  
  
BUILD SUCCESSFUL  
Total time: 1 second  
  
C:¥RegisterFormWeb>
```

(ant 実行後)



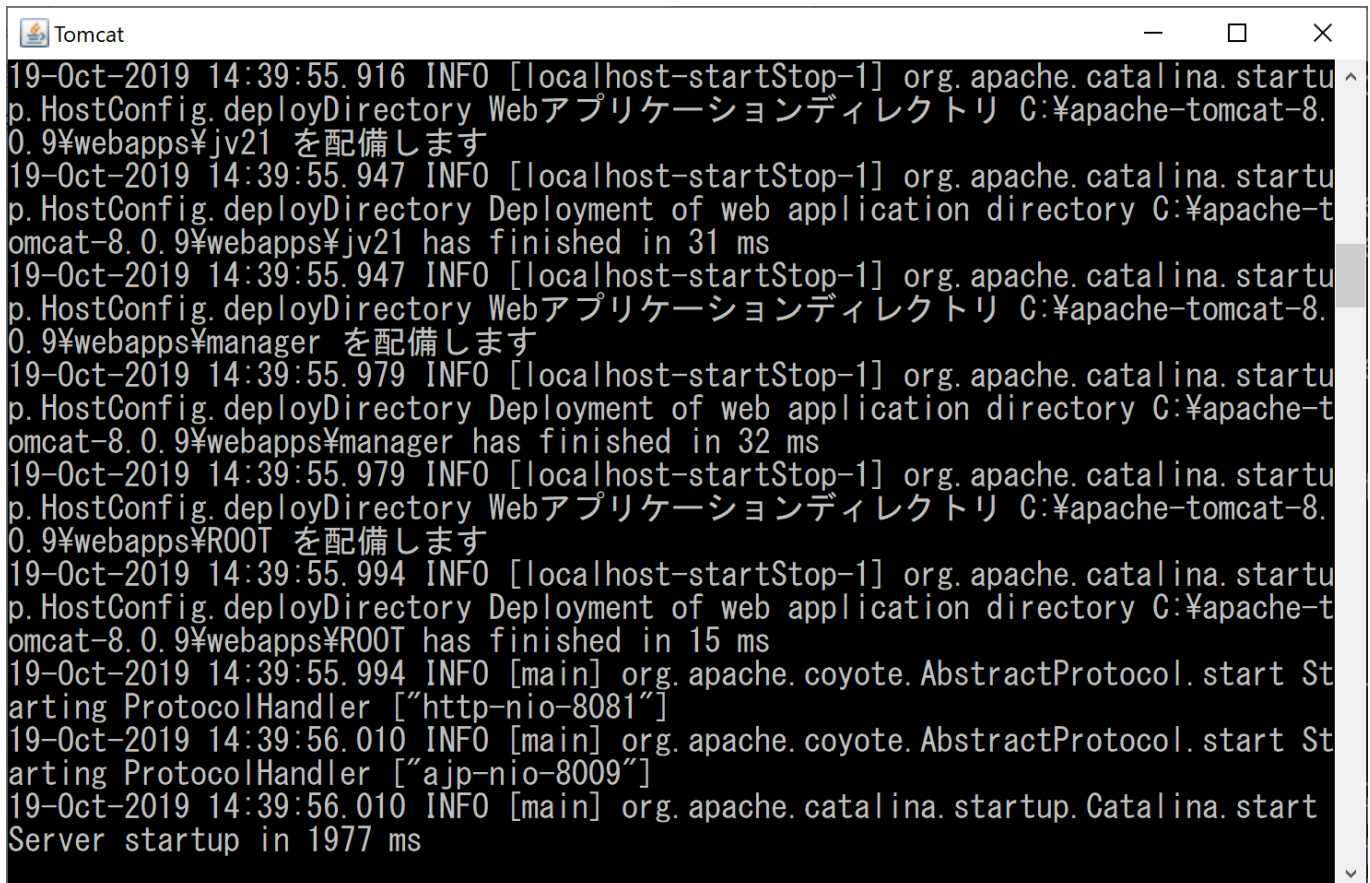
- (8) Dist の中に Web アプリケーション (war) がビルドされる。
