合肥學院
HEFEI UNIVERSITY

OOP with Java

26. Libraries and Executables

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · 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

Outline

website

Introduction

- Java programs usually consist of lots of `.java` or `.class` files

Introduction

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

Introduction

- 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

Introduction

- 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

Introduction

- 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

Introduction

- 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

Introduction

- 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

Introduction

- 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

Why Libraries?

- We have learned a two ways to structure code

Why Libraries?

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

Why Libraries?

- 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`

Why Libraries?

- 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

Why Libraries?

- 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.

Why Libraries?

- 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

Why Libraries?

- 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

Why Libraries?

- 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

Why Libraries?

- 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

Why Libraries?

- 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

Why Libraries?

- 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

Why Libraries?

- 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

Why Libraries?

- 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

Why Libraries?

- 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

The `Person` Class

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;
  }

  /** return a string representation of this person record */
  public String toString() {
    return this.givenName + '␣' + this.familyName;
  }
}
```

Professor Class

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. " + super.toString(); //$NON-NLS-1$
  }

}
```

Student Class

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 */
  @Override // mark this method explicitly as overridden
  public String toString() {
    return "student " + super.toString(); //$NON-NLS-1$
  }
}
```

Foreign Exchange Student Class

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;
  }

  /** override toString() from Person */
  @Override // mark this method explicitly as overridden
  public String toString() {
    return super.toString() + "␣from␣" + this.homeCountry;//$NON-NLS-1$
  }
}
```

A Person Reader

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
   * @return 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);
  }
}
```

A Professor Reader

Listing: Professor 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 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("Enter professor's family name:"); //$NON-NLS-1$
    String familyName = scanner.nextLine(); // read a string from scanner
    System.err.println("Enter professor's given name:"); //$NON-NLS-1$
    String givenName = scanner.nextLine(); // read a string from scanner

    return new Professor(familyName, givenName);
  }
}
```

A Student Reader

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 ID:"); //$NON-NLS-1$
    String id = scanner.nextLine(); // read a string from scanner

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

A Foreign Exchange Student Reader

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)
   * @return the new person record */
  @Override
  public ForeignExchangeStudent read(Scanner scanner) {
    System.err.println("Enter exchange student's family name:"); //$NON-NLS-1$
    String familyName = scanner.nextLine(); // read a string from scanner
    System.err.println("Enter exchange student's given name:"); //$NON-NLS-1$
    String givenName = scanner.nextLine(); // read a string from scanner
    System.err.println("Enter exchange student's ID:"); //$NON-NLS-1$
    String id = scanner.nextLine(); // read a string from scanner
    System.err.println("Enter exchange student's home country:"); //$NON-NLS-1$
    String country = scanner.nextLine(); // read a string from scanner

