





OOP with Java 26. Libraries and Executables

Thomas Weise · 汤卫思

 $tweise@hfuu.edu.cn \ \cdot \ http://iao.hfuu.edu.cn$

Hefei University, South Campus 2 Faculty of Computer Science and Technology Institute of Applied Optimization 230601 Shushan District, Hefei, Anhui, China Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区 计算机科学与技术系 应用优化研究所 中国 安徽省 合肥市 蜀山区 230601 经济技术开发区 锦绣大道99号









- 4 Using a Library
 - Executable JARs







• Java programs usually consist of lots of .java or .class files



- Java programs usually consist of lots of .java or .class files
- Also, there often are many resources such as text files and images



- Java programs usually consist of lots of .java or .class files
- Also, there often are many resources such as text files and images
- We can hardly ship a heap of 1000 files as application to a user



- Java programs usually consist of lots of .java or .class files
- Also, there often are many resources such as text files and images
- We can hardly ship a heap of 1000 files as application to a user
- For this purpose, jar files exist



- Java programs usually consist of lots of .java or .class files
- Also, there often are many resources such as text files and images
- We can hardly ship a heap of 1000 files as application to a user
- For this purpose, jar files exist
- A jar file is basically a special zip archive which contains all the files of a program or library



- Java programs usually consist of lots of .java or .class files
- Also, there often are many resources such as text files and images
- We can hardly ship a heap of 1000 files as application to a user
- For this purpose, jar files exist
- A jar file is basically a special zip archive which contains all the files of a program or library
- jar files can either be executable, i.e., be programs, or not, in which case they are libraries



- Java programs usually consist of lots of .java or .class files
- Also, there often are many resources such as text files and images
- We can hardly ship a heap of 1000 files as application to a user
- For this purpose, jar files exist
- A jar file is basically a special zip archive which contains all the files of a program or library
- jar files can either be executable, i.e., be programs, or not, in which case they are libraries
- That's already the most important stuff: Actually, you can create a x.zip archive with the contents of your /bin folder of an Eclipse project, rename it to x.jar and you got yourself a library



- Java programs usually consist of lots of .java or .class files
- Also, there often are many resources such as text files and images
- We can hardly ship a heap of 1000 files as application to a user
- For this purpose, jar files exist
- A jar file is basically a special zip archive which contains all the files of a program or library
- jar files can either be executable, i.e., be programs, or not, in which case they are libraries
- That's already the most important stuff: Actually, you can create a x.zip archive with the contents of your /bin folder of an Eclipse project, rename it to x.jar and you got yourself a library
- But let's look at this step-by-step



• We have learned a two ways to structure code



- We have learned a two ways to structure code, including
 - dividing code into different classes



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.
- Now we can just



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.
- Now we can just
 - put this code into one project



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.
- Now we can just
 - put this code into one project
 - develop each program in a separate dedicated project



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.
- Now we can just
 - put this code into one project
 - develop each program in a separate dedicated project
 - copy the "shared" code into each project



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.
- Now we can just
 - put this code into one project
 - develop each program in a separate dedicated project
 - copy the "shared" code into each project
- This is not a nice solution



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.
- Now we can just
 - put this code into one project
 - develop each program in a separate dedicated project
 - copy the "shared" code into each project
- This is not a nice solution, since very time our shared code changes,
 - we always have to copy many files



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.
- Now we can just
 - put this code into one project
 - develop each program in a separate dedicated project
 - copy the "shared" code into each project
- This is not a nice solution, since very time our shared code changes,
 - we always have to copy many files
 - if a file becomes no longer needed, delete



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.
- Now we can just
 - put this code into one project
 - develop each program in a separate dedicated project
 - copy the "shared" code into each project
- This is not a nice solution, since very time our shared code changes,
 - we always have to copy many files
 - if a file becomes no longer needed, delete
 - it's a maintenance nightmare



- We have learned a two ways to structure code, including
 - dividing code into different classes
 - dividing code focusing on different concerns into packages in Lesson 17: *Packages and import*
- Sometimes, we have code which is used by different programs
- Say, code to deal with reading and writing special files, code for rendering graphics, code implementing machine learning and mathematical routines, etc.
- Now we can just
 - put this code into one project
 - develop each program in a separate dedicated project
 - copy the "shared" code into each project
- This is not a nice solution, since very time our shared code changes,
 - we always have to copy many files
 - if a file becomes no longer needed, delete
 - it's a maintenance nightmare
- That's what libraries are good for: They are basically zip archives containing all classes of a project

Creating a Library



• Let's revisit our elaborate, package-based Person example from lesson Lesson 17: *Packages and import*

Creating a Library



- Let's revisit our elaborate, package-based Person example from lesson Lesson 17: *Packages and import*
- Here I print the classes again to refresh your memory



Listing: A Person class in package cn.edu.hfuu.iao.person

```
package cn.edu.hfuu.iao.person; // declare the package
   cn.edu.hfuu.iao.person
/** A class representing a person with constructor and toString
   method. */
public class Person {
 /** the family name of the person */
 String familyName;
 /** the given name of the person */
 String givenName;
 /** create a person record and set its name */
  public Person(String _familyName, String _givenName) {
    this.familyName = _familyName;
    this.givenName = _givenName;
 3
 /** return a string representation of this person record */
 public String toString() {
   return this.givenName + ''' + this.familyName;
 }
```



Listing: A Professor class in package cn.edu.hfuu.iao.person

```
package cn.edu.hfuu.iao.person; // declare the package cn.edu.hfuu.iao.person
/** A class representing a professor */
public class Professor extends Person { // class Processor extends class Person
    /** create a person record and set its name */
    public Professor(String _familyName, String _givenName) {
        super(_familyName, _givenName);
    }
    /** return "Prof. " + result of super.toString() = Person.toString() */
    @Override // mark this method explicitly as overridden
    public String toString() {
        return "Prof.u" + super.toString(); //$NON-NLS-1$
    }
}
```



Listing: A Student class in package cn.edu.hfuu.iao.person

```
package cn.edu.hfuu.iao.person; // declare the package cn.edu.hfuu.iao.person
/** A class representing a student */
public class Student extends Person { // class Student extends class Person
 /** the id of the student */
  String id;
  /** create a student record and set its name and student id */
  public Student(String _familyName, String _givenName, String _id) {
    super(_familyName, _givenName);
   this.id = _id;
  }
 /** return a string representation of this student record */
  COverride // mark this method explicitly as overridden
  public String toString() {
    return "student<sub>11</sub>" + super.toString(); //$NON-NLS-1$
```