war を「C:\¥apache-tomcat-8.0.9¥webapps」にコピーする。



(9) Tomcat を起動する。

> C:\¥apache-tomcat-8.0.9¥bin

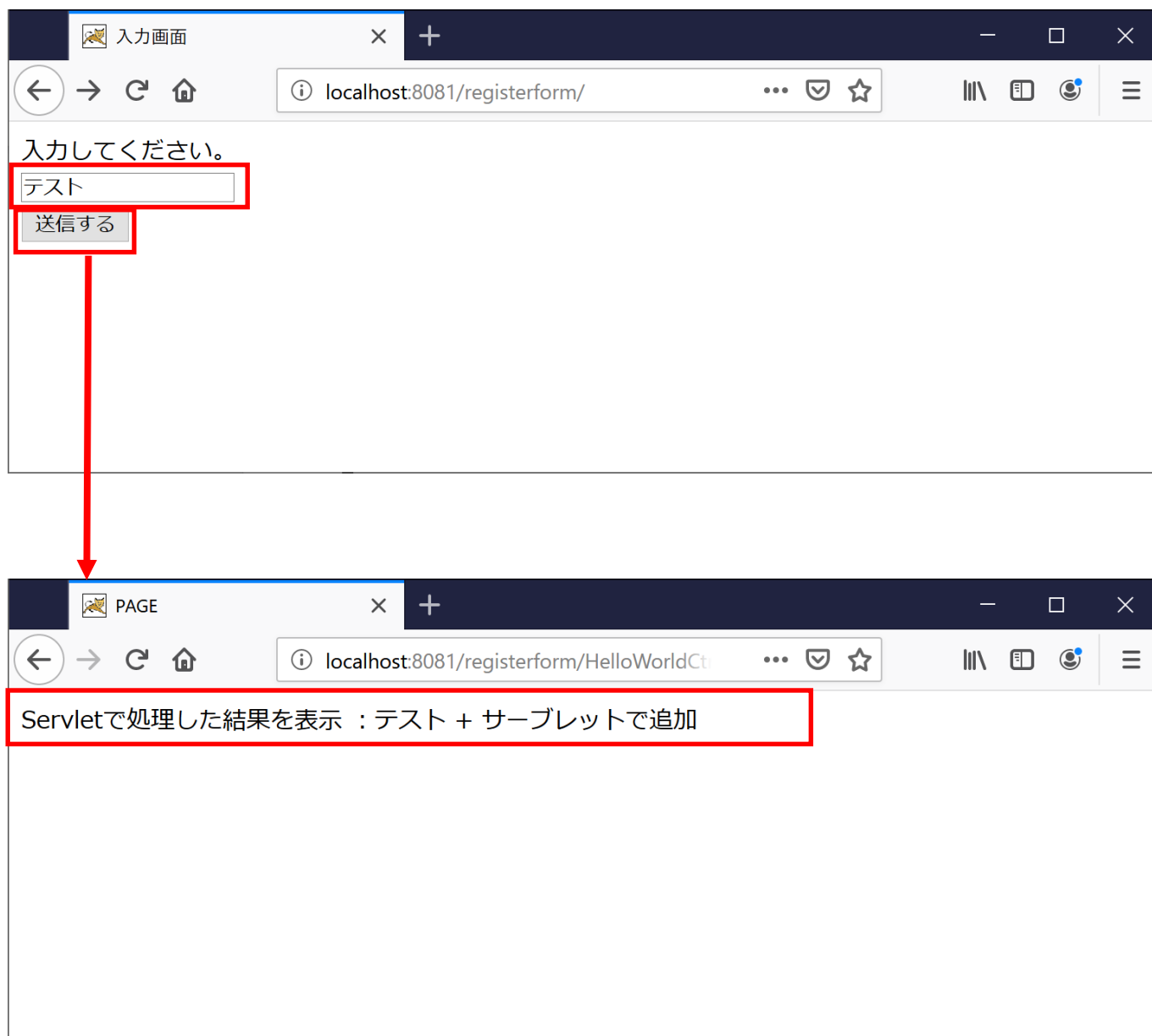
> startup.bat

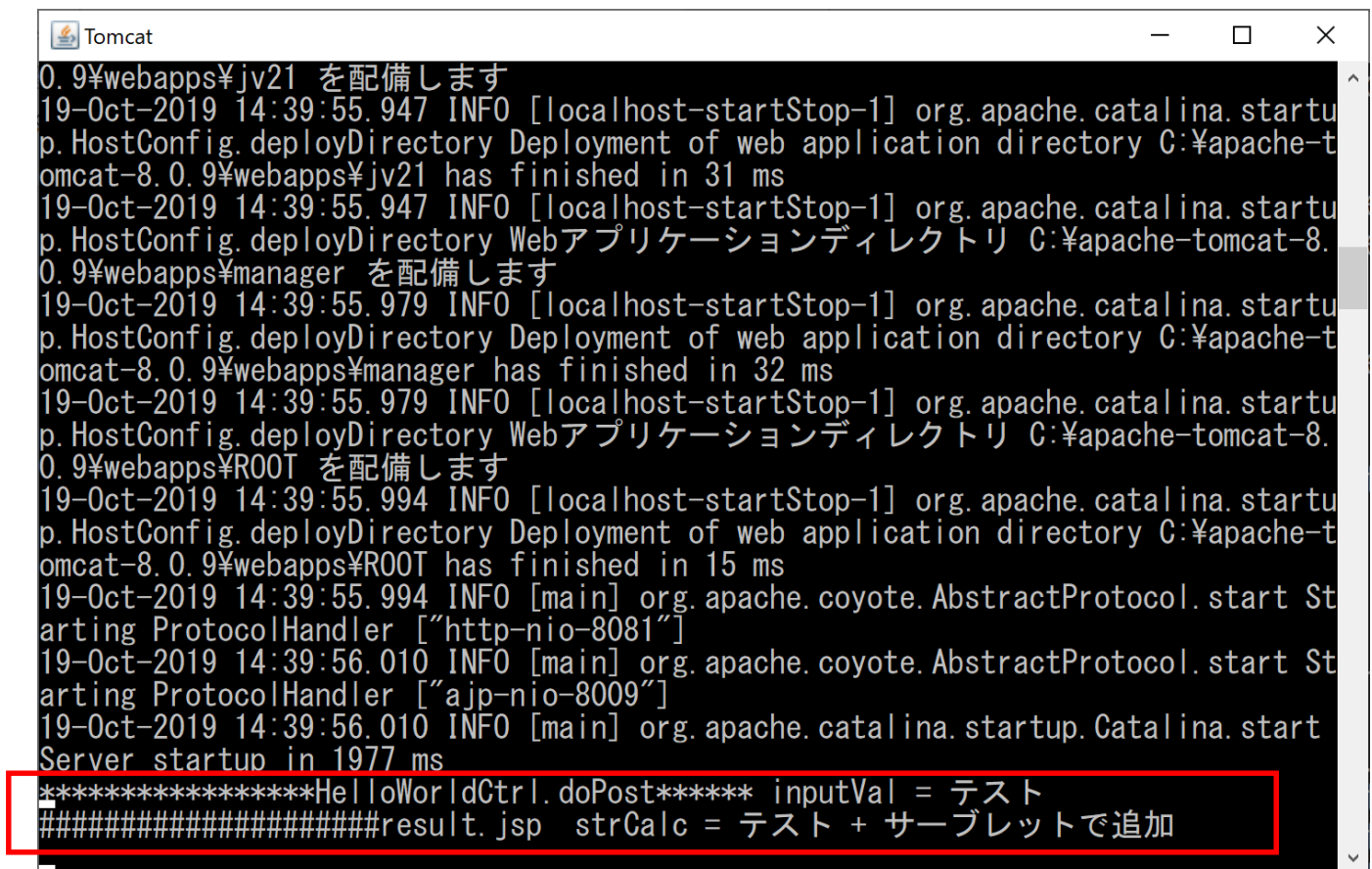