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

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

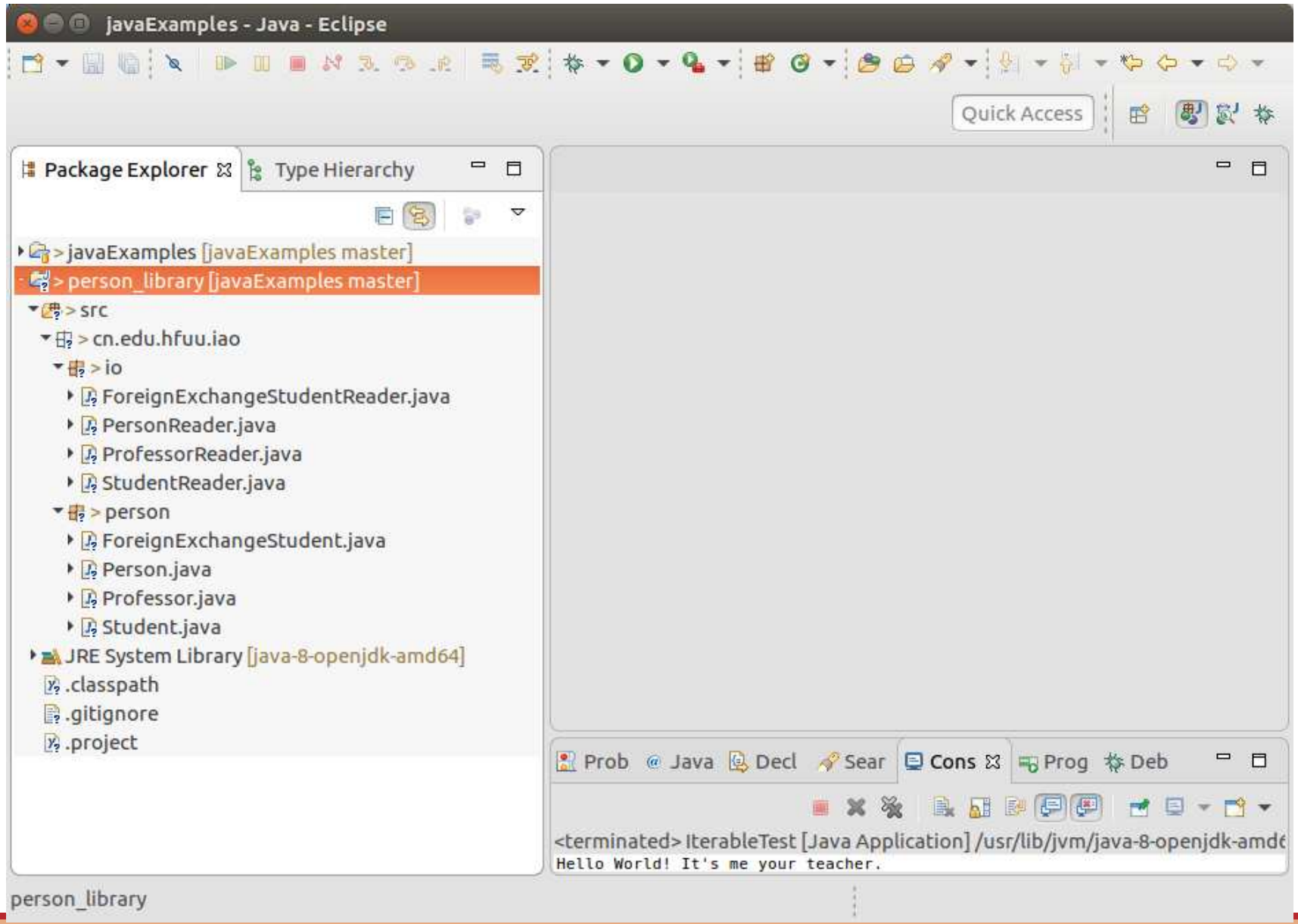
Creating a Library

- 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

Creating the `person.jar` Library

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

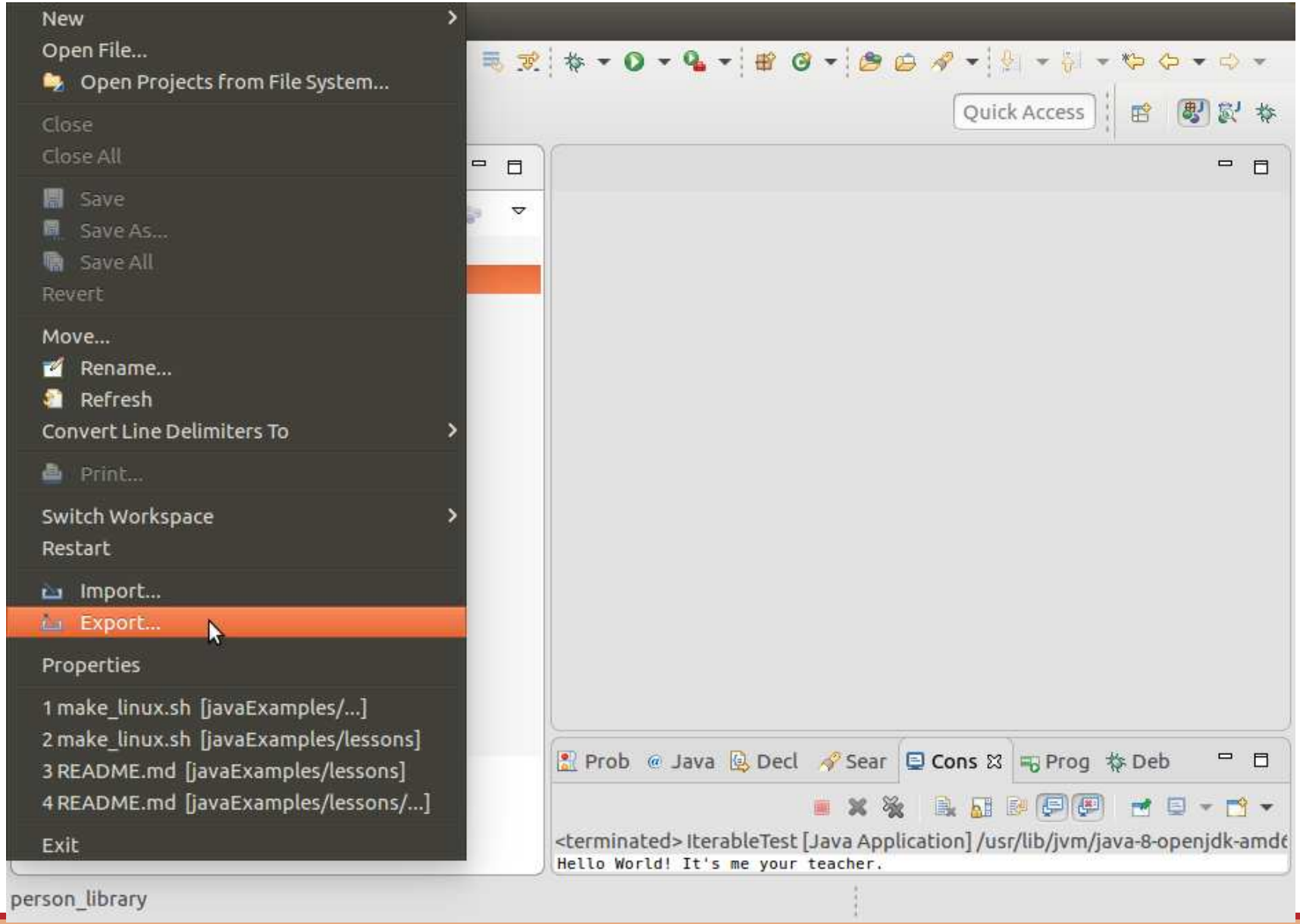
Creating the `person.jar` Library



Creating the `person.jar` Library

- 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

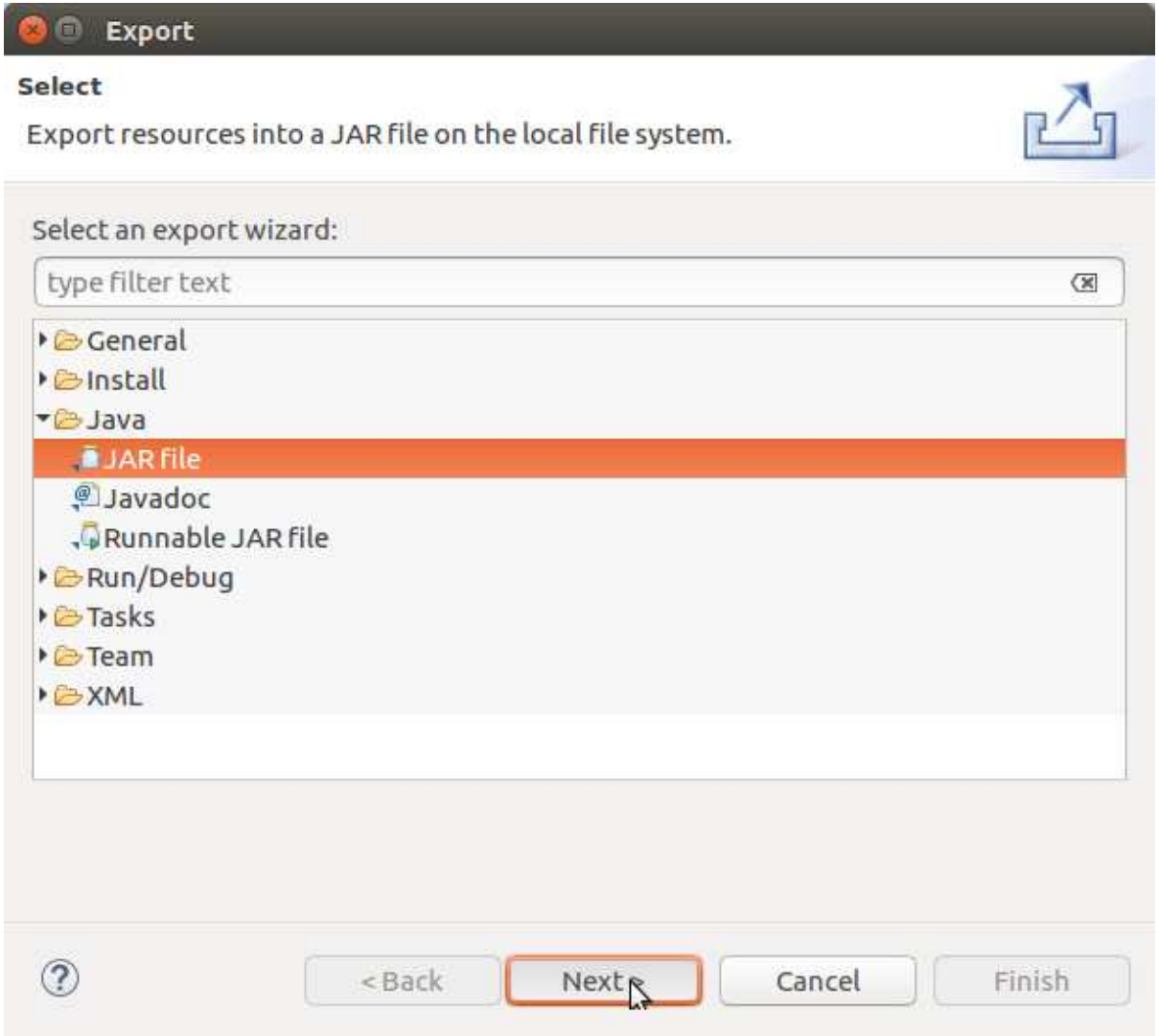
Creating the `person.jar` Library



Creating the `person.jar` Library

