

# Outline

- the *main* method in java applications
- the JAR (**J**ava **A**Rchive)Tool

# *main* method of java application

# Java application entry

- A class could enclose many methods and many attributes.
- But, there is one mandatory method for a valid java application entry
- Every java application should have an entry thread called **main** function.
- method signature: **void main(String[] args)**

# Structure of main method








```
public static void main(String[]  
    args){body of method}
```

- **Description :**
  - `public`: the method is globally accessible
  - `Static`: the method doesn't expect class instantiation to be called
  - `void`: the method does not return anything
  - `String[] args`: an array of strings parametre for the method
  - `body of method`: any valid java statements

# Command line argument for main

- Assume we want to write a java program that computes sum of two integers. Assume also that we need to pass the numbers to be summed up via arguments of the main method
- How could we do this?
- Solution: args[0] and args[1] can be the numbers to be summed. Then `int sum = args[0]+args[1]` is the summation result.
- Thus, `java 50 20` can be used while executing

# JAR tool

Name	Date modified	Type
 a.MF	3/7/2020 8:06 PM	MF File
 Course.class	3/19/2020 9:17 PM	CLASS File
 myJavaJar.jar	3/24/2020 9:40 AM	WinRAR archive
 OOP.class	3/19/2020 9:17 PM	CLASS File
 OOP.java	3/7/2020 9:57 PM	JAVA File
 OurFirstJavaProgram.class	3/21/2020 3:37 PM	CLASS File
 OurFirstJavaProgram.java	3/21/2020 3:29 PM	JAVA File



# Section Outline

- Basics of jar (java archive) file
- Create a .jar file for a java application in the command line
- Running a JAR-Packaged java application from the command line
- Extract (unjar) a .jar file of a java application
- Create a .jar file for an application using NetBeans IDE

# What is a jar (java archive) file?

- We usually zip our files, for example, before sending via email or to minimize disk storage consumption.
- Similarly, archiving files to one file is common in java for a number of motives.
  - We can archive any file including java source files (.java), java byte codes(.class), manifest files and images
  - Most of .jar files include compiled java codes (.class files)
- This is achieved by creating a.jar file



# What is a jar (java archive) file?...

- **.jar** refers a file extension that stands for java archive
- Java archive file is an archive of class files or java source codes or whatever content we need for ease handling of the java apps
- The jar file can, then, be executed in any device that has JRE installed

## Creating a .jar file for a java application:Jar Tool

- For example, you could develop a java application, create a jar file of only the java byte codes (.class files) appropriately and give to me so that I can run your application without having the java source code (.java)files
- In the JDK API, there is an executable called jar.exe.
- This executable file could, for example, be available at:

**C:\Program Files\Java\jdk1.8.0\_111\bin**  
(this is the **JAVA\_HOME\bin** indeed)

## Creating a .jar file for a java application...

- we can create a .jar file for our java application in the command line by issuing the command jar followed by some options, name of our jar file and elements to be included in the jar
  - **Jar -cf** jar-file input-file(s)
- This command will create a compressed JAR file with the specified file name and place it in the current directory.
- The command will also generate a **default manifest file** for the JAR archive.
  - /META-INF/MANIFEST.MF

## Creating a .jar file for a java application...

- When we open the default manifest file /META-INF/MANIFEST.MF, we get the following content:

```
Manifest-Version: 1.0
```

```
Created-By: 1.8.0_131 (Oracle  
Corporation)
```

- As customized manifest file needs to be added to a JAR file, we are supposed to add some features and include in our archive.
  - The entry class needs to be explicitly shown in the manifest file

# Creating a .jar file for a java application...

- If the entry class (that contain the main function) is *ClassEntry.class*, we need to include the following entry in the manifest file
  - `Main-Class: ClassEntry`
- To know how to use the jar command , we can type jar in the command line and then press enter, after which we will get the result shown in the next slide.

```
C:\Users\Tadele\Desktop\multi client service>jar
```