```
Tomcat
19-Oct-2019 14:39:55.916 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Webアプリケーションディレクトリ C:\¥apache-tomcat-8.
0.9¥webapps¥jv21 を配備します
19-Oct-2019 14:39:55.947 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Deployment of web application directory C:\¥apache-t
omcat-8.0.9¥webapps¥jv21 has finished in 31 ms
19-Oct-2019 14:39:55.947 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Webアプリケーションディレクトリ C:\¥apache-tomcat-8.
0.9¥webapps¥manager を配備します
19-Oct-2019 14:39:55.979 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Deployment of web application directory C:\¥apache-t
omcat-8.0.9¥webapps¥manager has finished in 32 ms
19-Oct-2019 14:39:55.979 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Webアプリケーションディレクトリ C:\¥apache-tomcat-8.
0.9¥webapps¥ROOT を配備します
19-Oct-2019 14:39:55.994 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Deployment of web application directory C:\¥apache-t
omcat-8.0.9¥webapps¥ROOT has finished in 15 ms
19-Oct-2019 14:39:55.994 INFO [main] org.apache.coyote.AbstractProtocol.start St
arting ProtocolHandler ["http-nio-8081"]
19-Oct-2019 14:39:56.010 INFO [main] org.apache.coyote.AbstractProtocol.start St
arting ProtocolHandler ["ajp-nio-8009"]
19-Oct-2019 14:39:56.010 INFO [main] org.apache.catalina.startup.Catalina.start
Server startup in 1977 ms
```

(10) ブラウザから次の URL を入力し WEB アプリを起動する。

`http://localhost:8081/registerform/`





```
Tomcat
0.9¥webapps¥jv21 を配備します
19-Oct-2019 14:39:55.947 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Deployment of web application directory C:¥apache-t
omcat-8.0.9¥webapps¥jv21 has finished in 31 ms
19-Oct-2019 14:39:55.947 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Webアプリケーションディレクトリ C:¥apache-tomcat-8.
0.9¥webapps¥manager を配備します
19-Oct-2019 14:39:55.979 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Deployment of web application directory C:¥apache-t
omcat-8.0.9¥webapps¥manager has finished in 32 ms
19-Oct-2019 14:39:55.979 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Webアプリケーションディレクトリ C:¥apache-tomcat-8.
0.9¥webapps¥ROOT を配備します
19-Oct-2019 14:39:55.994 INFO [localhost-startStop-1] org.apache.catalina.startup
p.HostConfig.deployDirectory Deployment of web application directory C:¥apache-t
omcat-8.0.9¥webapps¥ROOT has finished in 15 ms
19-Oct-2019 14:39:55.994 INFO [main] org.apache.coyote.AbstractProtocol.start St
arting ProtocolHandler ["http-nio-8081"]
19-Oct-2019 14:39:56.010 INFO [main] org.apache.coyote.AbstractProtocol.start St
arting ProtocolHandler ["ajp-nio-8009"]
19-Oct-2019 14:39:56.010 INFO [main] org.apache.catalina.startup.Catalina.start
Server startup in 1977 ms
*****HelloWorldCtrl.doPost***** inputVal = テスト
#####result.jsp strCalc = テスト + サーブレットで追加
```

上記のように表示すれば成功。

配列

配列とは同じ型のデータを複数個まとめて番号を付けたもの。配列の要素数は生成時に決める必要があり、途中で増やすことはできない。配列の添字は 0 から始まります (0-オリジン)。

1. 配列名を宣言する。

```
int[] a;          // int 型の配列 a を宣言
```

2. 配列の要素数を指定して、配列を格納するメモリ領域を確保する。
new はメモリ領域を確保する命令。

```
a = new int[5]; // int 型の要素を 5 個確保
```

3. 各要素に値を設定する。各要素は添字を使って指定する。

```
a[0] = 10;        // 添字は 0 から始める  
a[1] = a[0];      // このようにすると a[1] に a[0] の値が複写される
```

範囲外のアクセス

以下のように、上記 2. で確保した範囲を越えてアクセスすると、実行時に `ArrayIndexOutOfBoundsException` という例外が発行される。

```
× a[5] = 20;      // これはエラー。a の要素は 0 から 4 までの 5 個なので
```

※ このエラーはコンパイル時ではなく、実行時に検出される。添字の値は一般に実行してみないとわからないので、コンパイル時には評価しないから。

まとめた記述

上述の 1. と 2. とをまとめて以下の 01: のように一度で書くこともできる。また 1. ~3. をまとめて 02: または 03: のように書くこともできます。