- 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`

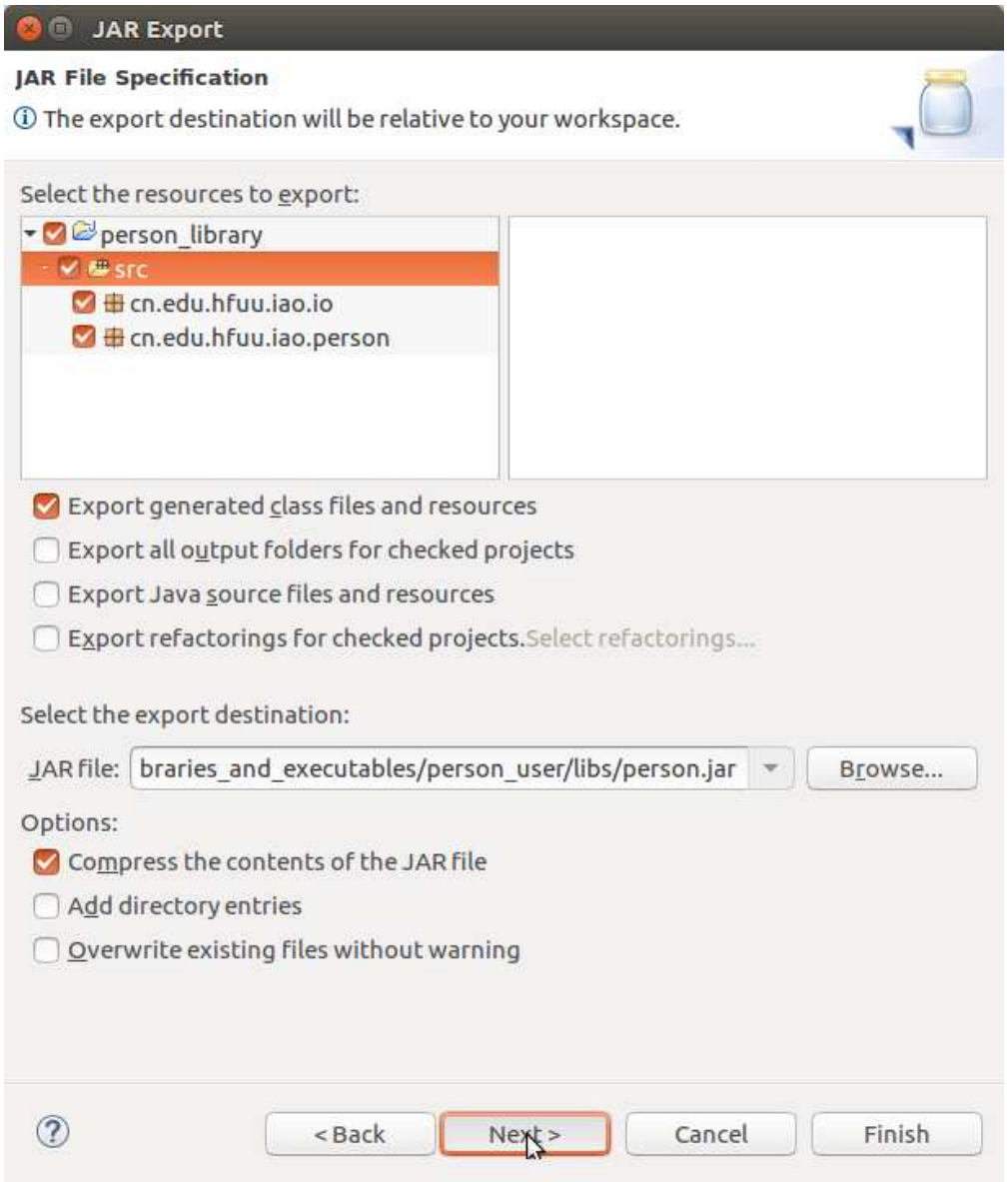
Creating the `person.jar` Library



Creating the `person.jar` Library

- 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

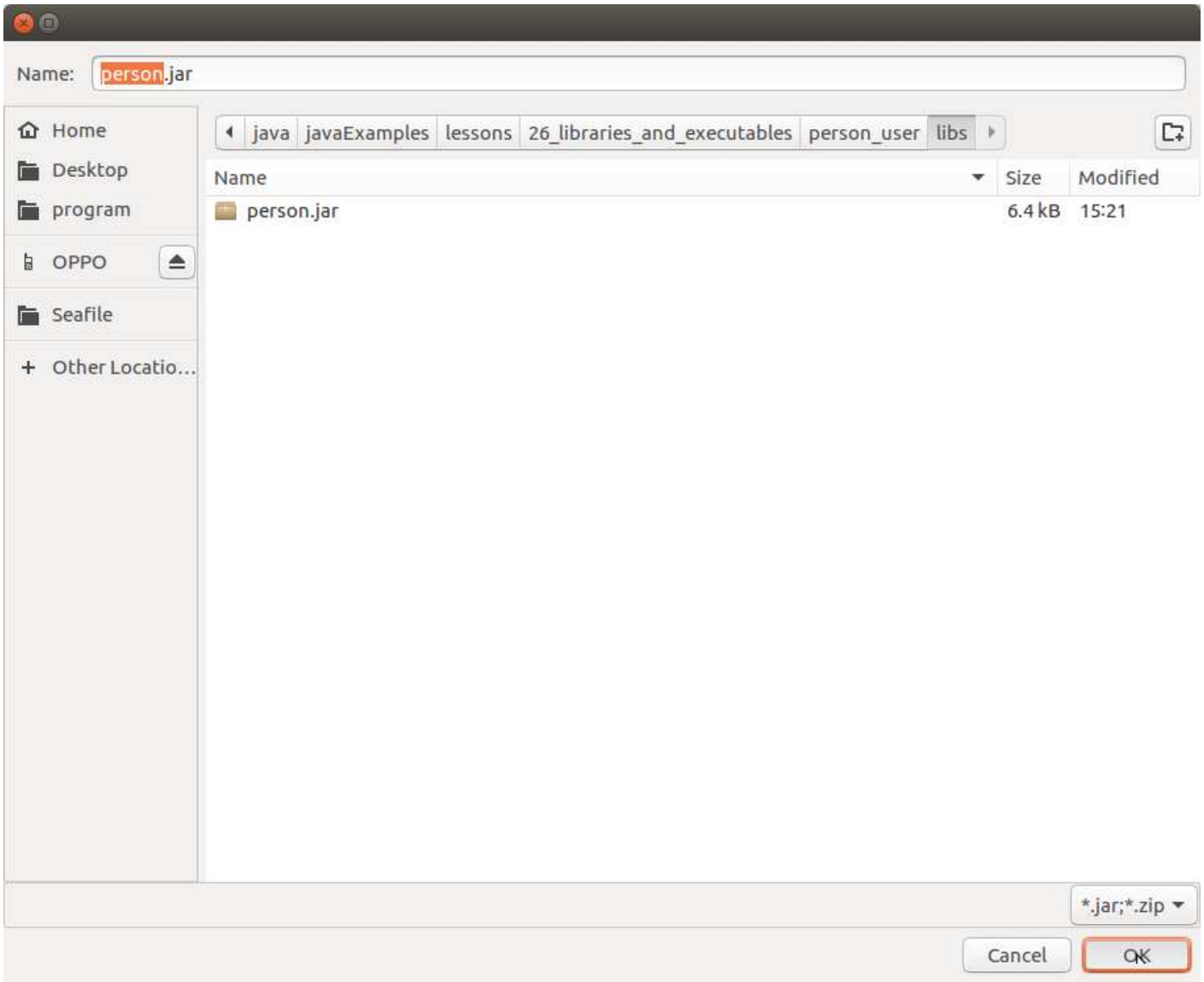
Creating the `person.jar` Library



Creating the `person.jar` Library

- 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`

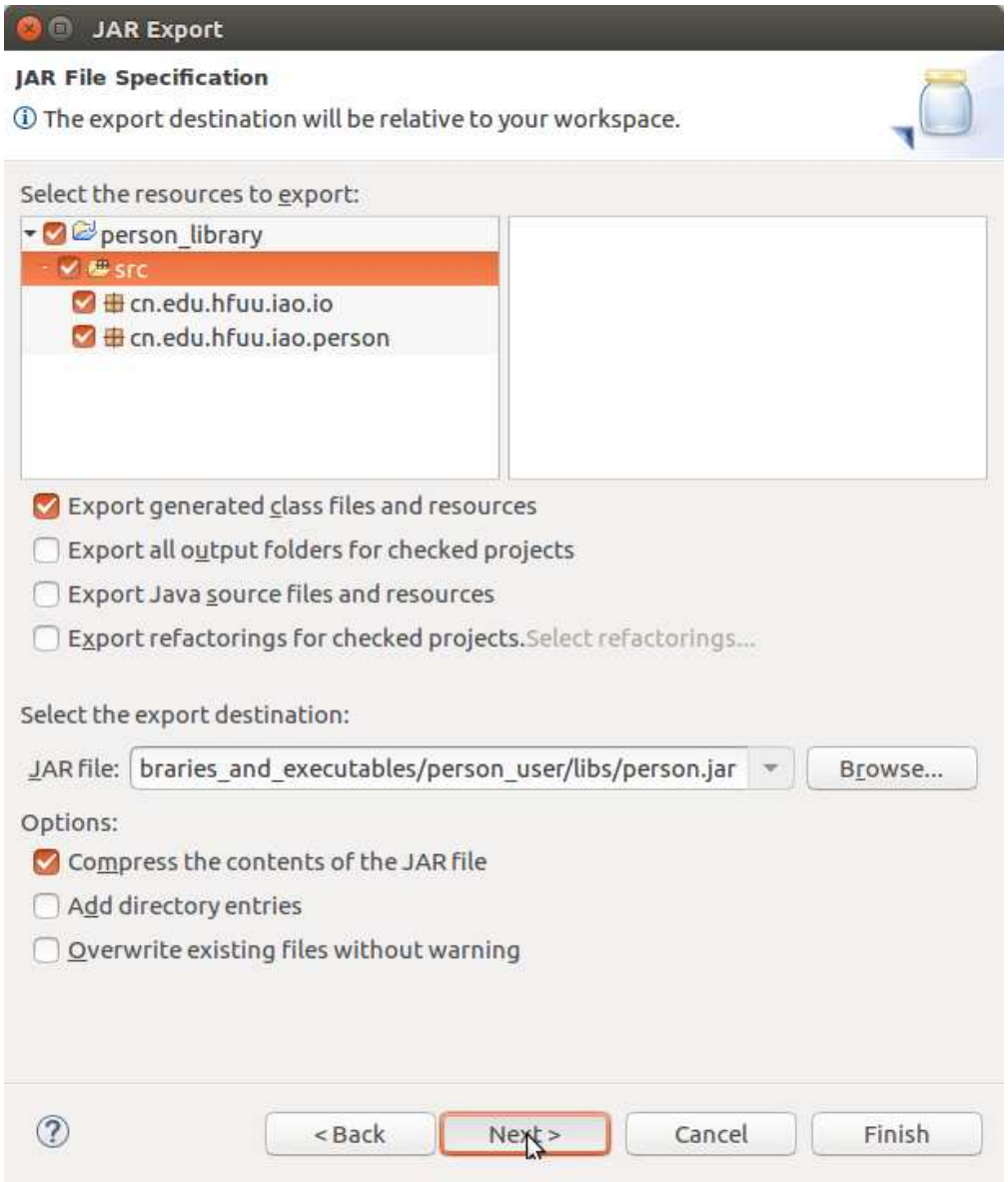
Creating the `person.jar` Library



Creating the `person.jar` Library

- 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...)