Listing: A Foreign Exchange Student class in package cn.edu.hfuu.iao.person

```
package cn.edu.hfuu.iao.person; // declare the package cn.edu.hfuu.iao.person
/** A class representing a foreign exchange student */
public class ForeignExchangeStudent extends Student {
 /** the home country of the student */
  String homeCountry; // we add a new field
  /** create a student record and set its name. student id. and home country */
  public ForeignExchangeStudent(String _familyName, String _givenName,
                                String _id,
                                                    String country) {
    super(_familyName, _givenName, _id);
    this.homeCountry = country;
  3
  /** override toString() from Person */
  QOverride // mark this method explicitly as overridden
  public String toString() {
    return super.toString() + ", from, + this.homeCountry; //$NON-NLS-1$
  }
}
```



Listing: Person Reader in package cn.edu.hfuu.iao.io

```
package cn.edu.hfuu.iao.io:
/** a class to read a person record from stdin: using canonical class names*/
public class PersonReader {
 /** the constructor */
 public PersonReader(){
 /** Read a person record from stdin. All class names are specified canonically
   * Greturn the new person record */
  public cn.edu.hfuu.iao.person.Person read(java.util.Scanner scanner) {
    System.err.println("Enter_person's_family_name:"); //$NON-NLS-1$
    String familyName = scanner.nextLine(); // read a string from scanner
    System.err.println("Enter_person's_given_name:"); //$NON-NLS-1$
    String givenName = scanner.nextLine(); // read a string from scanner
    return new cn.edu.hfuu.iao.person.Person(familyName, givenName);
```



Listing: Professor Reader in package cn.edu.hfuu.iao.io

```
package cn.edu.hfuu.iao.io;
```

}

OOP with Jav

```
import java.util.Scanner; // import class Scanner from java.util
```

```
//import class Professor from package cn.edu.hfuu.iao.person
import cn.edu.hfuu.iao.person.Professor;
```

```
/** a class to read a professor record from stdin*/
public class ProfessorReader extends PersonReader {
```

```
/** the constructor */
public ProfessorReader(){
}
/** read a profesor record from scanner (pointing to stdin)
* @return the new person record */
@Override
public Professor read(Scanner scanner) {
   System.err.println("Enteruprofessor'sufamilyuname:"); //$NON-NLS-1$
   String familyName = scanner.nextLine(); // read a string from scanner
   System.err.println("Enteruprofessor'sugivenuname:"); //$NON-NLS-1$
   String givenName = scanner.nextLine(); // read a string from scanner
   return new Professor(familyName, givenName);
}
```



Listing: Student Reader in package cn.edu.hfuu.iao.io

```
package cn.edu.hfuu.iao.io;
```

```
import java.util.Scanner; // import class Scanner from java.util
//import class Student from package cn.edu.hfuu.iao.person
import cn.edu.hfuu.iao.person.Student;
```

```
/** a class to read a student record from stdin*/
public class StudentReader extends PersonReader {
```

```
/** the constructor */
public StudentReader(){
```

```
/** read a student record from scanner (pointing to stdin)
 * @return the new person record */
```

```
@Override
```

```
public Student read(Scanner scanner) {
   System.err.println("Enter_student's_family_name:"); //$NON-NLS-1$
   String familyName = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_student's_given_name:"); //$NON-NLS-1$
   String givenName = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_student's_lD:"); //$NON-NLS-1$
   String id = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_student's_lD:"); //$NON-NLS-1$
```

```
return new Student(familyName, givenName, id);
```

```
}
```



Listing: Foreign Exchange Student Reader in package cn.edu.hfuu.iao.io

```
package cn.edu.hfuu.iao.io;
```

```
import java.util.Scanner; // import class Scanner from java.util
//import class ForeignExchangeStudent from package cn.edu.hfuu.iao.person
import cn.edu.hfuu.iao.person.ForeignExchangeStudent;
```

```
/** a class to read a student record from stdin*/
public class ForeignExchangeStudentReader extends PersonReader {
```

```
/** the constructor */
public ForeignExchangeStudentReader(){
}
```

```
/** read a foreign exchange student record from scanner (pointing to stdin) \ast @return the new person record \ast/
```

@Override

```
public ForeignExchangeStudent read(Scanner scanner) {
   System.err.println("Enter_uexchange_ustudent'su_family_name:"); //$NON-NLS-1$
   String familyName = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_uexchange_student'su_given_name:"); //$NON-NLS-1$
   String givenName = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_uexchange_student'su_Div="); //$NON-NLS-1$
   String id = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_uexchange_student'su_Div="); //$NON-NLS-1$
   String id = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_uexchange_student'su_Div="); //$NON-NLS-1$
   String country = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_uexchange_student'su_Div="); //$NON-NLS-1$
   String country = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_uexchange_student'su_Div="); //$NON-NLS-1$
   String country = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_uexchange_student'su_Div="); //$NON-NLS-1$
   String country = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_uexchange_student'su_Div="); //$NON-NLS-1$
   String country = scanner.nextLine(); // read a string from scanner
   System.err.println("Enter_uexchange_student'su_Div="); //$NON-NLS-1$
   String country = scanner.nextLine(); // read a string from scanner
```

```
return new ForeignExchangeStudent(familyName, givenName, id, country);
```

```
ι
```

Creating a Library



- Let's revisit our elaborate, package-based Person example from lesson Lesson 17: *Packages and import*
- Here I print the classes again to refresh your memory

Creating a Library



- Let's revisit our elaborate, package-based Person example from lesson Lesson 17: *Packages and import*
- Here I print the classes again to refresh your memory
- Ok, memory refreshed. We have eight classes in two packages.

Creating a Library



- Let's revisit our elaborate, package-based Person example from lesson Lesson 17: *Packages and import*
- Here I print the classes again to refresh your memory
- Ok, memory refreshed. We have eight classes in two packages.
- Let's say that these are classes needed by several applications in our enterprise, including the software of the human resources department and the financial department

Creating a Library



- Let's revisit our elaborate, package-based Person example from lesson Lesson 17: *Packages and import*
- Here I print the classes again to refresh your memory
- Ok, memory refreshed. We have eight classes in two packages.
- Let's say that these are classes needed by several applications in our enterprise, including the software of the human resources department and the financial department
- Thus, we want to put them into a library which can be shared among these applications



• Let's make a new Eclipse project called person_library and put all the code in there