```
01: int[] a = new int[5];  
02: int[] a = new int[] { 10, 20, 30, 40, 50 };  
03: int[] a = { 10, 20, 30, 40, 50 };
```

別の書き方

配列の記述の仕方には `int a[]` のように変数名側に角括弧を付ける書き方もある。このやり方で上の例を書くと以下になる。C 言語などに慣れた人はこの流儀で書くことが多い。この場合でも単独で new する時は 06: のように変数側には `[]` は付けないので注意すること。07: のように書くとコンパイルエラーになる。

```
04: int a[];  
05: int a[] = new int[5];  
06: a = new int[5];  
07: × a[] = new int[5];      // これはコンパイルエラー
```

参照型

基本データ型では変数域には値そのものが入るが、配列の場合は、配列の実体のデータではなく、格納域のアドレスが入る。したがって、以下のように配列 a を別の配列 b に代入すると、格納域へのアドレスが複写され、b も a と同じ実体である {10, 20, 30} を指すことになります。すなわち a[0]=15 とすると b[0] も 15 になる。配列の値をそっくり別の領域に移したいときには、1 つずつとりだしてコピーする必要がある。

```
int[] a = {10, 20, 30};
int[] b = {40, 50};
b = a;           // bには{10, 20, 30}へのアドレスが代入される
a[0] = 15;       // {10, 20, 30}が{15, 20, 30}になる
```

このように変数域に（データではなく）アドレスが入っている変数の型を、参照型と言う。Java では基本データ型以外の変数はすべて参照型。参照型のデータは new で領域を確保する必要がある。Java 学習の中心テーマである「クラス」も参照型。

※ String も変数域にアドレスが入っているので実は参照型。しかし String は良く使うので、特例として new なしで生成できるようになっている。実際、String str = new String("abc"); のように書くこともできる。

配列の要素数

配列 a の要素数は次のようにして取得できる。

```
a.length;
```

多次元配列

上述の説明は 1 次元の配列でしたが、2 次元以上の配列も宣言できる。以下に 2 次元配列の使い方の例を示す。

```
int[][] a = new int[3][4];
a[0][0] = 100;
a[0][1] = 200;
```

演算子

Javaには多くの演算子がある。変数を演算子で結びつけたものを式と言う。式の中では丸括弧（ ）で演算の順序を変更することができる。

算術演算

算術演算は数値を計算する。Javaでは以下の算術演算が可能。整数除算を行うと余りは切り捨てられる。整数を0で割るとArithmeticException例外が発生する。その他加減算より乗除算が優先して行われるなどの規則は一般の数式と同様。

+ (加算)
- (減算)
* (乗算)
/ (除算)
% (剰余)

※ 以下に演算の例を示す。

```
c = a * b;  
d = (e - f) / (e + f);
```

比較演算

比較演算は主にif文の条件の中で使われる。「等しい」は == で表し、「等しくない」は != で表現する。これらはC言語に由来するもの。比較演算の結果はboolean型になる。

< (小さい)
> (大きい)
<= (小さいか等しい)
>= (大きいか等しい)
== (等しい)
!= (等しくない)

論理演算

論理積と論理和には以下に示すように2種類ある。&& と || は左から右に評価し、かつ条件が整えばそこで判定を中止することが保証されている。これを活用すれば処理の効率を向上させることができる。

& および && (論理積)
| および || (論理和)
^ (排他的論理和)
! (否定)

ビット演算

ビット列を扱う演算ができる。上記の `&` `|` `^` もビット列演算に使うことができる。

`<<` (左シフト)
`>>` (符号付き右シフト)
`>>>` (符号無し右シフト)
`~` (反転)

例： `a=0xF6D3` のとき、`a<<1` は `0xEDA6`、`a>>3` は `0xFEDA`、`a>>>2` は `0x3DB4`

インクリメント演算／デクリメント演算

Java では入力を少しでも減らそうとして以下のような、特有の演算が用意されている。最初は違和感があるかも知れないが、使い慣れるとなかなか便利。

`++a` (前置型)、`a++` (後置型) : いずれも `a = a + 1` のこと
`--b` (前置型)、`b--` (後置型) : いずれも `b = b - 1` のこと

★ 前置型では `+1` されてから値が式に取り込まれ、後置型では値が式に取り込まれてから `+1` される。例えば `a=++b;` と書くと `b` を `+1` してから `b` を `a` に代入するが、`a=b++;` と書くと `b` を `a` に代入した後で `b` を `+1` する。

代入演算

以下も Java 特有の演算。これらはいずれも C 言語の流れをくむもの。

`a += b` : `a = a + b` を表す
`a -= b` : `a = a - b` を表す
`a *= b` : `a = a * b` を表す
`a /= b` : `a = a / b` を表す
`a %= b` : `a = a % b` を表す

条件演算

以下のように記述する。 `a` の真偽を評価し真なら `b` を、偽なら `c` を採用する。

`a ? b : c;`

例： `b=(a>0)?10:20;` `a` が正なら `b` に `10` を代入し、`a` が正でないなら `b` に `20` を代入する。

2 種類の等価演算

String 型などの参照型（変数域にデータ格納先アドレスが入っている型）のデータを比較するときは、次の 2 種類のやり方がある。以下では（1）が真なら、当然（2）も真。

- (1) `a == b` アドレスの値を比較
- (2) `a.equals(b)` アドレスが指している実体を比較

文字列などの比較は、普通はアドレスそのものではなく、文字列として同じか否かを比較したいので（2）の方式を使う。

制御文

制御文は処理の流れを変更するもの。ここで述べる制御文の多くはダイクストラが提唱した構造化プログラミングの考えに基づいて構成された。goto レス、例外処理などとともに、最近の多くのプログラミング言語で採用されている。各制御文には固有の構造がある。以下に主な制御文の機能とその構造を示す。

if 文

if 文の構造