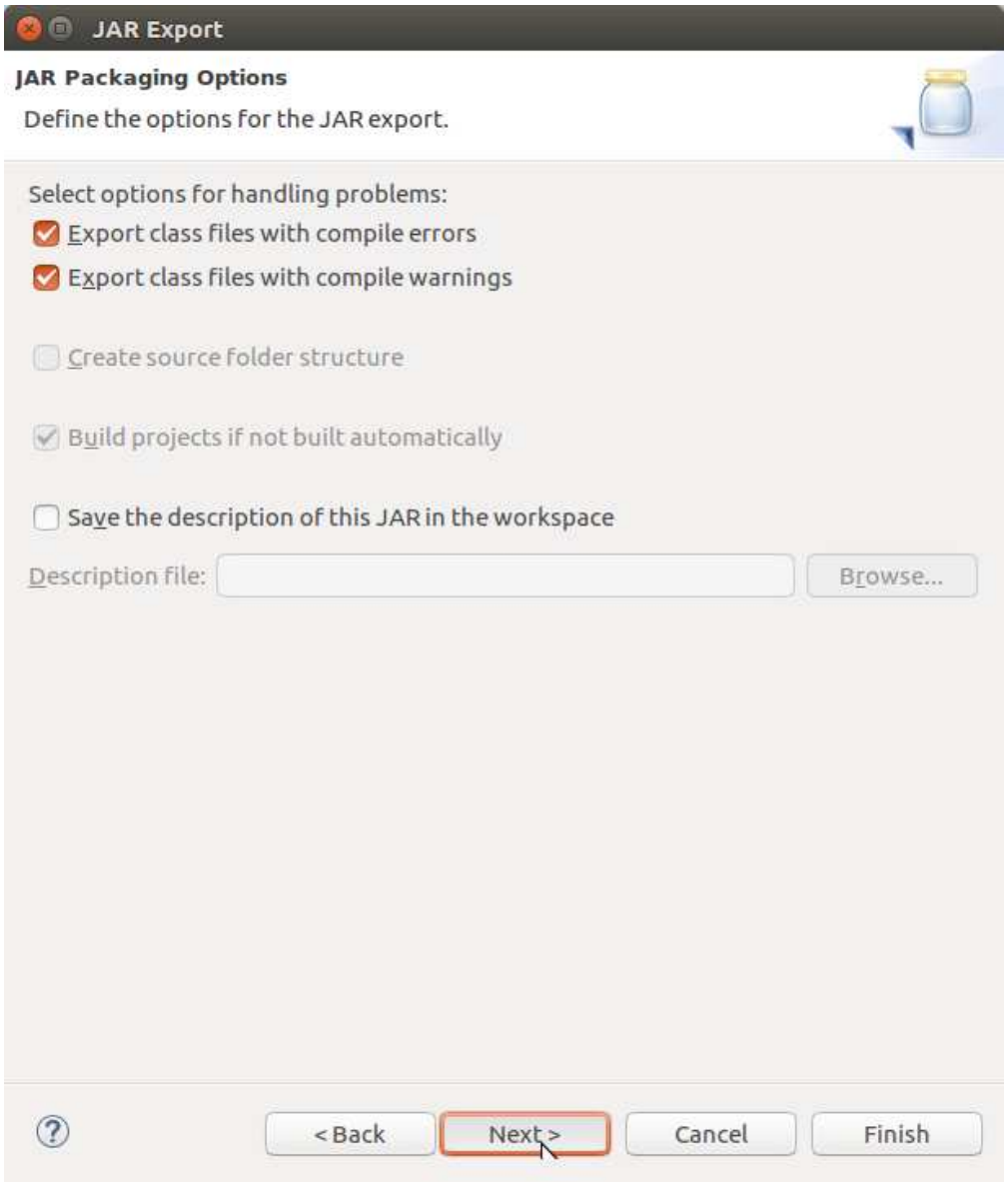
Creating the `person.jar` Library



Creating the `person.jar` Library

- 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`

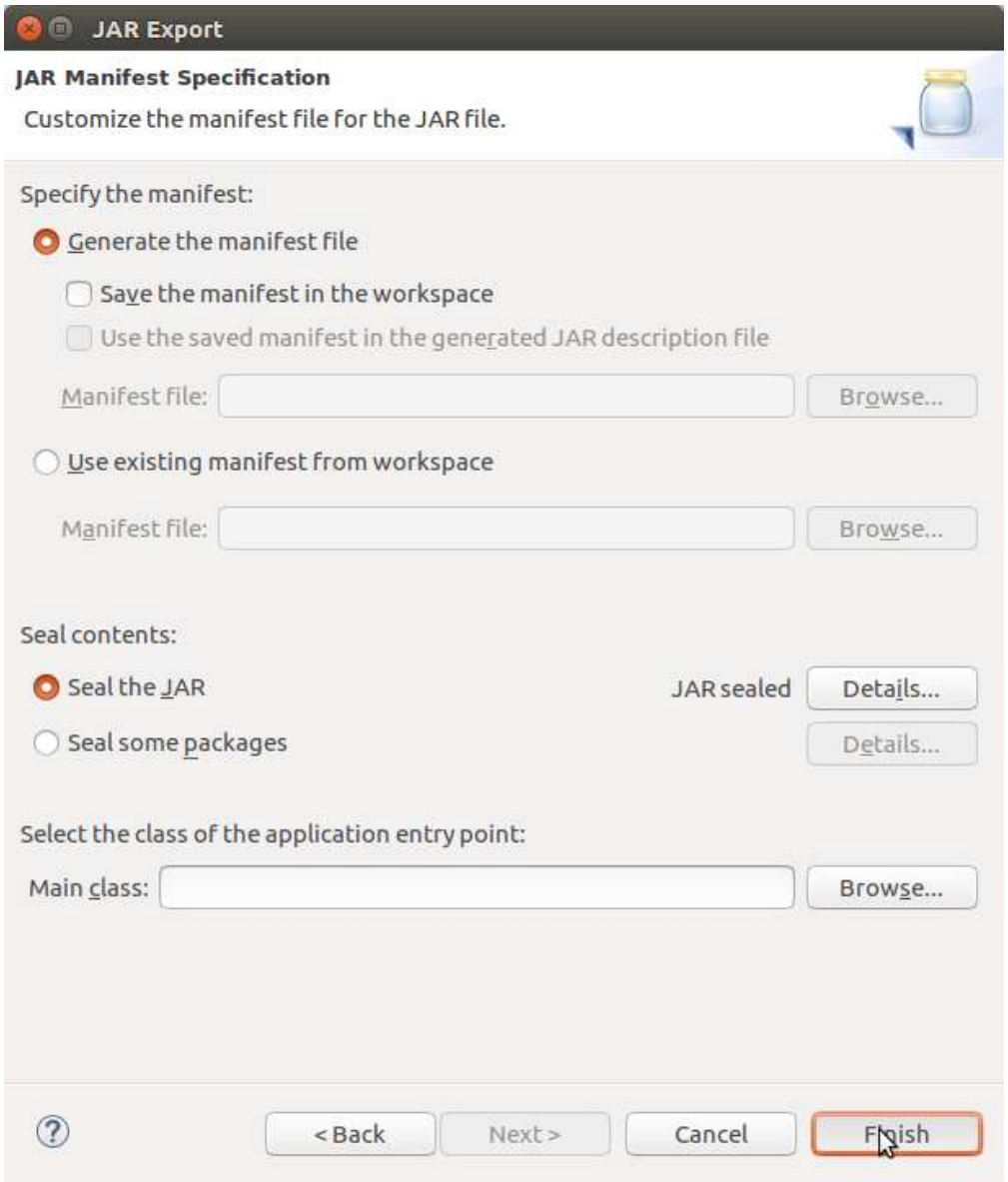
Creating the `person.jar` Library



Creating the `person.jar` Library

- 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

Creating the `person.jar` Library



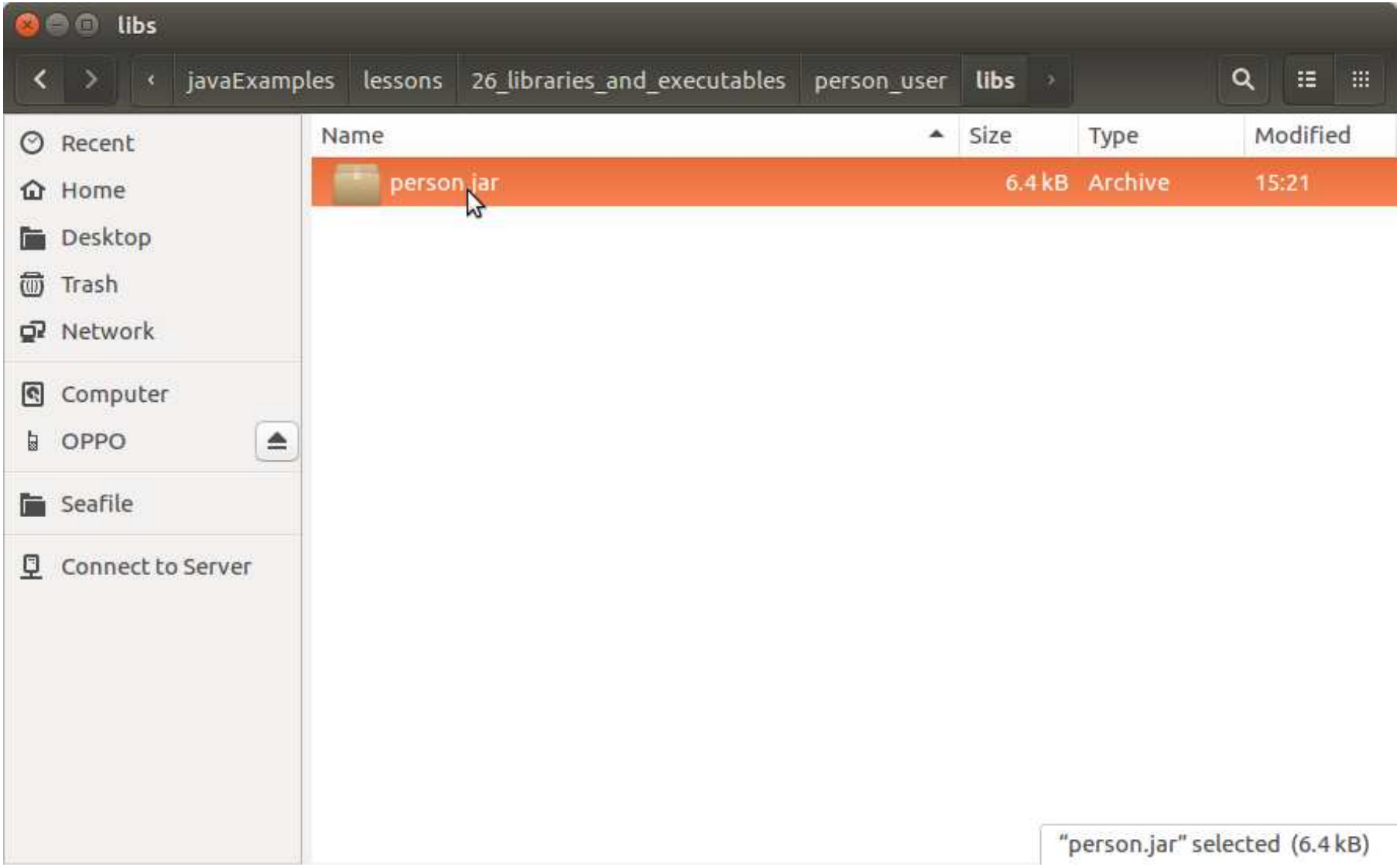
Creating the `person.jar` Library

- 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...

Checking the `person.jar` Library

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

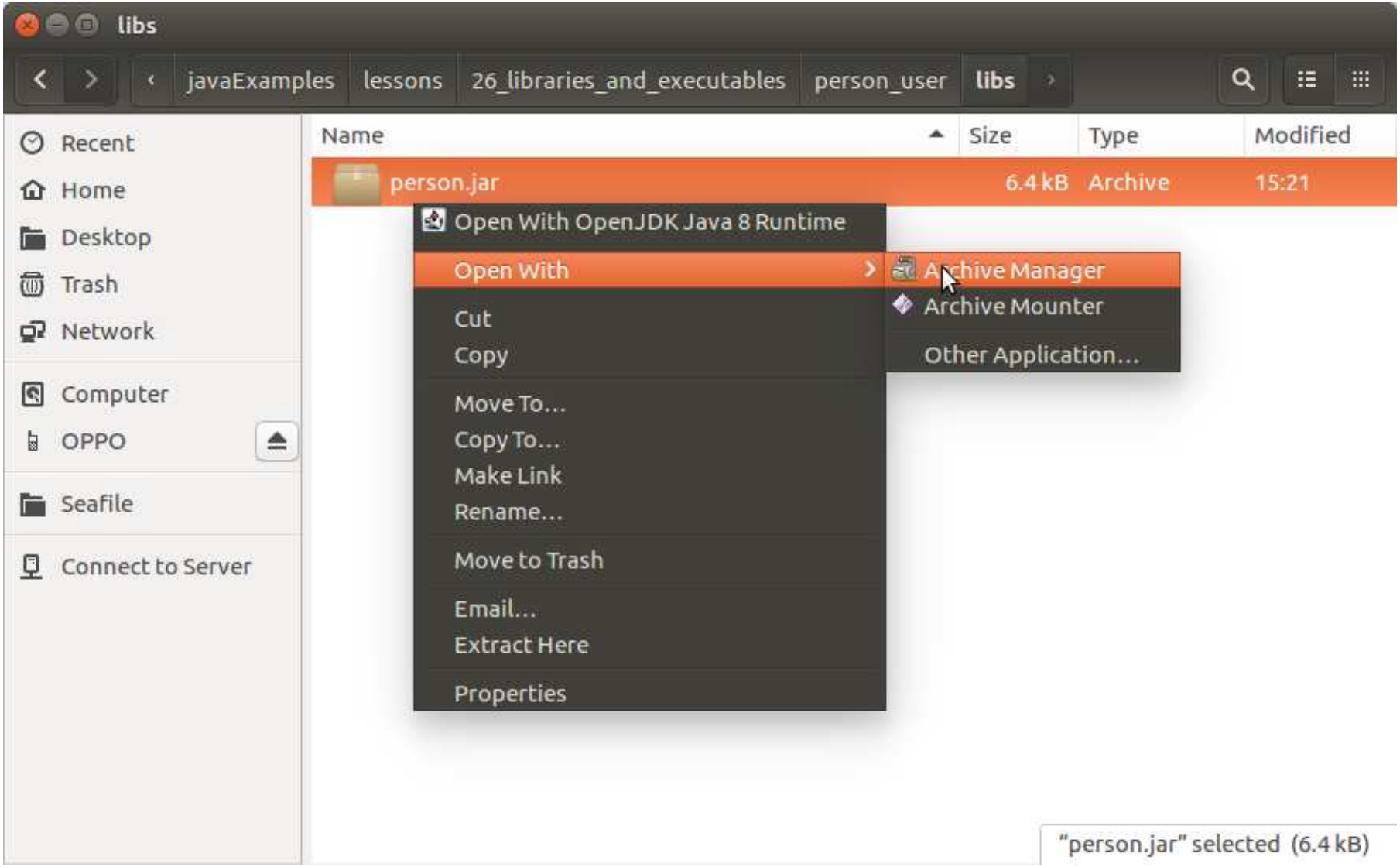
Checking the `person.jar` Library



Checking the `person.jar` Library

- 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*

Checking the `person.jar` Library



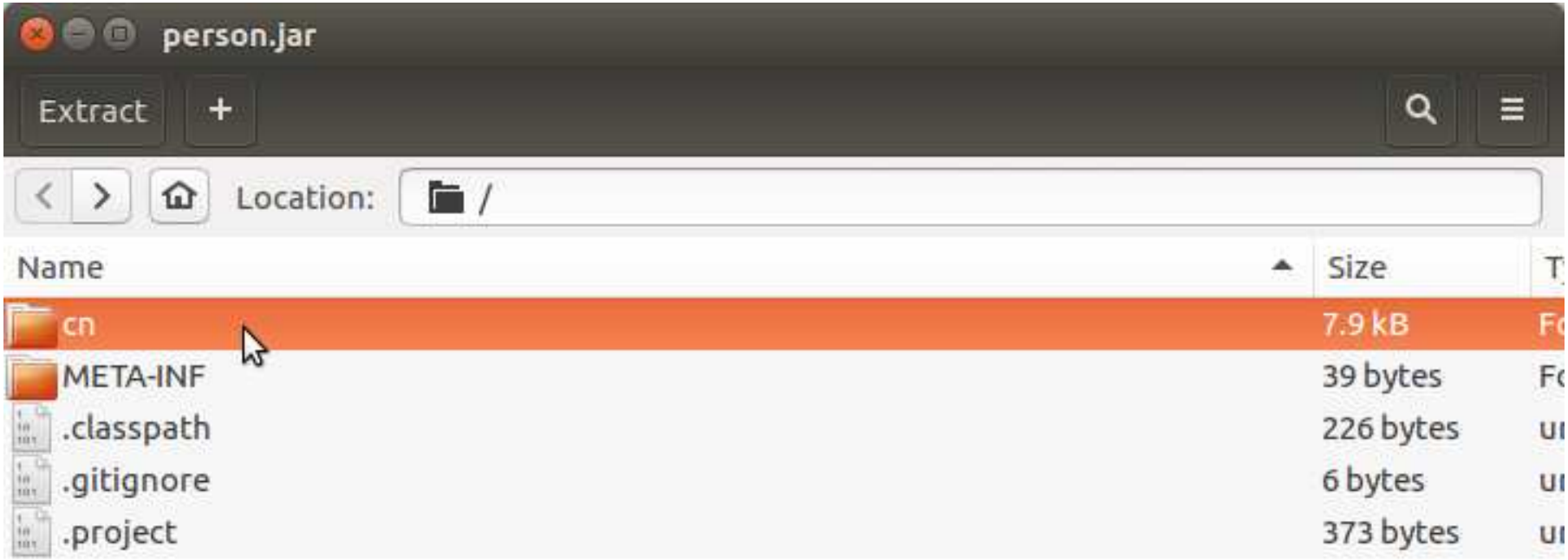
Checking the `person.jar` Library

- 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)