	Quick Access 🗄 😰 🐯
🛿 Package Explorer 🛛 🍃 Type Hierarchy 🔤 E	1 - 1
E 😒 😜	7
>javaExamples [javaExamples master]	
🖷 > person_library [javaExamples master]	
▼ඌ>src	
▼⊕ > cn.edu.hfuu.iao	
▼ ∰ > io	
ForeignExchangeStudentReader.java	
 PersonReader.java ProfessorReader.java 	
 ProressorReader.java 3 StudentReader.java 	
★ B > person	
Person <p< td=""><td></td></p<>	
Person.java	
Professor.java	
E Student.java	
🕨 📷 JRE System Library [java-8-openjdk-amd64]	
🔀 .classpath	
gitignore	
📝 .project	😰 Prob 🐵 Java 😣 Decl 🛷 Sear 📮 Cons 🛛 🖷 Prog 🎄 Deb 👘 🗖
	- X X R B B B B
	<terminated>IterableTest [Java Application] /usr/lib/jvm/java-8-openjdk-an Hello World! It's me your teacher.</terminated>
erson_library	
OOP with Java	Thomas Weise 15/26



- Let's make a new Eclipse project called person_library and put all the code in there
- We then choose Export from the File menu



New > Open File Open Projects from File System	シ ∞ ☆ ▾ ◑ ▾ ∿ ▾ ⊯ ♂ ▾ ⊵ ⊝ ⋪	
Close Close All		Quick Access
III Save IIII Save As IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII	▼ ■	J
Move Ø Rename Refresh Convert Line Delimiters To >		
a Print		
Switch Workspace > Restart		
🛍 Import 🗽 Export		
1 make_linux.sh [javaExamples/]		
2 make_linux.sh [javaExamples/lessons] 3 README.md [javaExamples/lessons]	😰 Prob @ Java 🗟 Decl 🛷 Sear 📮 Co	ns 🛙 🦏 Prog 🎋 Deb 🛛 🗖
4 README.md [javaExamples/lessons] Exit	<pre></pre>	a 🛃 😕 💭 💭 🚽 🖸 マ 📑 ♥ n] /usr/lib/jvm/java-8-openjdk-am
erson_library		
OOP with Java	Thomas Weise	15/26



- Let's make a new Eclipse project called person_library and put all the code in there
- We then choose Export from the File menu
- In the export wizzard, we choose Java and then JAR file and press Next



		×
JAR file		



- Let's make a new Eclipse project called person_library and put all the code in there
- We then choose Export from the File menu
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button



	pecificatio ort destinal resources t	tion will be	elative	to your works	pace.	_
	rson library					
· 🕑 🕮 si	rc					
	cn.edu.hfuu cn.edu.hfuu		1			
-						
Expor	rt generated rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi	ce files and gs for chec	checked resourc	d projects	factorings	
Expor	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi	folders for ce files and ags for chec tination:	checkeo resourc ked proj	d projects es		 Browse
Expor	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi	folders for ce files and ags for chec tination:	checkeo resourc ked proj	d projects es jects.Select re		• Browse
Expor	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi	ce files and ogs for chec tination: d_executab	checked resourc ked proj les/pers	d projects es jects.Select re son_user/libs/		• Browse
Expor	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi braries_an	folders for ce files and ags for chec tination: d_executat	checked resourc ked proj les/pers	d projects es jects.Select re son_user/libs/		• Browse
Expor Expor Expor Select the JAR file: Options: Comp Add d	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi braries_an	folders for ce files and ags for chec tination: d_executat ntents of th tries	checked resourc ked proj les/pers e JAR fil	d projects es jects.Select re son_user/libs/ le		▼ Browse
Expor Expor Expor Select the JAR file: Options: Comp Add d	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi braries_an press the con lirectory en	folders for ce files and ags for chec tination: d_executat ntents of th tries	checked resourc ked proj les/pers e JAR fil	d projects es jects.Select re son_user/libs/ le		Browse

OOP with Java

Thomas Weise



- Let's make a new Eclipse project called person_library and put all the code in there
- We then choose Export from the File menu
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it person.jar and hit OK



80							
Name: person.jar							
ග Home	• java javaExamples	lessons	26_libraries_and_executab	les person_user	libs	•	C7
🖿 Desktop	Name				*	Size	Modified
p rogram	person.jar					6.4 kB	15:21
в орро							
Seafile							
+ Other Locatio							
							.jar;.zip 🔻
						Cancel	OK



- Let's make a new Eclipse project called person_library and put all the code in there
- We then choose Export from the File menu
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it person.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next (well, we could as well click Finish now...)



	pecificatio ort destinal resources t	tion will be	elative	to your works	pace.	_
	rson library					
· 🕑 🕮 si	rc					
	cn.edu.hfuu cn.edu.hfuu		1			
-						
Expor	rt generated rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi	ce files and gs for chec	checked resourc	d projects	factorings	
Expor	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi	folders for ce files and ags for chec tination:	checkeo resourc ked proj	d projects es		 Browse
Expor	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi	folders for ce files and ags for chec tination:	checkeo resourc ked proj	d projects es jects.Select re		• Browse
Expor	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi	ce files and ogs for chec tination: d_executab	checked resourc ked proj les/pers	d projects es jects.Select re son_user/libs/		• Browse
Expor	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi braries_an	folders for ce files and ags for chec tination: d_executat	checked resourc ked proj les/pers	d projects es jects.Select re son_user/libs/		• Browse
Expor Expor Expor Select the JAR file: Options: Comp Add d	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi braries_an	folders for ce files and ags for chec tination: d_executat ntents of th tries	checked resourc ked proj les/pers e JAR fil	d projects es jects.Select re son_user/libs/ le		▼ Browse
Expor Expor Expor Select the JAR file: Options: Comp Add d	rt all o <u>u</u> tput rt Java <u>s</u> our rt refactorir export desi braries_an press the con lirectory en	folders for ce files and ags for chec tination: d_executat ntents of th tries	checked resourc ked proj les/pers e JAR fil	d projects es jects.Select re son_user/libs/ le		Browse

OOP with Java

Thomas Weise



- Let's make a new Eclipse project called person_library and put all the code in there
- We then choose Export from the File menu
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it person.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next (well, we could as well click Finish now...)
- We click Next



Define the options for the JAR export. Select options for handling problems: Export class files with compile errors Create source folder structure Build projects if not built automatically Saye the description of this JAR in the workspace Description file: Brows	-
Export class files with compile errors Export class files with compile warnings Create source folder structure Build projects if not built automatically Saye the description of this JAR in the workspace	
Export class files with compile warnings Greate source folder structure Build projects if not built automatically Saye the description of this JAR in the workspace	
 Create source folder structure ✓ Byild projects if not built automatically Saye the description of this JAR in the workspace 	
Build projects if not built automatically	
Save the description of this JAR in the workspace	
2escription file: Brow:	
	se
(?) <back next=""> Cancel Fini</back>	

OOP with Java

Thomas Weise

15/26



- Let's make a new Eclipse project called person_library and put all the code in there
- We then choose Export from the File menu
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it person.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next (well, we could as well click Finish now...)
- We click Next
- We click Finish and the jar archive will be created



AR Manifest Specification	-
Customize the manifest file for the JAR file.	
Specify the manifest:	
O Generate the manifest file	
Save the manifest in the workspace	
Use the saved manifest in the generated JAR description file	
Manifest file:	Browse
○ Use existing manifest from workspace	
Manifest file:	Browse
Seal the JAR JAR Sealed	Detajls
Sectore 2.11	Details
Seal some packages	
○ Seal some <u>p</u> ackages	
 Seal some packages elect the class of the application entry point: 	
elect the class of the application entry point:	Browse
-	Browse
elect the class of the application entry point:	Browse
elect the class of the application entry point:	Browse
elect the class of the application entry point:	Browse

OOP with Java

Thomas Weise



- Let's make a new Eclipse project called person_library and put all the code in there
- We then choose Export from the File menu
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it person.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next (well, we could as well click Finish now...)
- We click Next
- We click Finish and the jar archive will be created
- and we are done...