# How to create a .jar file for an application in the command line...

```
C:\Users\Tadele\Desktop\multi client service>jar
Usage: jar {ctxui}[vfmn0PMe] [jar-file] [manifest-file] [entry-point] [-C dir] files ...
Options:
  -c  create new archive
  -t  list table of contents for archive
  -x  extract named (or all) files from archive
  -u  update existing archive
  -v  generate verbose output on standard output
  -f  specify archive file name
  -m  include manifest information from specified manifest file
  -n  perform Pack200 normalization after creating a new archive
  -e  specify application entry point for stand-alone application
      bundled into an executable jar file
  -0  store only; use no ZIP compression
  -P  preserve leading '/' (absolute path) and ".." (parent directory) components from file names
  -M  do not create a manifest file for the entries
  -i  generate index information for the specified jar files
  -C  change to the specified directory and include the following file
```

If any file is a directory then it is processed recursively.

The manifest file name, the archive file name and the entry point name are specified in the same order as the 'm', 'f' and 'e' flags.

Example 1: to archive two class files into an archive called classes.jar:

```
jar cvf classes.jar Foo.class Bar.class
```

Example 2: use an existing manifest file 'mymanifest' and archive all the files in the foo/ directory into 'classes.jar':

```
jar cvfm classes.jar mymanifest -C foo/ .
```

# Creating a .jar file for an application...

- **Example:** Assume I have two java source codes inside the **multi client service** directory: `Client.java` and `MultiClientServiceServer.java`. Assume also that I want to include only the compiled versions (.class) of the above two .java files in the .jar file. Assume also that the name of the resulting jar file is `myJarFile.jar`. Create a .jar file of the java application based on this information.
- To answer this question, we are going to follow the following steps:
  - Change the working directory to **multi client service** directory
  - Compile the two .java files by issuing the command **javac** command  
**Javac Client.java**  
**Javac MultiClientServiceServer.java**
  - Create the .jar file using jar command by including the requested files  
**Jar -cf myJarFile.jar Client.class MultiClientServiceServer.class**
  - DONE!

# Archiving directory recursively

- I could have also followed the following procedure to get the required jar of the above question.
  - Create a folder to which compiled classes will be stored
    - **Example:** md `compiledClasses`
  - Then, compile the classes by specifying the created directory as output store of the compile.
  - I can compile both of the .java classes in one javac statement
    - **Javac** -d `compiledClasses` `Client.java` `Server_MultiClient.java`
  - Then I can ask the jar tool to make the jar file of the classes inside the folder
    - **Jar** -cf `myJarFile.jar` `compiledClasses` or
    - **Jar** -cf0 `myJarFile.jar` `compiledClasses` or
    - **Jar** -cfv0 `myJarFile.jar` `compiledClasses` or
    - **Jar** -cfv `myJarFile.jar` `compiledClasses`



## -C option

- By now, the .jar file with the desired jar file name must be created.
- You can type **dir** and see the .jar file has been added to the current directory
- Assume we have an image folder called **Images** inside our project that contains two images: **img1.jpg** and **img2.jpg**. We could use the following jar command to archive our project
  - **jar -fc myJarApp.jar compiledClasses Images**
- The above jar command add the images recursively inside a folder called Images
- What if we want to add the images recursively to our project directory (instead of putting it inside the Images folder)?
  - **Jar -fc myJarApp.jar compiledClasses -C Images .**

# Viewing the contents of .jar file

- Maybe, we are interested to view the contents that have been included in a .jar file.
- The basic format of the command for viewing the contents of a JAR file is:
  - Jar **tf** **jar-file**
- The **t** option indicates that you want to view the table of contents of the JAR file
- The **f** option indicates that the JAR file whose contents are to be viewed is specified on the command line
- The **jar-file** argument is the path and name of the JAR file whose contents you want to view
- This command will display the JAR file's table of contents to the console
- You can optionally add the verbose option, **v**, to produce additional information about the file sizes and last-modified dates in the output
- We can issue, for example, **jar tf myJarFile.jar** to see the table of contents of the jar file