Checking the `person.jar` Library

- 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

Checking the `person.jar` Library



Checking the `person.jar` Library

Checking the `person.jar` Library

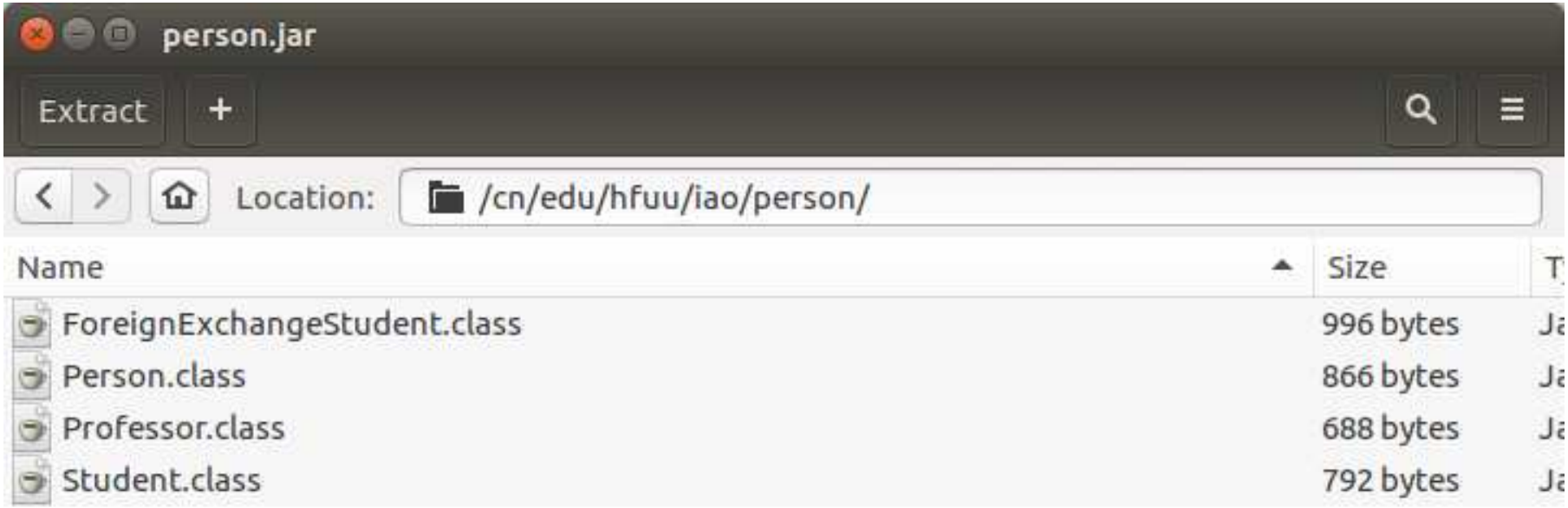
Checking the `person.jar` Library

Checking the `person.jar` Library

Checking the `person.jar` Library

- 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

Checking the `person.jar` Library



Checking the `person.jar` Library

- 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

Using a Library

- 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`

A Main class using our Person classes

Listing: A Main class using our Person classes

```
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;                 // and I/O classes from our sub-packages
import cn.edu.hfuu.iao.io.ProfessorReader;
import cn.edu.hfuu.iao.io.StudentReader;
import cn.edu.hfuu.iao.person.Person;

/** testing our package structure */
public class Main {
  /** The main routine reading person records of a certain type from stdin
   *  @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("Do you want to read (p)rofessors, (s)tudents, or (e)change students: "); //$NON-NLS-1$

    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
    }

    for (;;) { // loop forever, see loop condition at bottom of loop
      Person person = reader.read(scanner); // use the person read to read a person
      System.out.println("You entered: " + 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();        // we read the line and continue
        continue;                  // and do another iteration
      }                            // if she instead pressed Ctrl-D or stdin ends, there is
      return;                      // no next line and we exit the main routine
    }
  }
}
```

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`

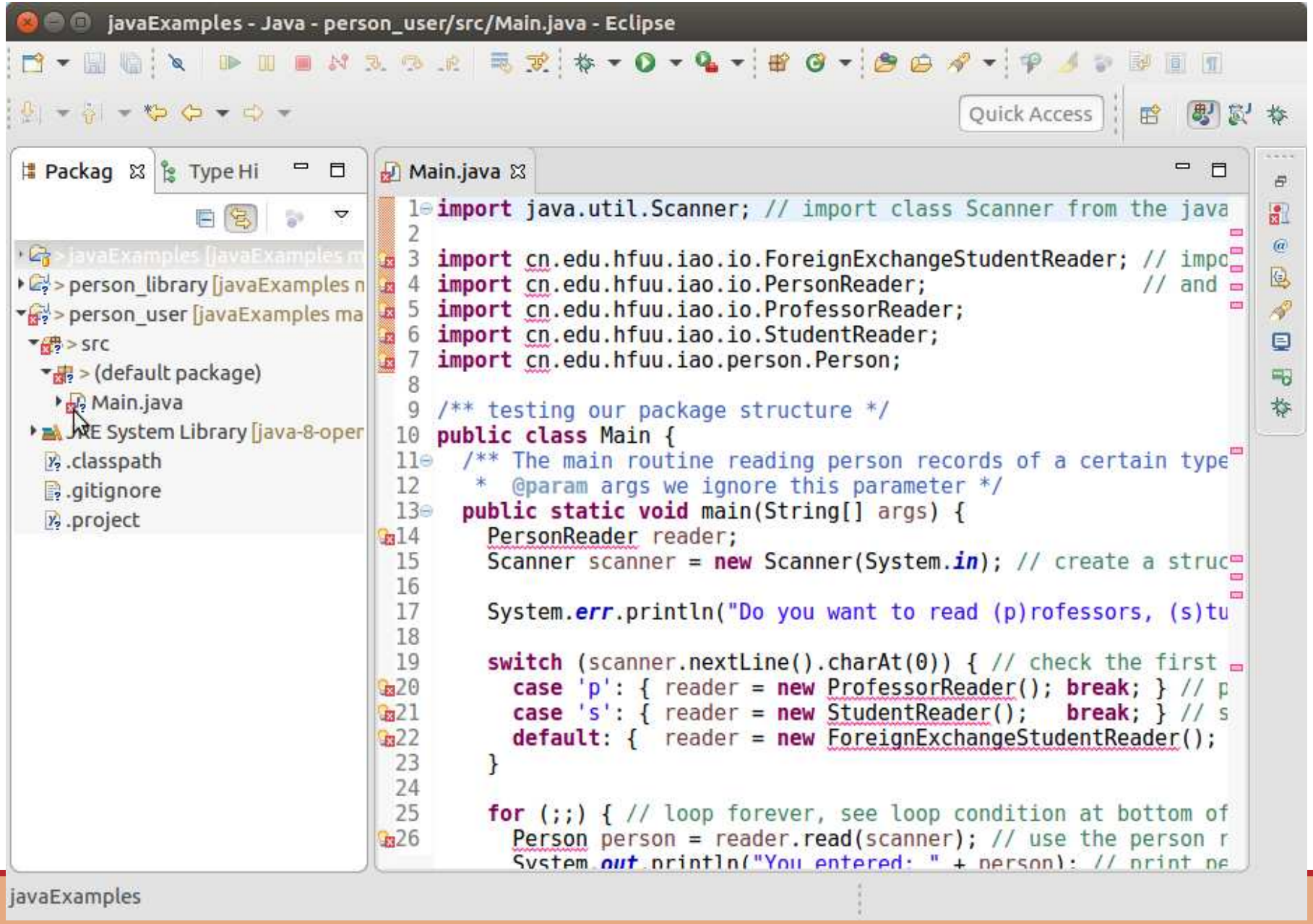
Using a Library

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

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`
- 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