• Let us now confirm that the generated file person.jar is actually a "special" zip archive



😣 🖨 🕕 libs					
< > < javaExamp	les lessons 26_libraries_and_executable	s person_user	libs >		ୟ ≣ ଞ
⊘ Recent	Name		Size	Туре	Modified
✿ Home	person jar		6.4 kB	Archive	15:21
🛅 Desktop					
Trash					
₽ Network					
Computer					
в орро					
🖿 Seafile					
Connect to Server					
			"	oerson.jar" se	lected (6.4 kB)



- Let us now confirm that the generated file person.jar is actually a "special" zip archive
- Under Ubuntu Linux, we therefore can open it with the *Archive Manager*



3 Recent	Name				Size	Туре	Modi	fied
	pe	rson.jar 🛃 Open With OpenJDK Java 8 Run	time		6.4 kB	Archive	15:21	Ĩ
Trash Network		Open With Cut	>	Ar	chive Mana chive Moun	ter		
Computer		Copy Move To Copy To Make Link Rename			her Applica	don		
Connect to Server		Move to Trash Email Extract Here Properties						



- Let us now confirm that the generated file person.jar is actually a "special" zip archive
- Under Ubuntu Linux, we therefore can open it with the *Archive Manager*
- Tada, it opens as archive, with some special folder (for the Manifest, let's ignore this)



- Let us now confirm that the generated file person.jar is actually a "special" zip archive
- Under Ubuntu Linux, we therefore can open it with the *Archive Manager*
- Tada, it opens as archive, with some special folder (for the Manifest, let's ignore this)
- But the folder structure in the archive perfectly reflects our package hierarchy



🛞 🗇 💷 person.jar		
Extract +	٩	=
< > Decation: 1		
Name	Size	T
r cn	7.9 kB	Fo
META-INF	39 bytes	F
classpath	226 bytes	U
.gitignore	6 bytes	U
.project	373 bytes	U



🛞 🖨 💷 person.jar		
Extract +	۹	Ξ
C> Decation: Cn/		
Name	Size	T
n edu N	7.9 kB	Fo
₽	1.5 1.5	- 51



🔞 🖻 💿 person.jar			
Extract +		۹	Ξ
< > 🙆 Location: 🛅 /cn/edu/			
Name	*	Size	т
իքսս 💦		7.9 kB	Fo
S.			



😣 🖻 💿 person.jar			
Extract +		۹	Ξ
<> Decation: // /cn/edu/hfuu/			
Name	*	Size	Ţ
iao N		7.9 kB	Fo



🛞 🖨 💿 person.jar		
Extract +	۹	Ξ
Location: (cn/edu/hfuu/iao/		
Name	Size	Ţ
io	4.5 kB	F
person 💦	3.3 kB	Fo



- Let us now confirm that the generated file person.jar is actually a "special" zip archive
- Under Ubuntu Linux, we therefore can open it with the *Archive Manager*
- Tada, it opens as archive, with some special folder (for the Manifest, let's ignore this)
- But the folder structure in the archive perfectly reflects our package hierarchy
- And the folder cn/edu/hfuu/iao/person includes the compiled class files for the Person -related classes



⊗⊜ © person.jar Extract +	٩	Ξ
Location: (n/edu/hfuu/iao/person/		
Name 🔺	Size	T
ForeignExchangeStudent.class	996 bytes	Ja
Person.class	866 bytes	Ja
Professor.class	688 bytes	Ja
Student.class	792 bytes	Ja

2



- Let us now confirm that the generated file person.jar is actually a "special" zip archive
- Under Ubuntu Linux, we therefore can open it with the *Archive Manager*
- Tada, it opens as archive, with some special folder (for the Manifest, let's ignore this)
- But the folder structure in the archive perfectly reflects our package hierarchy
- And the folder cn/edu/hfuu/iao/person includes the compiled class files for the Person -related classes
- Cool, so now we have our library and it actually is a handy archive of all the necessary stuff in one single file



• We now want to use our library in our code

Using a Library



- We now want to use our library in our code
- Our main application shall be the same as in Lesson 17: *Packages* and *import*



18/26

Listing: A Main class using our Person classes

OOP with Java

```
import java.util.Scanner; // import class Scanner from the java.util package
import cn.edu.hfuu.iao.io.ForeignExchangeStudentReader; // import all needed data structure
import cn.edu.hfuu.iao.io.PersonReader:
import cn.edu.hfuu.iao.io.ProfessorReader;
import cn.edu.hfuu.iao.io.StudentReader:
import cn.edu.hfuu.iao.person.Person;
public class Main {
   * @param args we ignore this parameter */
  public static void main(String[] args) {
    PersonReader reader:
    Scanner scanner = new Scanner(System.in); // create a structured text reader
   System.err.println("Doilyouliwantitoireadii(p)rofessors, (s)tudents, (ori(e)changeistudents: ); //$NON-NLS-1$
   switch (scanner.nextLine().charAt(0)) { // check the first character entered by the user
      case 'p': { reader = new ProfessorReader(); break; } // p -> read professors
     case 's': { reader = new StudentReader(); break; } // s -> read students
     default: { reader = new ForeignExchangeStudentReader(); break; } // otherwise: read exchange students
   3
    for (;;) { // loop forever, see loop condition at bottom of loop
      Person person = reader.read(scanner): // use the verson read to read a verson
      System.out.println("Younentered: + person); // print person.toString //$NON-NLS-1$
      System.out.println("Type_enter_to_continue,_Ctrl-D_to_exit."); //$NON-NLS-1$
     if (scanner.hasNextLine()) { // if user pressed enter
        scanner.nextLine();
        continue:
                                  // and do another iteration
     3
                                   // no next line and we exit the main routine
      return:
```

Thomas Weise



- We now want to use our library in our code
- Our main application shall be the same as in Lesson 17: *Packages* and *import*