```
if (条件) {  
    文  
} else {  
    文  
}
```

条件には boolean 型の式を入れる。中括弧 { } の位置が特徴的ですが、これは構造を見やすくするための工夫で、Java の言語仕様で定めているものではない。中括弧始め { を行頭に置く流儀もありますが、最近は上記のスタイルを使う人が多い。

以下は a が偶数なら b に “even” を、奇数なら “odd” をセットする例。

```
if (a % 2 == 0) {  
    b = “even”;  
} else {  
    b = “odd”;  
}
```

省略形

else 以下は省略可能。また上の例のように文が 1 つしかないときは、中括弧も省略して次のように記述することもできる。

```
if (a % 2 == 0) b = “even”;  
else b = “odd”;
```

if 文が多重に使われるときに else 句を省略すると if と else との対応が曖昧になりがちですが「else は直近の (else を従えていない) if に対応する」というルールに従う。

else if 形式

次のように else if 文を繰り返す形式も可能。この構造では複数の条件が満たされても最初に満たした条件に対応する文しか実行されない。

```
if (条件 1) {  
    文 A  
} else if (条件 2) {  
    文 B  
    ...  
} else if (条件 n) {  
    文 Y  
} else {  
    文 Z  
}
```

for 文

for 文の構造

```
for (初期化; 条件; 値の更新) {  
    文  
}
```

for 文は繰り返し処理時によく使う機能を備えた巧妙な構造になっている。上記の for 文は次のように実行される。

- (1) 初期化の実施
- (2) 条件の判定→条件が否定されたら for 文から抜ける
- (3) 文の実行
- (4) 値の更新の実施

以降 (2) ~ (4) の繰り返し。

以下は 1 から 10 までの数の足し算をするプログラムの例。

```
int i, sum=0;  
for (i=1; i<=10; i++) {  
    sum = sum + i;  
}
```

※ 「条件が真の間だけ繰り返す」ことに注意。

for 文内での変数の型宣言

次の例のようにループで使う変数を for 文の () の中で型宣言することができる。このようにするとこの変数 *i* は for 文の構造の中だけで有効。すなわち for 文の外部で *i* という変数を使ってもそれとは無関係にローカルな変数として *i* が利用できる。

```
for (int i=0; i<10; i++)      // int をここで書くとローカル変数
```

記述の自由度

for 文の () の中の式は同じ変数を使う必要はない。プログラムとして間違っていなければどんなものを書いてもかまわない（理解しづらくなり保守性は下がるかもしれないが）。また記述を省略したり、逆に複数の命令を書いてもかまわない。

```
for (i=0; a!=0; b++)    // このようなものでも可
for (; i<10; )          // 初期化済で値の更新は{ }内でやるならこれでも可
for (i=0, j=10; i<10; i++, j--)  // 複数の処理を書いても可
```

拡張 for 文

拡張 for 文の構造

```
for (型 変数名 : 配列名) {
    文
}
```

拡張 for 文は J2SE5.0 で導入された新しい構文。上記で配列名の部分には単なる配列だけでなくコレクションも指定できる。拡張 for 文は配列に含まれるすべての要素を順次取り出して 1 つずつ処理するために使われる。上の拡張 for 文は次のように実行される。

- (1) 配列から値を 1 つ取り出し、変数名で示される変数に代入
- (2) 文の実行

以降、配列の全要素の数だけ (1) ~ (2) の繰り返し。

※ コレクションについては別途説明します。

拡張 for 文の例を示す。以下で *sum* は int 配列 *x* の全要素の和である 100 になる。

```
int[] x = { 10, 20, 30, 40 };
int sum = 0;
for (int a: x) {
    sum = sum + a;
}
```

while 文

while 文の構造

```
while (条件) {  
    文  
}
```

while 文も繰り返し処理を行うときに使う。上記の while 文は次のように実行される。最初に判定を行うので、文が一度も実行されないこともある。

(1) 条件の判定→条件が否定されたら while 文から抜ける

(2) 文の実行

以降 (1) ~ (2) の繰り返し。

1~10 の足し算のプログラムを while 文を使って書くと次のようになる。この例では i を減算して行くが、もちろん加算しても結構。

```
int i=10, sum=0;  
while (i>0) {  
    sum = sum + i;  
    i--;  
}
```

※ ここでも「条件が真の間だけ繰り返えす」。

while(true) と書いて無限ループとし、{ } ブロックの中で、後述の break 文を書いてループから抜けるパターンも良く使われる。この場合に C 言語では true 値として 1 を使い while(1) と書きましたが、Java では true と書かねばならないので注意すること。

do-while 文

do-while 文の構造

```
do{  
    文  
} while (条件);
```

do-while 文は while 文と似ているが、文が必ず 1 回は実行される場所が異なる。上記の構造の do-while 文は次のように実行される。

- (1) 文の実行
 - (2) 条件の判定→条件が否定されたら while 文から抜ける
- 以降 (1) ~ (2) の繰り返し。

足し算のプログラムを do-while 文で書くと次のようになる。

```
int i=10, sum=0;  
do{  
    sum = sum + i;  
    i--;  
} while (i>0);
```

switch 文

switch 文の構造