Using a Library



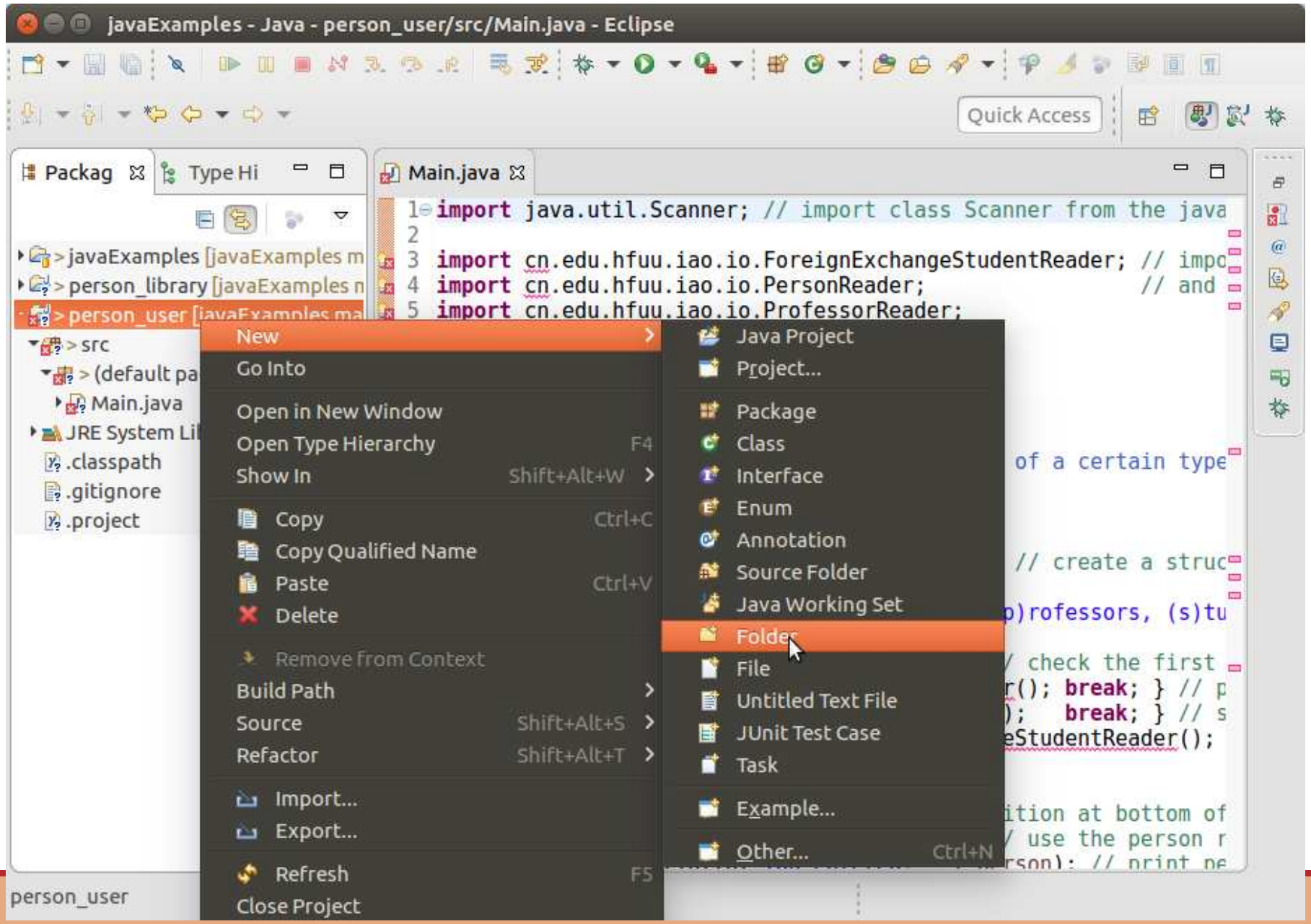
Using a Library

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

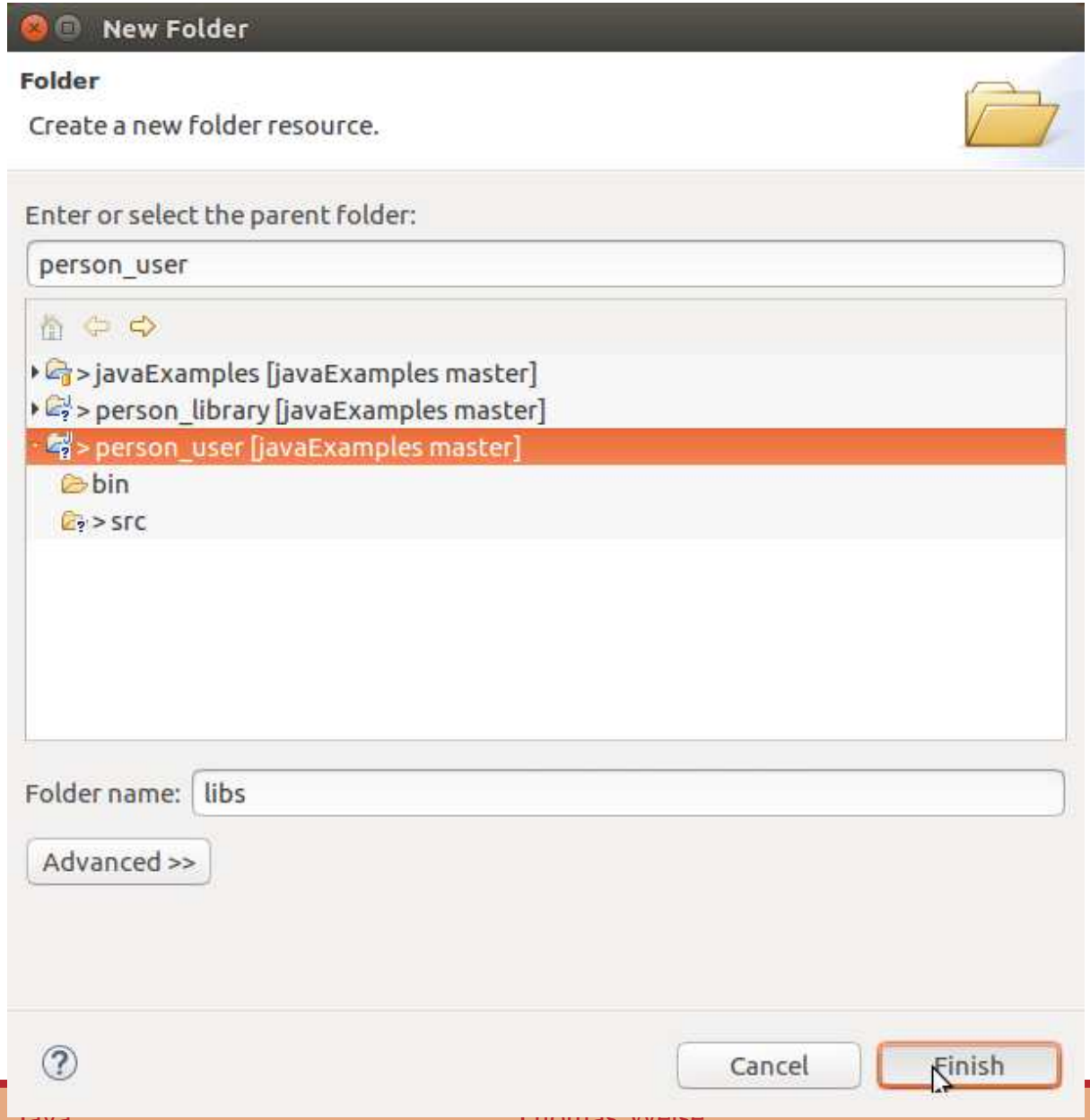
Using a Library

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

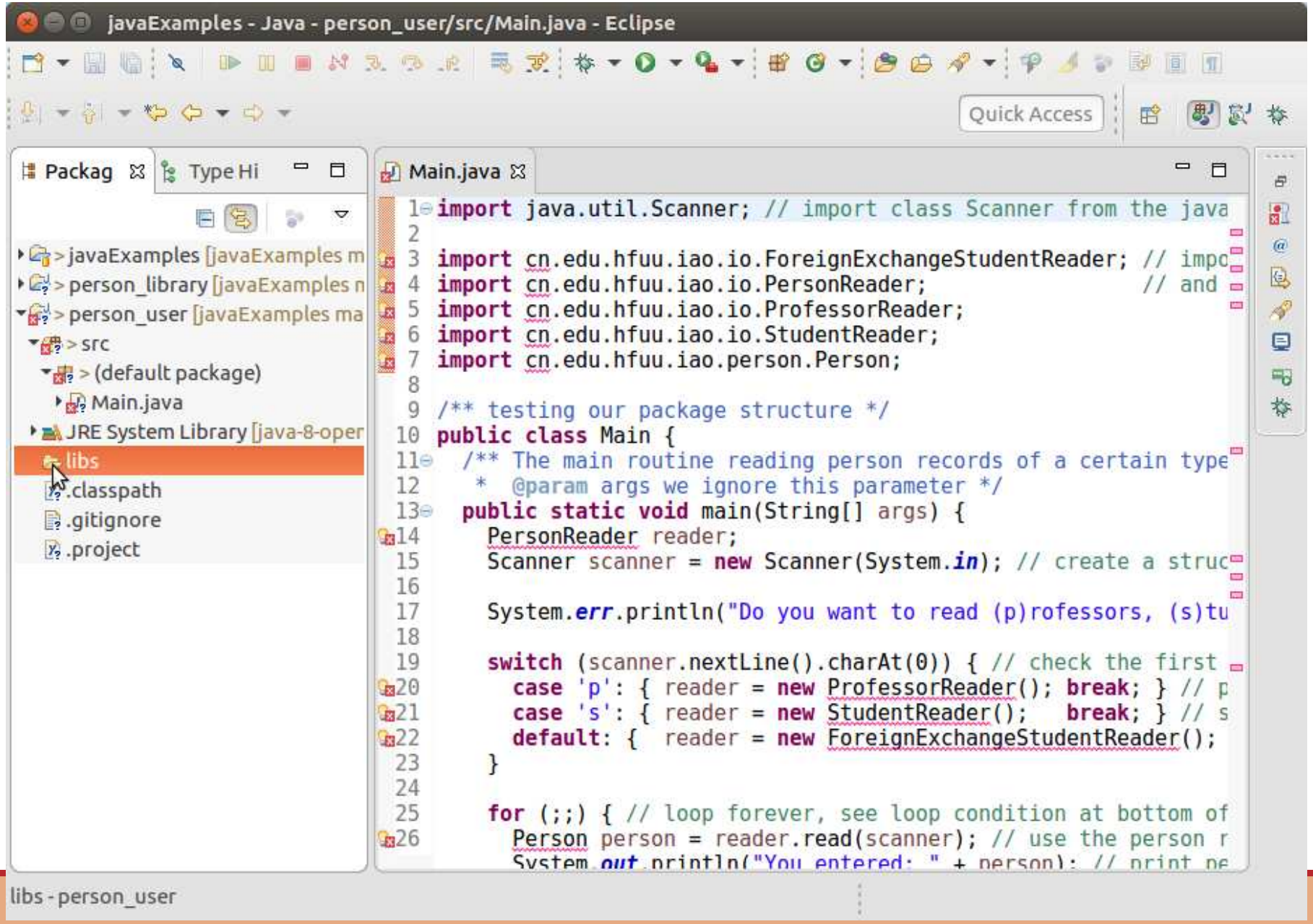
Using a Library



Using a Library



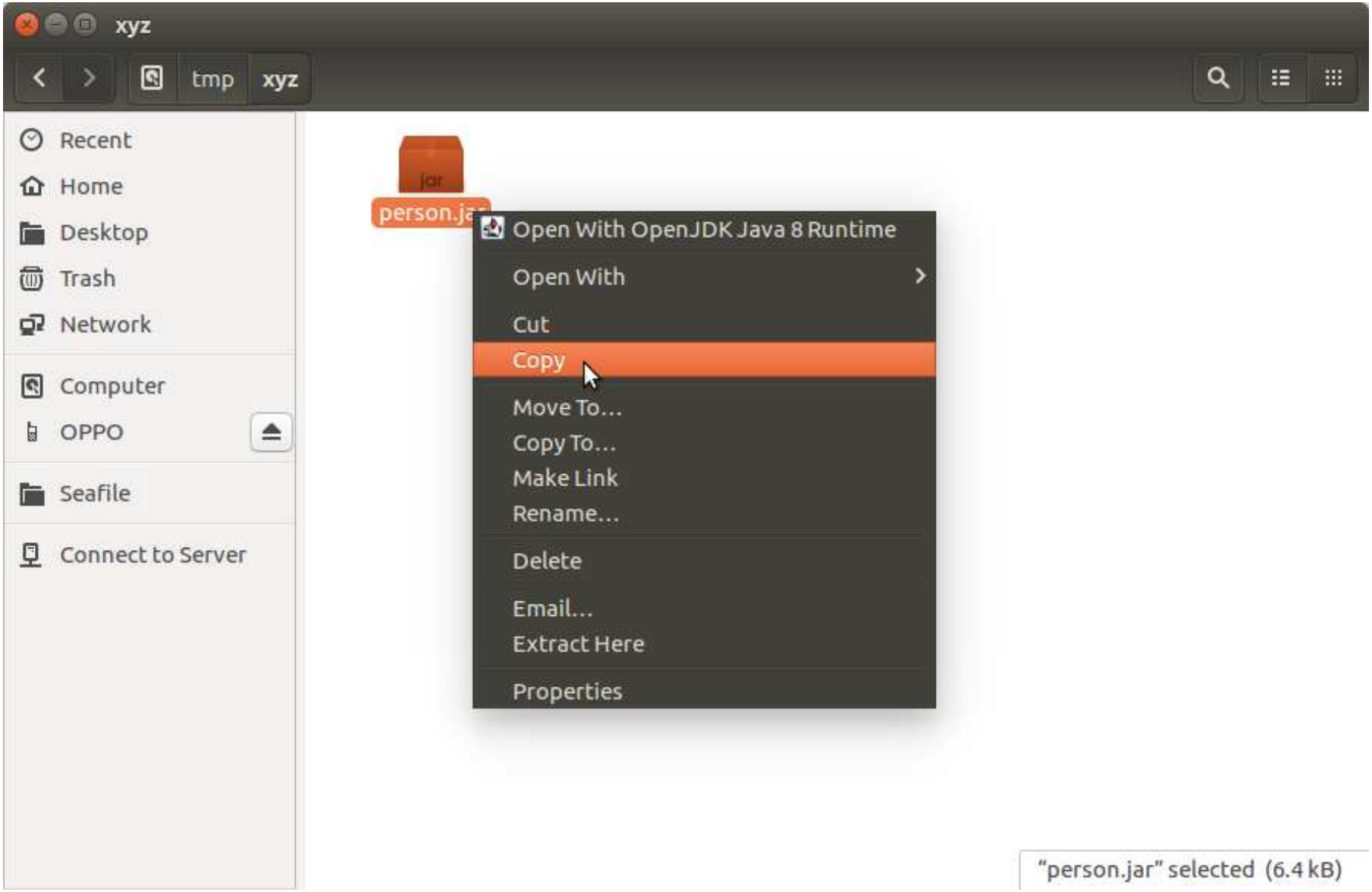