- We now want to use our library in our code
- Our main application shall be the same as in Lesson 17: *Packages* and *import*
- We create a new Eclipse project person_user and copy our Main class there



- We now want to use our library in our code
- Our main application shall be the same as in Lesson 17: *Packages* and *import*
- We create a new Eclipse project person_user and copy our Main class there
- Sadly, it won't compile, because now it misses all the required Person -related code



∲ • ∲ • \$ \$ \$ • \$ •	Quick Access 🛛 😰 🐯
😫 Packag 🛛 🔭 Type Hi 🗖 🗖	🔐 Main.java 🛙 🗖
 ► The second sec	<pre>1=import java.util.Scanner; // import class Scanner from the java 2 3 import cn.edu.hfuu.iao.io.ForeignExchangeStudentReader; // import 4 import cn.edu.hfuu.iao.io.PersonReader; // and 5 import cn.edu.hfuu.iao.io.StudentReader; 6 import cn.edu.hfuu.iao.io.StudentReader; 7 import cn.edu.hfuu.iao.person.Person; 9 /** testing our package structure */ 10 public class Main { 11 / /** The main routine reading person records of a certain type 12 * @param args we ignore this parameter */ 13 public static void main(String[] args) { 14 PersonReader reader; 15 Scanner scanner = new Scanner(System.in); // create a struct 16 17 System.err.println("Do you want to read (p)rofessors, (s)tu</pre>
	<pre>18 19 19 switch (scanner.nextLine().charAt(0)) { // check the first = 20 case 'p': { reader = new ProfessorReader(); break; } // p 21 case 's': { reader = new StudentReader(); break; } // s 22 23 24 25 for (;;) { // loop forever, see loop condition at bottom of 25 for (;;) { // loop forever, see loop condition at bottom of 26 Person person = reader.read(scanner); // use the person p 27 28 29 29 20 20 20 20 20 20 20 20 20 20 20 20 20</pre>

javaExamples



- We now want to use our library in our code
- Our main application shall be the same as in Lesson 17: *Packages* and *import*
- We create a new Eclipse project person_user and copy our Main class there
- Sadly, it won't compile, because now it misses all the required Person -related code
- Let's fix this.



- We now want to use our library in our code
- Our main application shall be the same as in Lesson 17: *Packages* and *import*
- We create a new Eclipse project person_user and copy our Main class there
- Sadly, it won't compile, because now it misses all the required Person -related code
- Let's fix this.
- First, we create a new folder libs in our project to store all the libraries needed



9 - 9 - 6 - 19	• • • • • • • • • • • • • • • • • • •			Quick Access
> G> javaExamples [j > G> person_library SB> person_user [i ▼GP > src	B → ↓ 1⇒ impor 2 avaExamples m n 3 impor javaExamples n n 4 impor	t java.util.Scar t <u>cn</u> .edu.hfuu.ia t <u>cn</u> .edu.hfuu.ia	ao.io.ForeignExchange ao.io.PersonReader; ao.io.ProfessorReader 📽 Java Project	E
 ★ (default pa ★ Main.java ★ JRE System Lil ℜ .classpath ℜ .classpath ℜ .gitignore ℜ .project 	Open in New Window Open Type Hierarchy Show In Copy Copy Qualified Name Paste Colete	F4 Shift+Alt+W ≯ Ctrl+C Ctrl+V	Project Package Class Clas	of a certain type // create a struce p)rofessors, (s)tu
	 Remove from Context Build Path Source Refactor 	> Shift+Alt+S > Shift+Alt+T >	Folder File Untitled Text File JUnit Test Case Task	<pre>/ check the first = r(); break; } // p); break; } // s eStudentReader();</pre>
person_user	 Import Export Refresh Close Project 	FS F S	📑 E <u>x</u> ample 📑 Other Ci	ition at bottom of / use the person r ruentrian // print ne

OOP wit



older	
Create a new folder resource.	
inter or select the parent folder:	
person_user	
ð 🗢 🗘	
> javaExamples [javaExamples master] > person_library [javaExamples master]	
🍣 > person_user [javaExamples master]	
in is src	
older name: libs	
older name: libs Advanced >>	

19/26



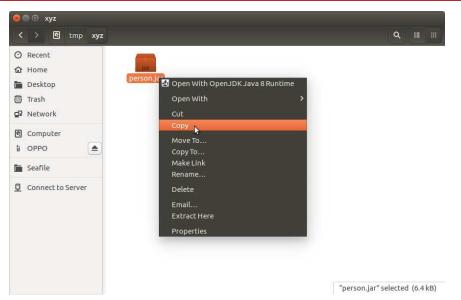
§ • ġ • ♥ Φ • Φ •	Quick Access 🔡 🐯 💱 🕈
🛿 Packag 🛛 🔭 🗖	妃 Main.java 🛙 🗖 🗖
 E S javaExamples [javaExamples ma person_library [javaExamples ma person_user [javaExamples ma >person_user [javaExamples ma > person_user [javaExam > person_user [javaExamples ma 	

libs - person_user



- We now want to use our library in our code
- Our main application shall be the same as in Lesson 17: *Packages* and *import*
- We create a new Eclipse project person_user and copy our Main class there
- Sadly, it won't compile, because now it misses all the required Person -related code
- Let's fix this.
- First, we create a new folder **libs** in our project to store all the libraries needed
- Then we copy our library person.jar into this folder





OOP with Java

Thomas Weise

19/26



원 - 상 - 박	Q • Q •	Quick Access 🗄 🐯 💱 🕯
🖥 Packag 🖾	💲 Type Hi 🗖 🗖	🖉 Main.java 🛛 🗖 🗖
> person_lib	E 🛞 👂 マ oles (javaExamples m prary (javaExamples n	<pre>1=import java.util.Scanner; // import class Scanner from the java 2 3 import <u>cn</u>.edu.hfuu.iao.io.ForeignExchangeStudentReader; // impc 4 import <u>cn</u>.edu.hfuu.iao.io.PersonReader; // and = 4</pre>
> person_us > # > src > # > (default > @ Main.jav		<pre>a 5 import cn.edu.hfuu.iao.io.ProfessorReader; a 6 import cn.edu.hfuu.iao.io.StudentReader; 7 import cn.edu.hfuu.iao.person.Person; 8 9 /** testing our package structure */</pre>
► JRE System ► libs	n Library [java-8-oper	10 public class Main { 11 /** The main routine reading person records of a certain type 7 gs we ignore this parameter */
gitignor . .project	Open in <u>N</u> ew Windo Sho <u>w</u> In	c void main(String[] args) {
	II] <u>C</u> opy ■ Copy Qualified	
	i∎ <u>P</u> aste X <u>D</u> elete	<pre>Crity anner.nextLine().charAt(0) { // check the first _</pre>
_	Remove from C <u>B</u> uild Path Refactor	·
bs-person u	in Import	<pre>Shift+Alt+T > // loop forever, see loop condition at bottom of erson = reader.read(scanner); // use the person r ut_orintln("You entered: " + nerson): // nrint ne :</pre>