```
switch (評価式) {  
    case 定数:  
        文  
        break;  
    case 定数:  
        文  
        break;  
    ...  
    default:  
        文  
}
```

switch 文では評価式の値に基づいて、値が一致する定数を持つ case ラベルへジャンプする。その後、break 文または switch ブロックの最後になるまで処理を継続する。各 case 文の最後には break 文を書くのが普通だが、もし break 文がないと、次の case ラベル以降の内容も引き続いて実行する。式の評価値がどの case の定数とも一致しないときは、default ラベルが書かれているとその部分を実行する。

★ 評価式と定数は byte、char、short、int、String 型、または enum のいずれかでなくてはならない。以下に switch 文の例を示す。

```
switch (score) {  
    case 3:  
        System.out.print("たいへん");           // 印刷文(改行なし)  
    case 2:  
        System.out.println("良かったです"); // 印刷文(改行あり)  
        break;  
    case 1:  
        System.out.println("普通です");  
        break;  
    default:  
        System.out.println("評価できません");  
}
```

印刷文が初めて出てきた。この書き方を覚えること。単に print と書くと () の内容を印刷するだけだが、println とすると、印刷後改行する。ln は line の略。この他、印刷の書式が指定できる printf 文がある。これについては別途説明する。

上の例では score の値により、3→“たいへん良かったです”、2→“良かったです”、1→“普通です”、それ以外→“評価できません” が印刷される。

※ この例は break 文省略時の動作説明のために多少無理して作ったもので、このような作り方を推奨するものではない。普通は case3 の場合にもちゃんと全文を書いて break 文を入れた方が良い。break 文を書かないときは、その理由をコメントにしておくとも保守性が良い。

その他

Java には goto 文が無い。以下の文でプログラムの処理の流れを制御する。

break 文

break 文は for、while、do などのループから脱出するときに使用する。ループが多重になっている時は一つ外のループに抜ける。while 文にラベルをつけ、break 文でそのラベル名を指定すると多重ループの中にいても、ラベルをつけた while 文の外に出る。

continue 文

continue 文はやはりループの中で用いる。continue 文があるとループの終端までスキップする。for 文なら更新の実行から、while 文なら条件の判定から続行する。次の回以降のループ処理は普通に実行する。

return 文

return 文はメソッド（サブルーチン：次章で説明）から抜け出るときに使う。普通はメソッドの最後に記述するが、途中で記述すると、そこで処理を打ち切りメソッドから抜ける。return 文の直後に式を書くことができる。式があるとその値をメソッドの呼び出し元に返す。

メソッド (method)

Java のメソッド（ほかの言語でいう関数のこと。Java では関数とは言わない）は、メソッドが処理結果の値を 1 つだけしか返却できない。この値のことを戻り値と言う。Java ではメインプログラムもメソッドの形態をとるので、すべての処理はメソッドとして記述される。

メソッドの例

簡単なメソッドの例を示す。以下は 2 つの数を受けとって、その和を返すメソッド。先頭の `int` は戻り値の型。このメソッドの結果を `int` 型で返すことを示してる。次の `add` がこのメソッドの名前で、その次の丸括弧 `()` の中が引数。引数はデータ型と引数名とをペアにして書く。複数ある時はコンマで区切る。次の中括弧 `{` 以降に処理の中味を記述する。`return` はこのメソッドが返す値を指定する。最後の中括弧 `}` でメソッドを閉じる。

```
int add (int n1, int n2) {  
    int sum;  
    sum = n1 + n2;  
    return sum;  
}
```

メソッドの呼び出し方

Java には、CALL のように明確にメソッドを呼び出す書式はない。メソッドの名前を書くだけで、そのメソッドが実行される。戻り値を返す時は、以下のようにメソッド自身を変数 `c` に代入する形で使用される。メソッドはこのような形式で使われるので関数と呼ばれることもある。

```
c = add(a, b);
```

※ メソッドの定義では引数にデータ型と引数名とをペアで書くが、呼び出すときは単に引数名だけを書く。初めのうちは混同しやすいので、よく覚えること。

メソッドの構成

```
修飾子 戻り値の型 メソッド名(引数のリスト) {  
    メソッド本体  
}
```

メソッドの一般形は上のようなになる。それぞれの項目について、簡単に説明する。上の書式の1行目書式、すなわち修飾子から丸括弧の終わりまでを、そのメソッドのシグニチャーと呼ぶ。

修飾子

修飾子はメソッドに様々な属性を与えるもので、必要に応じて記述する。修飾子はクラスにかかわるものが多いので、後のクラスの章で詳しく説明する。

戻り値の型

メソッドが処理結果として持って帰る値（戻り値）の型を記述する。戻り値が無いときはvoid（ボイド）型としてその旨を表明する。voidは英語で空（くう）のとか無効のとかいった意味。メソッドの本体では、voidの時を除き、ここで記述した型を持つデータをreturn文で返さなければならない。

※ C言語では、当初は戻り値の型を省略するとint型を示すことになっていた。しかしそれでは本当に戻り値がない時との区別がつかず、改良してvoid型が作られた。Javaではこれらの経緯を踏まえて最初からvoid型が規定されています。

メソッド名

メソッド名には自由な名前をつけられる。但し識別子なので、使用する文字は1章で述べた命名規則に従う必要がある。

丸括弧（）

メソッド名の直後の丸括弧は引数がなくても、必ず書く。それによって変数かメソッドかの区別をしている。

引数のリスト

引数のリストは、データ型名と引数名とをペアで記述する。引数が複数あるときはこのペアをコンマで区切って書く。メソッドを呼び出すときは引数のデータ型と数、および順序が一致していなければならない。

データの渡し方

メソッドが呼ばれると引数の値が（コピーして）メソッドに渡さる。次のプログラムを参考に説明をする。
a と b の値を入れ換えている。