# Running a JAR-Packaged Software

- Now that you have learned how to create JAR files, how do you actually run the code you packaged?
- Consider a scenario that your JAR file contains an application that is to be started from the command line.
- You can run JAR packaged applications with the java launcher (java command). The basic command is:
  - **Java** `-jar jar-file`
- Make sure that you have included a manifest file in your .jar file. Otherwise, you could hardly run your .jar file
- The manifest file contains the main-class/ entry point of the application (of the jar file)

# How to extract (“unjar”) a .jar file

- The basic command to use for extracting the contents of a JAR file is:
  - `jar xf jar-file [archived-file(s)]`
- The `x` option indicates that you want to extract files from the JAR archive
- The `f` option indicates that the JAR file from which files are to be extracted is specified on the command line
- The `jar-file` argument is the filename of the JAR file from which to extract files
- `[archived file(s)]` is an optional argument consisting of a space-separated list of the files to be extracted from the archive. If this argument is not present, the Jar tool will extract all the files in the archive
- As many files as desired can be extracted from the JAR file in the same way. When the command doesn't specify which files to extract, the Jar tool extracts all files in the archive

# How to create a .jar file for an application in NetBeans

- Click run → Build project
- Then the .jar file for the project is created in the {project directory}/dist folder

# Working with Manifest Files

- When you create a JAR file, it automatically receives a default manifest file.
- There can be only one manifest file in an archive, and it always has the pathname: **META-INF/MANIFEST.MF**
- When you create a JAR file, the default manifest file simply contains the following:
  - Manifest-Version: 1.0
  - Created-By: 1.7.0\_06 (Oracle Corporation)
- These lines show that a manifest's entries take the form of "header:value" pairs
- The name of a header is separated from its value by colon

# Working with Manifest Files...

- The manifest can also contain information about the other files that are packaged in the archive.
- Exactly what file information should be recorded in the manifest depends on how you intend to use the JAR file.
- The default manifest makes no assumptions about what information it should record about other files.

# Modifying a Manifest file

- You use the `m` command-line option to add custom information to the manifest during creation of a JAR file
- To modify the manifest, you must first prepare a text file containing the information you wish to add to the manifest. You then use the JAR Tool's `m` option to add the information in your file to the manifest
- The basic command has this format
  - `Jar cfm jar-file manifest-addition input-file(s)`
- The `m` and `f` options must be in the same order as the corresponding arguments



# An Example

- Assume I have the following java file (that has two classes). And, I want to create a .jar file of this application.

```
public class Class1 {  
    public static void main(String [] args){  
        System.out.println("This is a statement in Main  
class");  
        AdditionalClass obj= new AdditionalClass("Tadele");  
        System.out.println("The next statement uses  
another non main class");  
        System.out.println ("Your name is: " + obj.name);  
    }  
}  
class Class2{  
    public String name;  
    AdditionalClass(String name){  
        this.name =name;  
    }  
}
```

# An Example...

We could follow the following steps to create a runnable .jar file

- Create a text file that contain the manifest file's contain
  - For example, create a file a .txt that contain "Main-Class: Class1"
  - **The text file must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return**
- Write the .java file and compile the java file
  - So, we will have Class1.class and Class2.class
- Change your directory to the directory where your project is located
  - My project is located at
- Type the Jar Tool command `C:\Users\Tadele\Desktop\Jaring>`
  - `jar cfm myOutputJar.jar a.txt *`  
(myOutputJar.jar is the expected jar file; cfm are the options as we discussed in the previous slides; a.txt is a text file we created above- this file's content will be used in creating the manifest file for the .jar file)

# An Example:

- The above command creates the JAR file with a manifest with the following contents:

Manifest-Version: 1.0

Created-By: 1.8.0\_111 (Oracle Corporation)

Main-Class: Class1

- When you run the JAR file with the following command, the main method of Class1 executes:

Java -jar myOutputJar.jar

# End of Chapter 2