	Quick Access 🛛 😰 🐯 💱
😫 Packag 🛛 🔭 Type Hi 👘 🗖	妃 Main.java 🛙 🗖
Image: Second Secon	

person.jar - person_user/libs



- We now want to use our library in our code
- Our main application shall be the same as in Lesson 17: *Packages* and *import*
- We create a new Eclipse project person_user and copy our Main class there
- Sadly, it won't compile, because now it misses all the required Person -related code
- Let's fix this.
- First, we create a new folder **libs** in our project to store all the libraries needed
- Then we copy our library person.jar into this folder
- We now right-click our project and select Properties



	• \$\$ •	Quick Access 🛛 😰 🐻
Packag 😫 🍃 Typ	oe Hi 😐 🗖 🔬 Main.java	n – –
S > JavaExamples [] > person_library > person_user [] * > src * ∰ > (default pa → Main.java	avaExamples m 2 3 impor	t java.util.Scanner; // import class Scanner from the java t cn.edu.hfuu.iao.io.ForeignExchangeStudentReader; // import C. cn.edu.hfuu.iao.io.PersonReader; t cn.edu.hfuu.iao.io.ProfessorReader; i.o. io.tostudentReader; .io. jerson.Person; ckage structure */
a) JRE System Lif Grlibs (∰person.jar ≩.classpath B.gitignore ≩.project	Open Type Hierarchy Show in © Copy © Copy Qualified Name © Paste % Delete % Remove From Context guild Path Suid Path Refactor Refactor © Import	<pre>re{ site-reading person records of a certain type we ignore this parameter */ carted a sunifictring[] args { the sain(String[] args { the sain(String[] args { the sain(String[] args { the sain(String[] args { the sain(String[] args {</pre>
	Report Report Report Report Report Report Report Run As gebug As yalidate Restore from Local History Tgam Compare With Replace With Comfoure	<pre>h = reader.read(scanner); // use the person re rprintin("You entered: *+ person); // print person re rintin("Type enter to continue, Ctrl-b text hasistilme()); // if user presed enter tilme(); // if user presed enter tilme(); // if she instead presed Ci</pre>

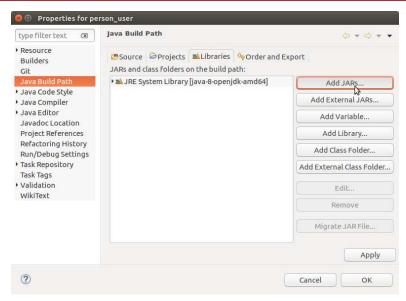
OOP with Java

Thomas Weise



- Our main application shall be the same as in Lesson 17: *Packages* and *import*
- We create a new Eclipse project person_user and copy our Main class there
- Sadly, it won't compile, because now it misses all the required Person -related code
- Let's fix this.
- First, we create a new folder **libs** in our project to store all the libraries needed
- Then we copy our library person.jar into this folder
- We now right-click our project and select Properties
- We choose Java Build Path , then Libraries , then click Add JARs...







- We create a new Eclipse project person_user and copy our Main class there
- Sadly, it won't compile, because now it misses all the required Person -related code
- Let's fix this.
- First, we create a new folder libs in our project to store all the libraries needed
- Then we copy our library person.jar into this folder
- We now right-click our project and select Properties
- We choose Java Build Path , then Libraries , then click Add JARs...
- We choose person.jar in the libs folder of our project and click



JAR Selection

Choose the archives to be added to the build path:

type filter text	Q
) 🗁 javaExamples	
🖻 person_library	
person_user	
• 🗁 bin	
▼	
🛃 person.jar	
• 🧀 src	
.classpath	
📄 .gitignore	
🗵 .project	
	Cancel OK

OOP with Java

Thomas Weise



- Sadly, it won't compile, because now it misses all the required Person -related code
- Let's fix this.
- First, we create a new folder libs in our project to store all the libraries needed
- Then we copy our library person.jar into this folder
- We now right-click our project and select Properties
- We choose Java Build Path, then Libraries, then click Add JARs...
- We choose person.jar in the libs folder of our project and click
- We click OK



type filter text 🛛 🗷	Java Build Path	⇔ - ⇔ -
Resource Builders Git	Bource BProjects ■Libraries Order and E JARs and class folders on the build path:	xport
Java Build Path	🛚 👼 person.jar - person_user/libs	Add JARs
Java Code Style Java Compiler	▶ 🛋 JRE System Library [java-8-openjdk-amd64]	Add External JARs
Java Editor		Add Variable
Javadoc Location Project References		Add Library
Refactoring History Run/Debug Settings		Add Class Folder
Task Repository Task Tags		Add External Class Folder
Validation WikiText		Edit
WINITEAL		Remove
		Migrate JAR File
		Apply



- Let's fix this.
- First, we create a new folder **libs** in our project to store all the libraries needed
- Then we copy our library person.jar into this folder
- We now right-click our project and select Properties
- We choose Java Build Path , then Libraries , then click Add JARs...
- We choose person.jar in the libs folder of our project and click
- We click OK
- And our project compiles, the compiler errors disappear.



Ů ▾ ∛ ▾ ❤ ❤ ❤ ❤ ❤ ♥ ▾	8. ?> .£ = 🕏 🕱 🕸 ▼ 🖸 ▼ 💁 ▼ 🔮 🥙 ▼ 🥵 🔗 ▼ 🖓 🥩 🖉 🗓 🗓
ੀ ▾ ∛ ▾ ♥ ♥ ♥ ♥ ♥ ♥	Quick Access 🕴 😤 🐯 😵
Packag 31 1 Type Hi Image: Constraint of the second s	Mainjava B I hainjava B <pi b<="" hainjava="" p=""> I hainjava B I hain</pi>

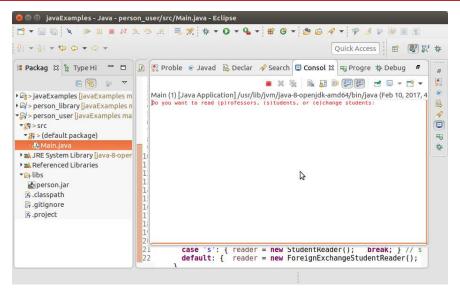


- First, we create a new folder **libs** in our project to store all the libraries needed
- Then we copy our library person.jar into this folder
- We now right-click our project and select Properties
- We choose Java Build Path , then Libraries , then click Add JARs...
- We choose person.jar in the libs folder of our project and click
- We click OK
- And our project compiles, the compiler errors disappear.
- We can now run it, and it will run beautifully