Using a Library



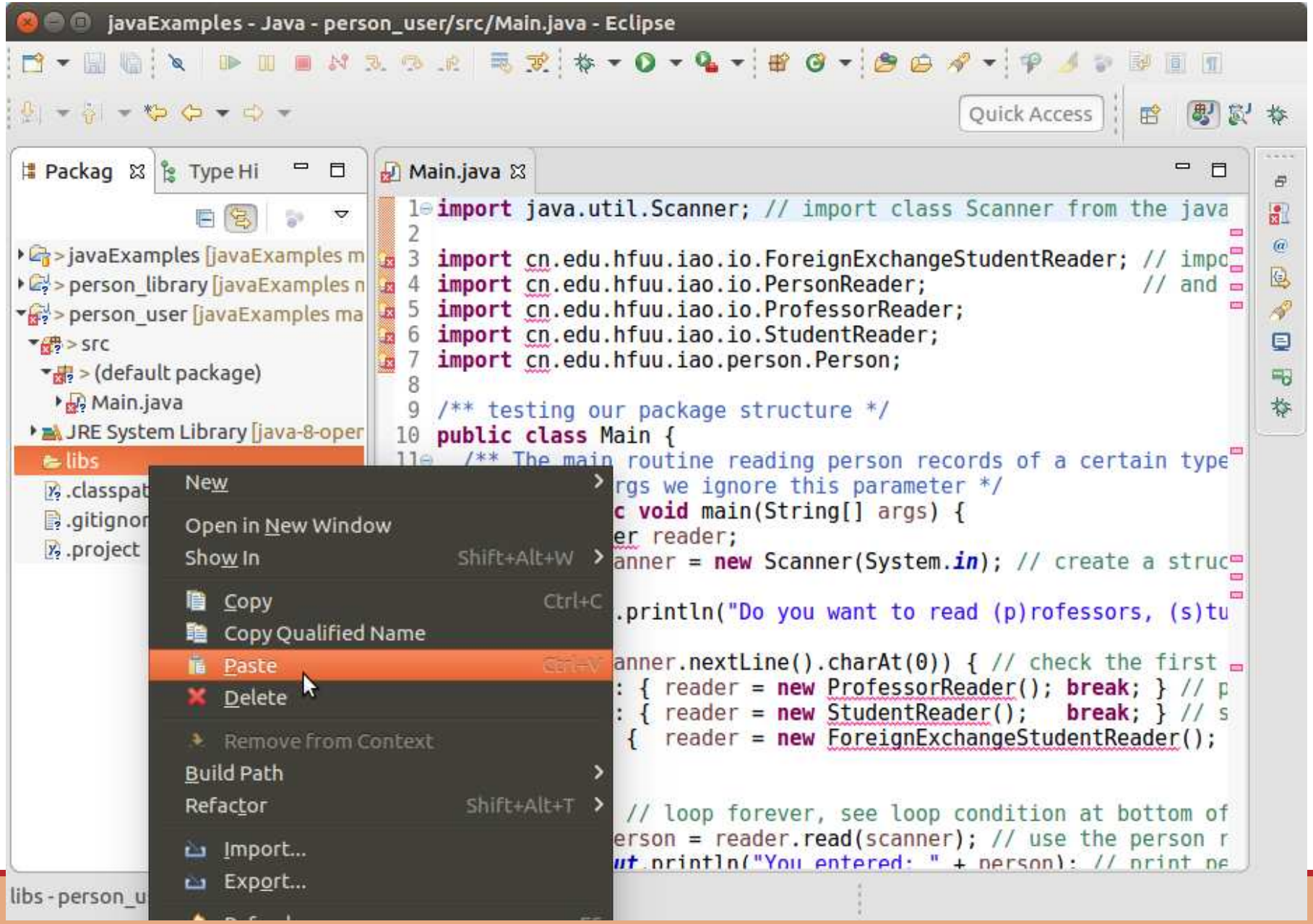
Using a Library

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

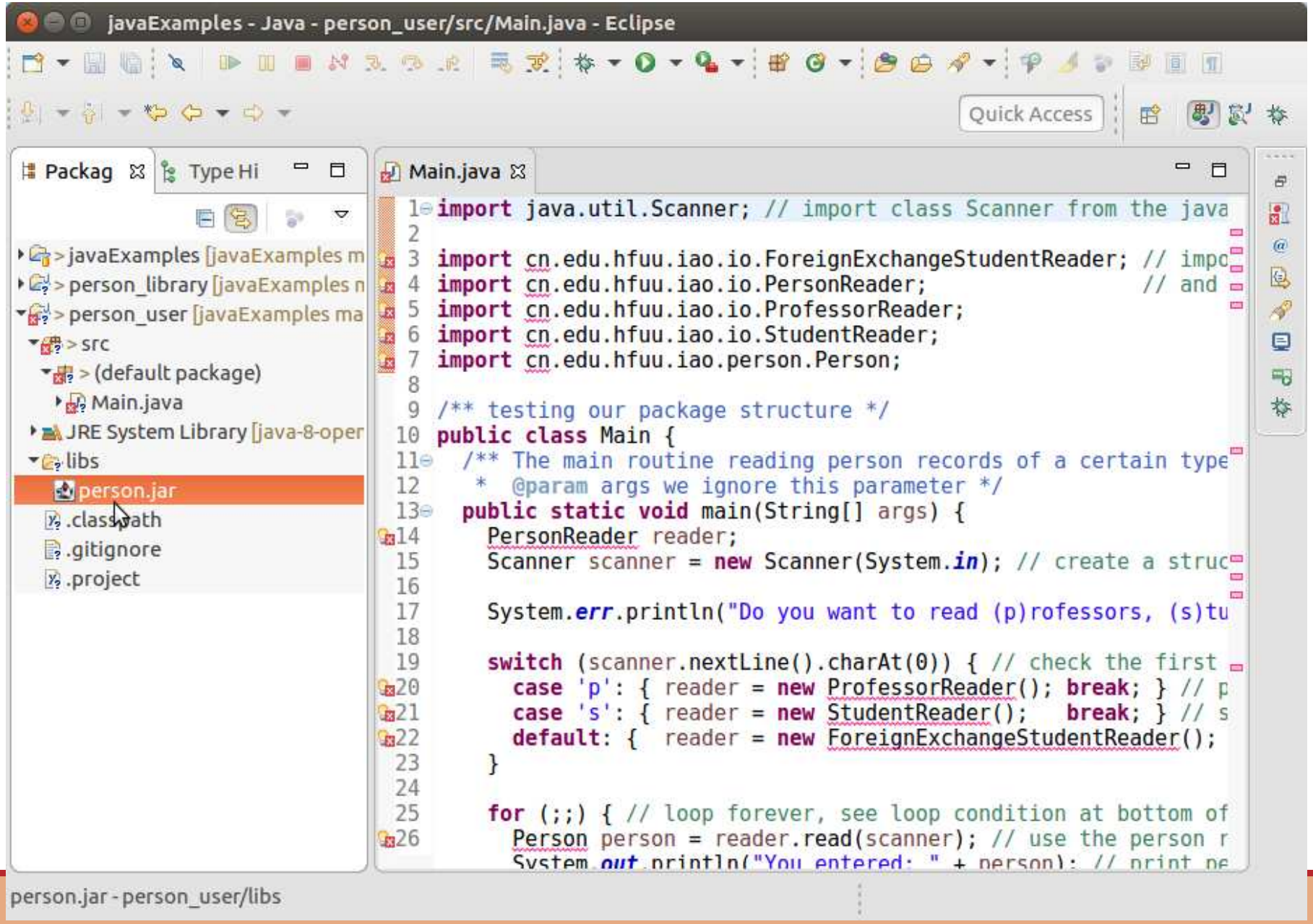
Using a Library



Using a Library



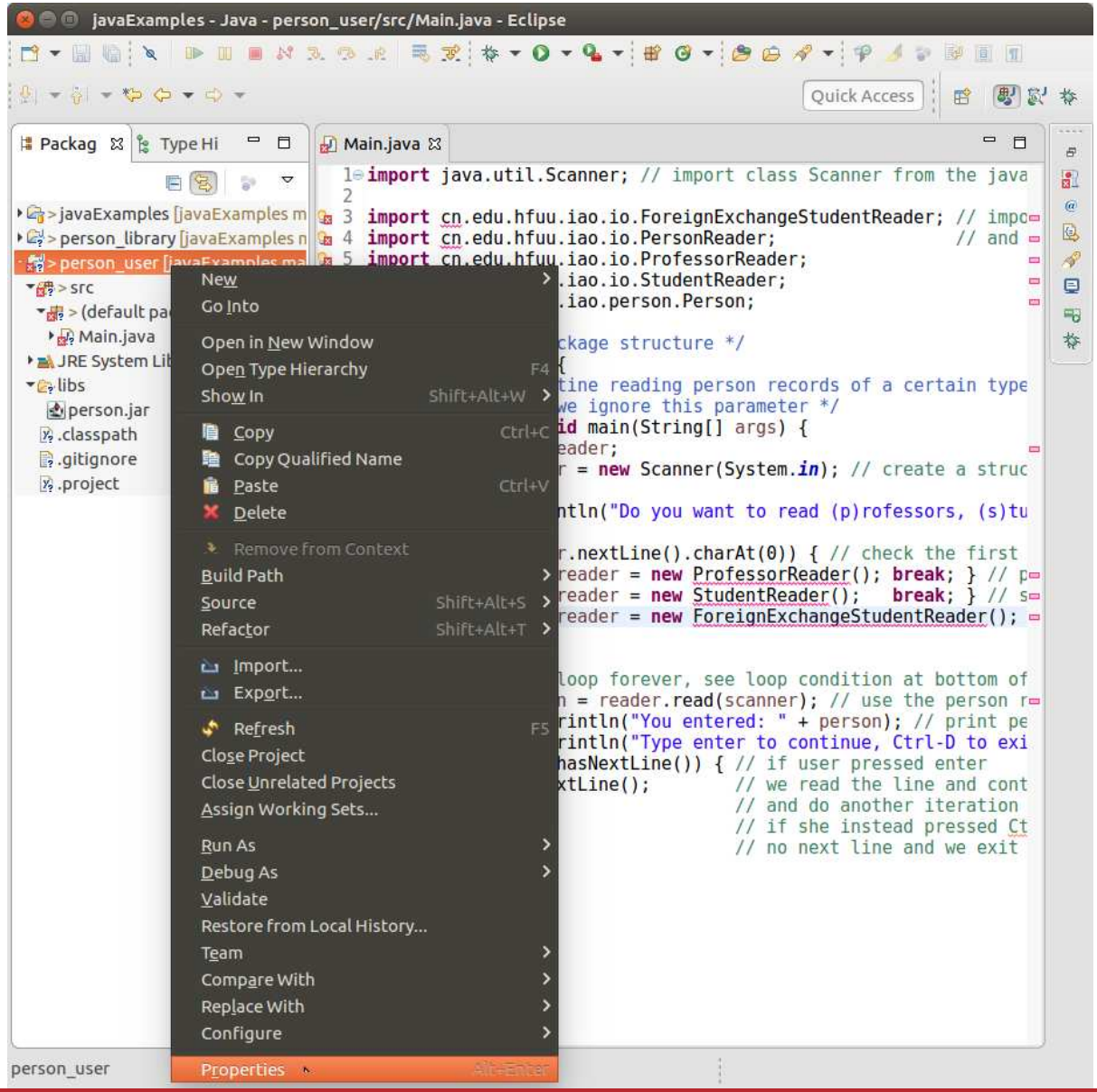
Using a Library



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`
- 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`