```
void swap(int a, int b) {  
    int work = a;  
    a = b;  
    b = work;  
}
```

しかしこのメソッドを `swap(x, y)` と呼んでも、x と y の値は入れ代わらない。メソッドを呼ぶ時には x と y の値をコピーして渡しているだけなので、元の x と y には何の影響も及ぼさない。

このように入力パラメーターとして、値そのものを渡すやり方を「値渡し」と言う。データの渡し方はこの他に、その格納アドレスを渡す「参照渡し」があります（参照とはアドレスのことです）。Java のメソッドはすべて「値渡し」になる。

参照型データの受渡し

先に配列のところでも述べたように、基本データ型以外の変数は参照型と言って、値ではなくその値の格納域のアドレスが入っている。参照型データをメソッドに渡すと、このアドレスが「値渡し」されるので、事実上「参照渡し」になる。

配列を使うと次のように内容を交換するプログラムを記述することができる。

```
void swap(int[] a) {    // a[0]とa[1]との値を交換する  
    int work = a[0];  
    a[0] = a[1];  
    a[1] = work;  
}
```

これを呼ぶプログラムは以下の通り。これにより x[0] は 20、x[1] は 10 になる。

```
int[] x = {10, 20};  
swap(x);
```

可変長引数

J2SE5.0 からメソッドの引数の数を変えられる可変長引数の機能が追加される。この機能はメソッドを宣言する時に以下のようにデータ型名の後ろに「...」と記述しておくことで、引数の数を可変にできる。

```
void vari(String... parm) {
```

呼び出す側では、以下のように引数の個数を自由に選べる。

```
vari("a");  
vari("xxx", "yyy", "zzz");
```

可変長引数として宣言された変数は、実際に配列になる。したがってメソッドの中で引数を使うときは次のように配列を使用する。

```
void vari(String... parm) {  
    for(int i = 0; i < parm.length; i++) {  
        System.out.println(parm[i]);  
    }  
}
```

以下、可変長引数を使用するときの注意点を示す。

- ・ 引数のデータ型は共通でなければなりません。
- ・ 他に引数があるとき、可変長引数は最後の引数にしか適用できません。
- ・ 呼び出す時の引数の個数が0個でもかまいません。
- ・ 名前が同じで引数の個数が固定のメソッドと可変のメソッドが混在した場合（メソッドがオーバーロード[後述]されている場合）、引数の個数が固定のメソッドが優先されます。

main メソッド

独立した Java のアプリケーションプログラムは main という名前のメソッドから実行を開始する。最初に実行されるメソッド。 main メソッドのシグニチャは次の通り。

```
public static void main(String[] args)
```

public

どのプログラムからもアクセスできるという意味。 GLOBAL の概念に似ています。 詳細はクラスの章で説明する。

static

通常のメソッドは（参照型なので）new しなければメモリ領域に展開されませんが、static とすると、new しなくても展開される。 main メソッドは最初に実行されるので、誰も new しなくても、static にしておく必要がある。 詳細はクラスの章で説明する。

void

戻り値が無いことを示す。 main メソッドでは値を返しても誰も受け取ってくれないので、必ず void にする。

main

これはメソッドの名前。 必ず main にする。

String[]

OS から Java アプリケーションを起動するときに指定したパラメーターを、引数として受け取る。 その引数が文字列の配列であることを示す。

args

引数の変数名です。 変数名ですから何と書いても良いが、このように書くのが慣例。 「アーギュス」「アレグ」と発音する。

コマンドライン

Windows などの OS から Java アプリケーションを起動するときの、一連の命令（コマンド）の書式をコマンドラインと言う。 コマンドラインの例を示す。

```
java Sample apple banana // ☆
```

上記で java は java を実行させるコマンド、Sample がプログラム名、apple と banana がパラメーター。 これらを受け取って、プログラムで使うときの変数が args。 以下に args の使用例を示す。

```
class Sample {  
    public static void main(String[] args) {  
        System.out.println("最初の引数は " + args[0] + " です");  
        System.out.println("2 番目の引数は " + args[1] + " です");  
    }  
}
```

★ このプログラムは上述のコマンド☆に対し次のように出力する。

```
最初の引数は apple です  
2 番目の引数は banana です
```

※ このプログラムはこのままでは問題がある。 それは実行時に ☆ のようにパラメーターを必ず入力するとは限らない。

ユーザーがパラメーター無しで単に「java Sample」のように入力してしまうと、args[0] は定義されないので未定義の配列をアクセスすることになり、例外が発生する。 これを避けるには args[0] を呼ぶ前に、args.length（args 配列の長さ）を調べて、想定値と異なるときはその旨を出力して、再入力を促す必要がある。 例えばパラメーターを 1 つだけ要求する場合は次のようにする。