⇒ person_ > person_ * > src * > src * > (defa b ⇒ JRE \$ b ⇒ Refet * ;; libs @ per	The second secon	2 3 import cn. 4 import cn. 5 import cn. 6 import cn. 7 import cn. 8		ssorReader; ntReader; erson;	
> a JRE S > a Refe ♥@ylibs @per	Open			re */	
	Open Type Hierarchy Show In		ain routine reading m args we ignore this atic void main(String eader reader:	<pre>person records of a certa s parameter */ g[] args) { er(System.in); // create</pre>	
🔐 .gitig 🖗 .proj	 Copy Copy Qualified Name Paste Delete 		<pre>c err.println("Do you y v (scanner.nextLine(). 'p': { reader = new }</pre>	<pre>want to read (p)rofessors charAt(0)) { // check the ProfessorReader(); break; StudentReader(); break;</pre>	<pre>s, (s)tu first } // p</pre>
	 Remove from Context Build Path Source Refactor 		<pre>lt: { reader = new > > >) { // loop forever, > n person = reader.</pre>	ForeignExchangeStudentRea see loop condition at bo ad(scanner); // use the p	der();
	 Import Export 		m.out.println("Type	<pre>ntered: " + person); // p enter to continue, Ctrl-D) { // if user pressed en</pre>	to exinter
	References Declarations		<pre>> nner.nextLine(); tinue; ></pre>	<pre>// we read the line a // and do another ite // if she instead pre</pre>	eration essed Ct
	Refresh Assign Working Sets		s <mark>n:</mark>	// no next line and w	e exit
	Run As		> Java Application		
	Debug As Validate Restore from Local Histor Team Compare With		> Ru <u>n</u> Configurations		





Using Libraries outside Eclipse



• But what if you cannot use Eclipse and want to compile/execute your program from the command line?



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library
- To compile/run a program depending on a library, the library must be in the classpath, the place where Java looks for classes



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library
- To compile/run a program depending on a library, the library must be in the classpath, the place where Java looks for classes
- Compilation (under Linux)



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library
- To compile/run a program depending on a library, the library must be in the classpath, the place where Java looks for classes
- Compilation (under Linux):
 - 🌒 open your terminal, cd into src folder



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library
- To compile/run a program depending on a library, the library must be in the classpath, the place where Java looks for classes
- Compilation (under Linux):
 - open your terminal, cd into src folder
 - type in javac -d ../bin -cp .:../libs/person.jar Main.java and press return



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library
- To compile/run a program depending on a library, the library must be in the classpath, the place where Java looks for classes
- Compilation (under Linux):
 - 🌒 open your terminal, cd into src folder
 - type in javac -d ../bin -cp .:../libs/person.jar Main.java and press return
 - explanation: the -d ../bin tells the compiler to put the .class files into the bin folder, the -cp .:../libs/person.jar tells the compiler that the library person.jar is part of the classpath, i.e., can be used to look up classes, Main.java is the class to compile



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library
- To compile/run a program depending on a library, the library must be in the classpath, the place where Java looks for classes
- Compilation (under Linux)
- Execution (under Linux)



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library
- To compile/run a program depending on a library, the library must be in the classpath, the place where Java looks for classes
- Compilation (under Linux)
- Execution (under Linux):
 - open your terminal, cd into bin folder



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library
- To compile/run a program depending on a library, the library must be in the classpath, the place where Java looks for classes
- Compilation (under Linux)
- Execution (under Linux):
 - open your terminal, cd into bin folder
 - 🧶 type in java -cp .:../libs/person.jar Main and press return



- But what if you cannot use Eclipse and want to compile/execute your program from the command line?
- Assume the same directory structure as before: a project folder containing a folder src with the source code, a folder bin for the compiled .class files, and a folder lib containing our person.jar library
- To compile/run a program depending on a library, the library must be in the classpath, the place where Java looks for classes
- Compilation (under Linux)
- Execution (under Linux):
 - open your terminal, cd into bin folder
 - 🥹 type in java -cp .:../libs/person.jar Main and press return
 - explanation: the -cp .:../libs/person.jar tells the compiler that the library person.jar is part of the classpath, i.e., can be used to look up classes, Main is the class to execute



• Executable jar archives are just the same as normal jar archives



- Executable jar archives are just the same as normal jar archives
- With the small difference that their manifest specifies a main class



- Executable jar archives are just the same as normal jar archives
- With the small difference that their manifest specifies a main class
- Let us now make an executable out of our simple vertical ball throw example from Lesson 6: Console I/O



Listing: Vertical Ball Throw with Console I/O

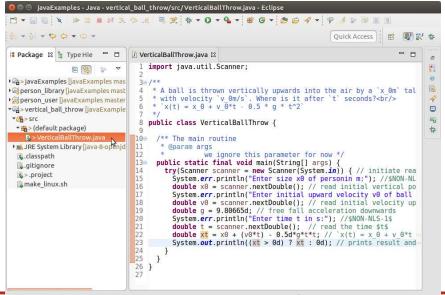
```
import java.util.Scanner;
```

```
* A ball is thrown vertically upwards into the air by a xom tall person
public class VerticalBallThrow {
 public static final void main(String[] args) {
    try (Scanner scanner = new Scanner (System.in)) { // initiate reading from System.in, ignore for now
      System.err.println("Enter_size_1x0_lof_personin_m:"); //$NON-NLS-1$
      double x0 = scanner.nextDouble(); // read initial vertical position x_0
      System.err.println("Enteruinitialuupwarduvelocityuv0uofuballuinum/s:"); //$NON-NLS-1$
      double v0 = scanner.nextDouble(); // read initial velocity upwards vo
      double g = 9.80665d; // free fall acceleration downwards
      System.err.println("Enter_time_t_in_s:"); //$NON-NLS-1$
      double t = scanner.nextDouble(); // read the time $t$
      double xt = x0 + (v0*t) - 0.5d*g*t*t; // x(t) = x_0 + v_0 * t - 0.5 * g * t^2
      System.out.println((xt > 0d) ? xt : 0d); // prints result and makes sure the ball stops at
    7
```



• We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it



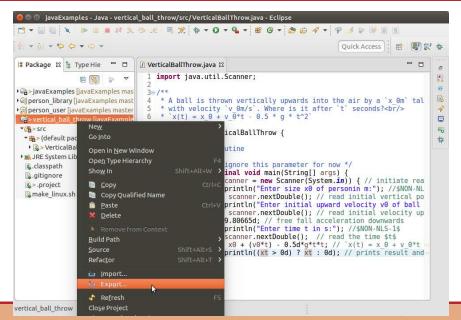


VerticalBallThrow.java - vertical_ball_throw/src



- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...







- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...
- In the export wizzard, we choose Java and then JAR file and press Next



elect	7
xport resources into a JAR file on the local file system.	
Select an export wizard:	
type filter text	()
🗁 General	
🔁 Install	
🔊 Java	
JAR file	
@Javadoc	
GRunnable JAR file	
▶	
> Team	
> ML	



- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button



Select the resources to <u>e</u> xport	
	R.classpath B.gitignore B.groject make_linux.sh
Export generated class fil Export all output folders I Export Java source files a Export refactorings for ch Select the export destination:	or checked projects
 Export all output folders f Export Java source files a Export refactorings for ch 	or checked projects nd resources
Export all output folders 1 Export Java source files a Export refactorings for ch Select the export destination:	or checked projects nd resources ecked projects.Select refactorings T Btowse

OOP with Java

Thomas Weise



- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it executable.jar and hit OK



	e.jar									
Home Desktop		javaExamples	lessons	26_librarie	s_and_executa	ibles vert	ical_ball_throw		-1	
program	Name							*	Size	Modified 06:00
OPPO	逼 src									06:00
Other Locatio										
										*.jar;



- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it executable.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next



elect the resources to <u>e</u> xport:	
	Ø. classpath B. gitignore Ø. project B. make_linux.sh
Export generated class files a	nd resources
elect the export destination:	resources ted projects.Select refactorings
Export Java source files and i	resources ted projects.Select refactorings /26_libraries_and_executables/

OOP with Java

Thomas Weise



- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it executable.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next
- We click Next



AR Packaging (ins for the JAR export.	
	or handling problems:	10
	files with compile errors	
and the second second second	files with compile warnings	
Create sour	te folder structure	
🖉 Build projec	ts if not built automatically	
Save the des	cription of this JAR in the workspace	
Description file		Browse

Thomas Weise



- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it executable.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next
- We click Next
- We now need to select a Main <u>class</u> and therefore click Browse... after the corresponding input field



pecify the manifest:	
Generate the manifest file	
Save the manifest in the workspace	
Use the saved manifest in the generated JAR description	on file
Manifest file:	Browse
O Use existing manifest from workspace	
Manifest file:	Browse
Seal contents: Seal the JAR JAF	Rsealed Details
Seal some packages	Details
select the class of the application entry point: Main class:	
Mail Liass.	Browse

OOP with Java

Thomas Weise



- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it executable.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next
- We click Next
- We now need to select a Main <u>class</u> and therefore click Browse... after the corresponding input field
- We choose our class VerticalBallThrow in the next screen and click
 OK



Select Main Class

Select the class which is the application's entry point:

©_■ VerticalBallThrow

(default package) - vertical ball throw/src

? Cancel

OK



- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it executable.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next
- We click Next
- We now need to select a Main <u>class</u> and therefore click Browse... after the corresponding input field
- We choose our class VerticalBallThrow in the next screen and click
 OK
- We click Finish and the jar archive will be created



AR Manifest Specification	-
Customize the manifest file for the JAR file.	-
Specify the manifest:	
O Generate the manifest file	
Save the manifest in the workspace	
Use the saved manifest in the generated JAR description file	
Manifest file:	Browse
○ Use existing manifest from workspace	
Manifest file:	Browse
Seal contents: Seal the JAR JAR sealed	Detajls
○ Seal some <u>p</u> ackages	Details
Select the class of the application entry point:	
Main class: VerticalBallThrow	Browse

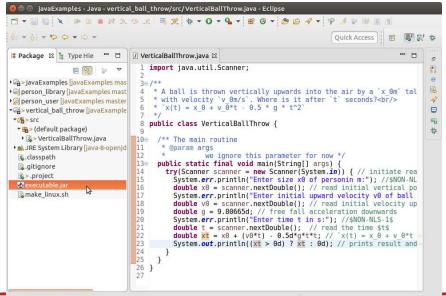
OOP with Java

Thomas Weise



- We first make an Eclipse project called vertical_throw with our class VerticalBallThrow in it
- We again right-click on the project and then click Export...
- In the export wizzard, we choose Java and then JAR file and press Next
- In the next screen, we hit the Browse button
- We choose a nice destination for our library call it executable.jar and hit OK
- Back in the previous screen, we mark our code folders and click Next
- We click Next
- We now need to select a Main <u>class</u> and therefore click Browse... after the corresponding input field
- We choose our class VerticalBallThrow in the next screen and click
 OK
- We click Finish and the jar archive will be created
- and we are done, the new file executable.jar has appeared...





executable.jar - vertical_ball_throw



• jar archives with a selected main class can be executed via java -jar command



- jar archives with a selected main class can be executed via java -jar command
- Open a terminal



- jar archives with a selected main class can be executed via java -jar command
- Open a terminal
- cd into the folder where jar archive executable.jar is located



- jar archives with a selected main class can be executed via java -jar command
- Open a terminal
- cd into the folder where jar archive executable.jar is located
- Type in java -jar executable.jar and press enter



- jar archives with a selected main class can be executed via java -jar command
- Open a terminal
- cd into the folder where jar archive executable.jar is located
- Type in java -jar executable.jar and press enter
- The program now runs!



- jar archives with a selected main class can be executed via java -jar command
- Open a terminal
- cd into the folder where jar archive executable.jar is located
- Type in java -jar executable.jar and press enter
- The program now runs!
- Note: If your jar archive depends on other libraries, you need to specify them via the -cp option which must come before the -jar stuff



- jar archives with a selected main class can be executed via java -jar command
- Open a terminal
- cd into the folder where jar archive executable.jar is located
- Type in java -jar executable.jar and press enter
- The program now runs!
- Note: If your jar archive depends on other libraries, you need to specify them via the -cp option which must come before the -jar stuff
- Note 2: Everything coming after -jar executable.jar will be passed as command line arguments to the args parameter of the static void main method of the program



- We have learned about jar archives which are special zip archives
- They store the .class files and created resources of a Java project
- They can be used as libraries, i.e., to package a set of classes which belong together into one archive and use this archive in many different applications
- We can also define a main class for a jar archive, then the archive becomes executable, i.e., we can ship a whole application as a single file instead of a bunch of files, package folders, and resources
- jar archives are a big thing in the Java world, any project you will work with will definitely use lots of libraries
- There exist incredibly many open source libraries in the Java world
- If we need some general functionality (I/O, maths, AI, parallelization, ...), we should always first look for an open source library





谢谢 Thank you

Thomas Weise [汤卫思] tweise@hfuu.edu.cn http://iao.hfuu.edu.cn

Hefei University, South Campus 2 Institute of Applied Optimization Shushan District, Hefei, Anhui, China