```
if ( args.length < 1 ) {  
    System.out.println ("xx を入力してください。");  
    exit(1);  
}
```

※ C 言語では args[0] にはプログラム名が入っていて、args[1] に最初の引数が入っている。 しかし Java では最初の引数を args[0] で取得するので、C プログラマーは注意が必要。

変数のスコープと初期化

メソッドの中で一時的に使用する変数を宣言することができる。「データの渡し方」の項のプログラム例で使った work などがその例。これらはメソッド内部のみで有効であり、ローカル変数と言う。ローカル変数は他のメソッドで宣言された同名のローカル変数と衝突しない。

変数のスコープ

ローカル変数という考え方はメソッドに限らず、ブロック（中括弧 { } でくくられた範囲）について有効である。以前 for 文のところ（for 文内での変数の型宣言）で説明したが、for 文の有効範囲も中括弧 { } で囲まれているのでブロックする。このような変数の有効範囲を変数のスコープと言う。

初期化のデフォルト設定

メソッドの外で宣言された基本データ型の変数は、デフォルト値（基本データ型参照）で初期化されるが、メソッドの中で宣言された変数は値が入っていないので、初期化せずに使うとコンパイルエラーになる。

オーバーロード (Overload)

オーバーロードとは、同じクラスの中でメソッド名と戻り値の型が同じで、引数の型や数、並び順が違うメソッドを2つ以上定義することを言う。同じような処理だが、一部を変数にして処理結果をより自由に変えたい場合などに使う。

例を示す。

```
public class Main {

    public static void main(String args[]) {
        msg("佐藤");
        msg("山田", "はじめまして");
    }

    static void msg(String name) {
        System.out.println("こんにちは、" + name + "です");
    }

    static void msg(String name, String greeting) {
        System.out.println(greeting + "、" + name + "です");
    }

}
```

実行結果：

```
こんにちは、佐藤です
はじめまして、山田です
```

上記例では、同じメソッド名で引数の数が違う 2 つのメソッド msg を実行している。メソッドを呼び出して引数を指定する際に String 型の変数を 1 つ指定すれば、String 型の引数を 1 つ持つメソッドが呼び出されて実行される。また、String 型の変数を 2 つ指定すれば String 型の引数を 2 つ持つメソッドが呼び出されて実行される。

このように、同じメソッド名の中からどのメソッドを呼び出すかは、指定する引数の型や数、順番を元にコンパイラが判断してくれるようになっている。

オーバーロードができない場合

オーバーロードをするためには決まりごとがある。オーバーロードは、引数の型と個数と全く同じでは使えない。これはメソッド名と引数が全く同じだと、どちらのメソッドを使えばいいのか判断できなくなるため。そのため、オーバーロードをするときには必ず引数の「型」と「数」と「並び順」を変えて定義するようにする。注意点として、戻り値の方を変えれば異なるメソッドと判断されるのではないかと思うかもしれないが、引数が同じ場合には使用できない。

オーバーロードをするときには必ず引数の型と数と並び順を変えるようにすること。

引数が継承関係にあるときの優先順位

引数の数や順番が同じで、型だけが違う場合に注意しなければならないことがある。型に継承関係がある場合は、どのメソッドが呼び出されるか優先順位が決められている。

```
public class Main {

    public static void main(String args[]) {
        Integer i = new Integer(1);
        Number num = i; // アップキャスト
        msg(num);
    }

    static void msg(Integer i) {
        System.out.println("引数が Integer 型のメソッドを呼び出しました");
    }

    static void msg(Number n) {
        System.out.println("引数が Number 型のメソッドを呼び出しました");
    }

}
```


実行結果：

引数が Number 型のメソッドを呼び出しました

上記例では、Integer 型のオブジェクト `i` を Number 型のオブジェクト `num` に格納している。Integer クラスは Number クラスを継承したクラスなので、このように格納することが可能。
これをアップキャストと言う。
このような場合も実行結果のとおり、指定する引数の型にあわせてメソッドが呼び出される。
生成したインスタンスの型ではないので注意すること。

オーバーライド (Override)

オーバーライドは、継承した子クラスで親クラスと同じメソッド名のまま、処理内容を変更して再定義することと言う。オーバーライドするには、メソッド名を同じにする必要があるのに加えて、メソッドの引数は同じ数、同じ順番にする必要がある。

ちなみに、スーパークラスとは継承された親となるクラスのこと、サブクラスとは継承したクラスのこと。また「super」句を使うと、オーバーライドした場合でもスーパークラスで定義したメソッドを変更せずにそのまま使用することも可能。

オーバーライドは継承したサブクラスのメソッドで使用する。継承するメリットは、再利用できて、書く記述量を減らすことができるという点。変更が必要な場合も、変更する箇所も少なくできる。「再利用しつつも、変えたいところは自由に換えられる」というポリモーフィズム(多様性)の思想を実現している。

例を示す。

```
// スーパークラス
class ClassSuper {
    protected int i1 = 3;
    protected int i2 = 5;

    public void calc() {
        System.out.println("i1 + i2 = " + (i1 + i2));
    }
}

//サブクラス
class ClassSub extends ClassSuper{
    public void calc() {
        System.out.println("i1 * i2 = " + (i1 * i2)); // 処理を変更
    }
}

public class Main {

    public static void main(String[] args) {
        ClassSub cs = new ClassSub();
        cs.calc();
    }

}
```

実行結果：

```
i1 * i2 = 15
```

```
// スーパークラス
class ClassSuper {
    protected int i1;
    protected int i2;

    public ClassSuper(int i1, int i2) {
        this.i1 = i1;
        this.i2 = i2;
    }

    public void calc() {
        System.out.println("i1 + i2 = " + (i1 + i2));
    }
}

// サブクラス
class ClassSub extends ClassSuper {
    public ClassSub(int i1, int i2) {
        super(i1, i2);
    }

    @Override
    public void calc() {
        System.out.println("i1 * i2 = " + (i1 * i2)); // 処理を変更
        super.calc(); // スーパークラスと同じ処理
    }
}

public class Main {

    public static void main(String[] args) {
        ClassSub cs = new ClassSub(3, 5);
        cs.calc();
    }
}
```

実行結果:

i1 * i2 = 15

i1 + i2 = 8

<END>