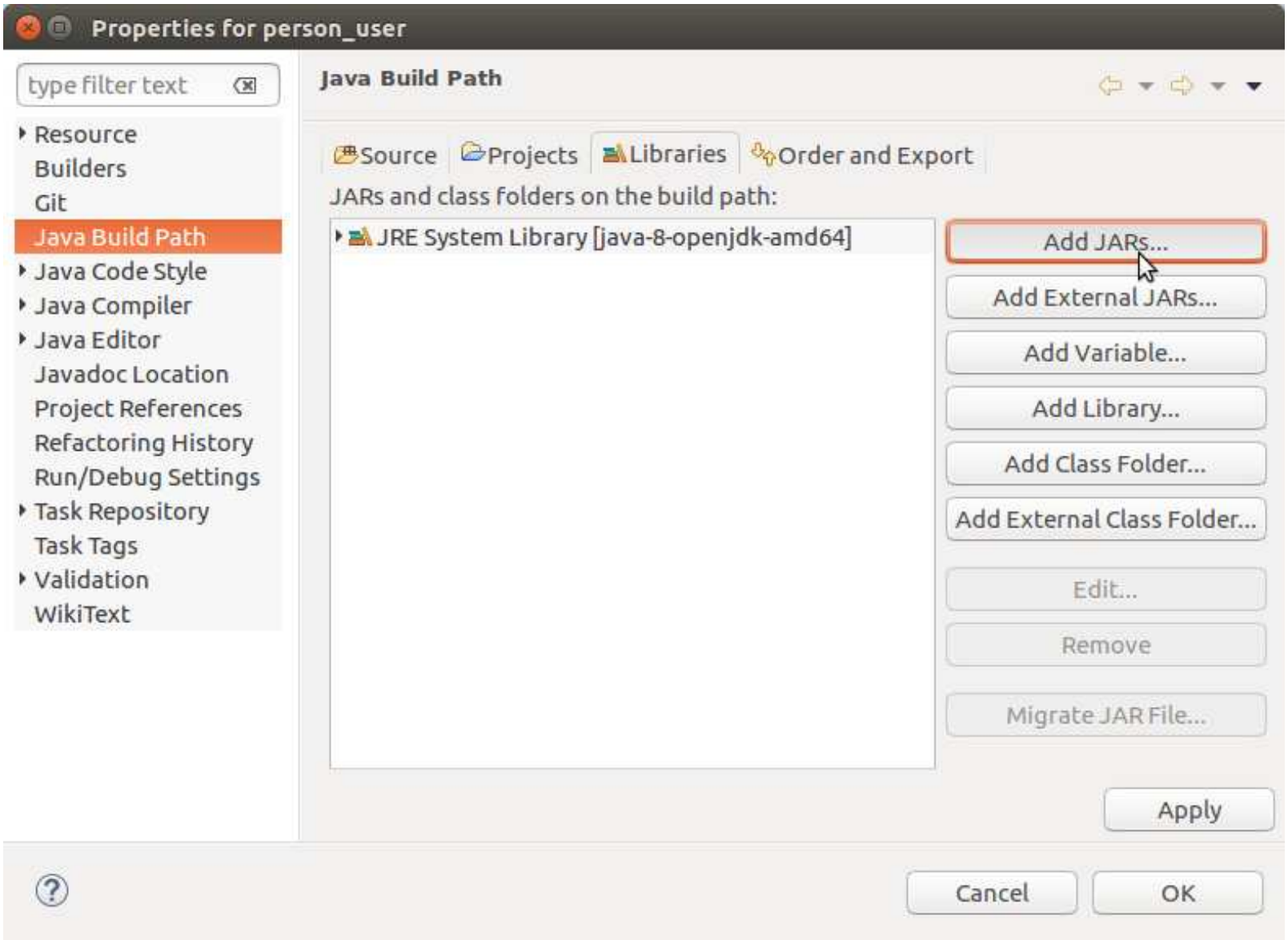
Using a Library



Using a Library

- 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...`

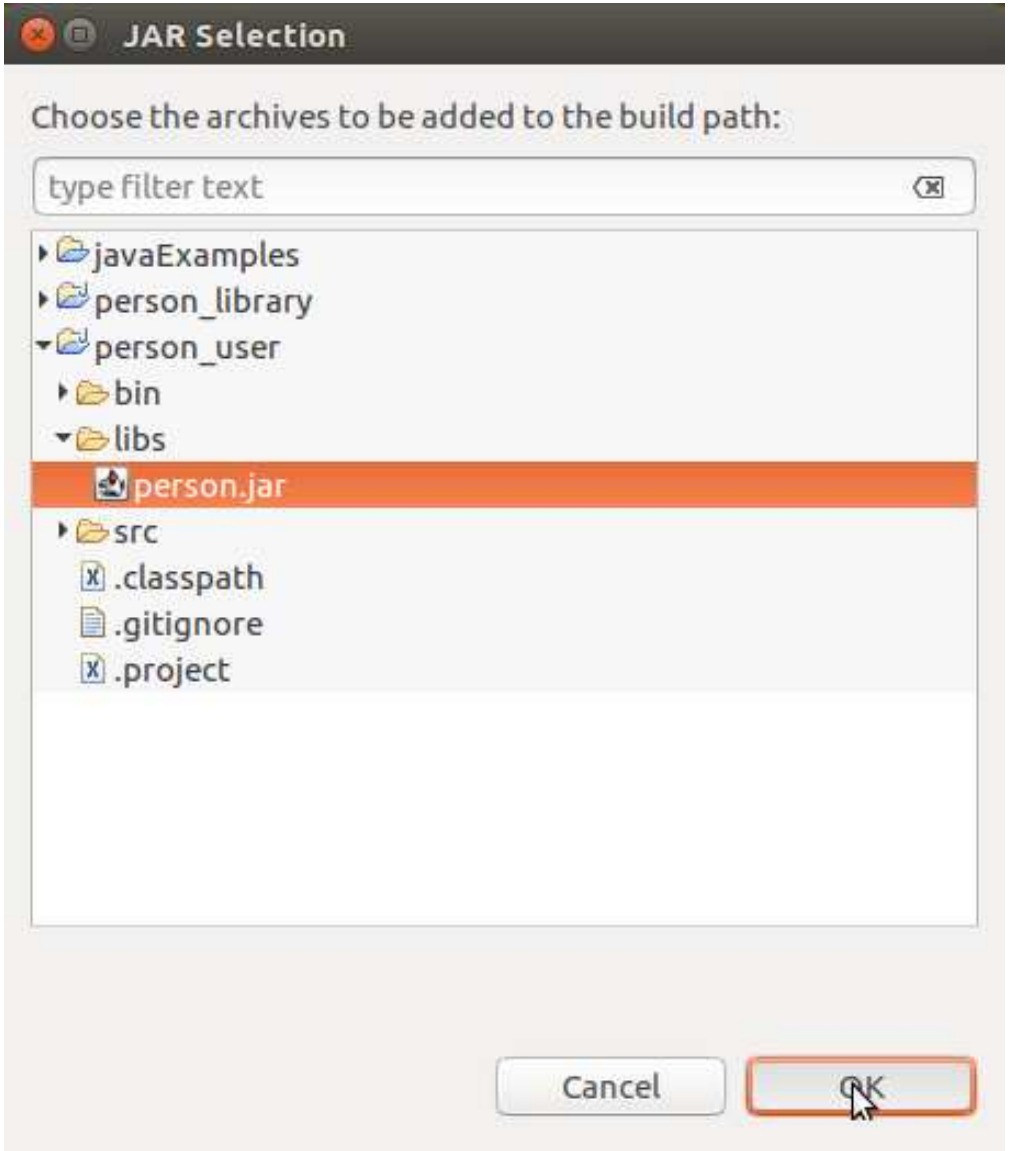
Using a Library



Using a Library

- 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 `OK`

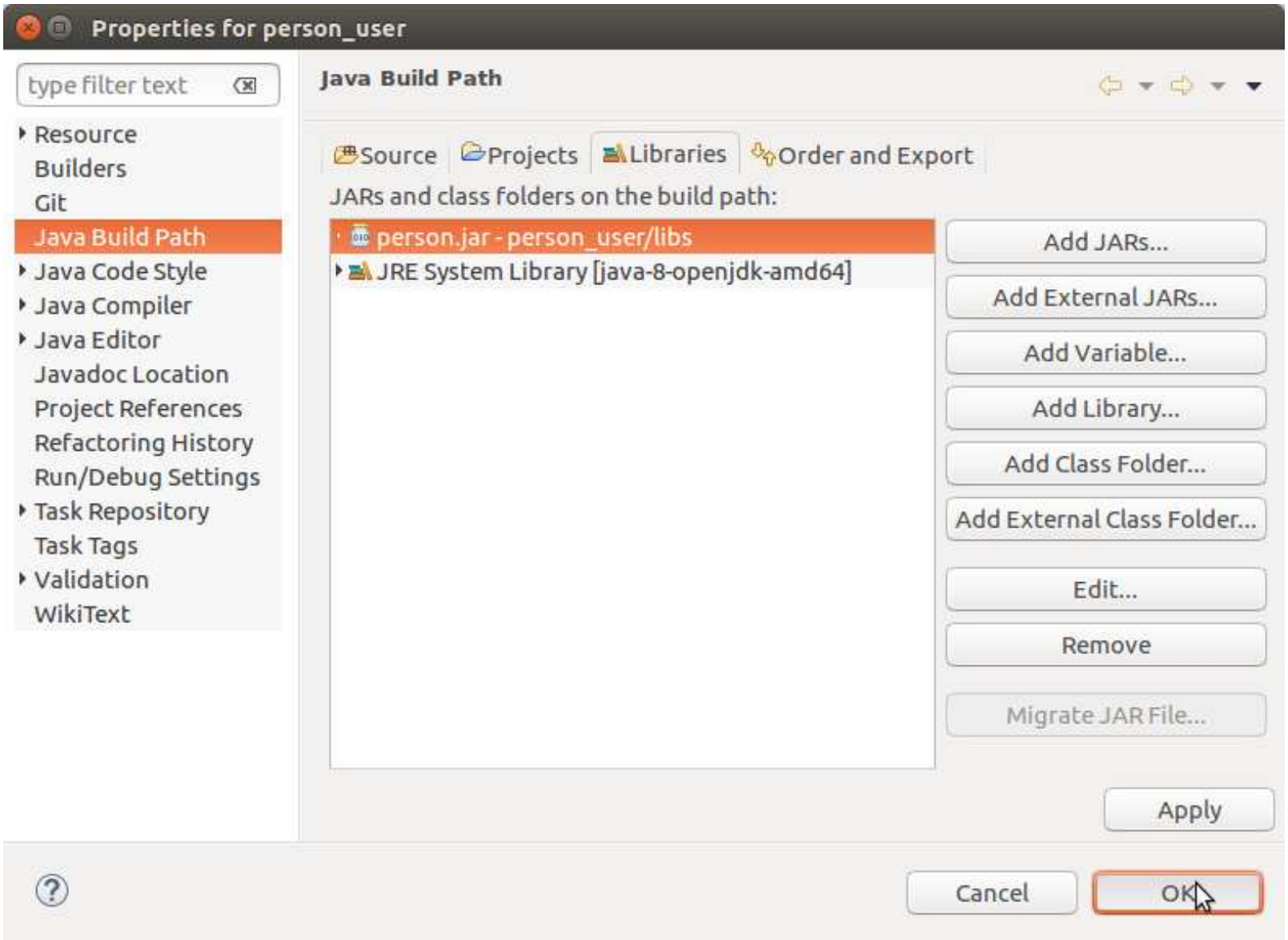
Using a Library



Using a Library

- 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 `OK`
- We click `OK`

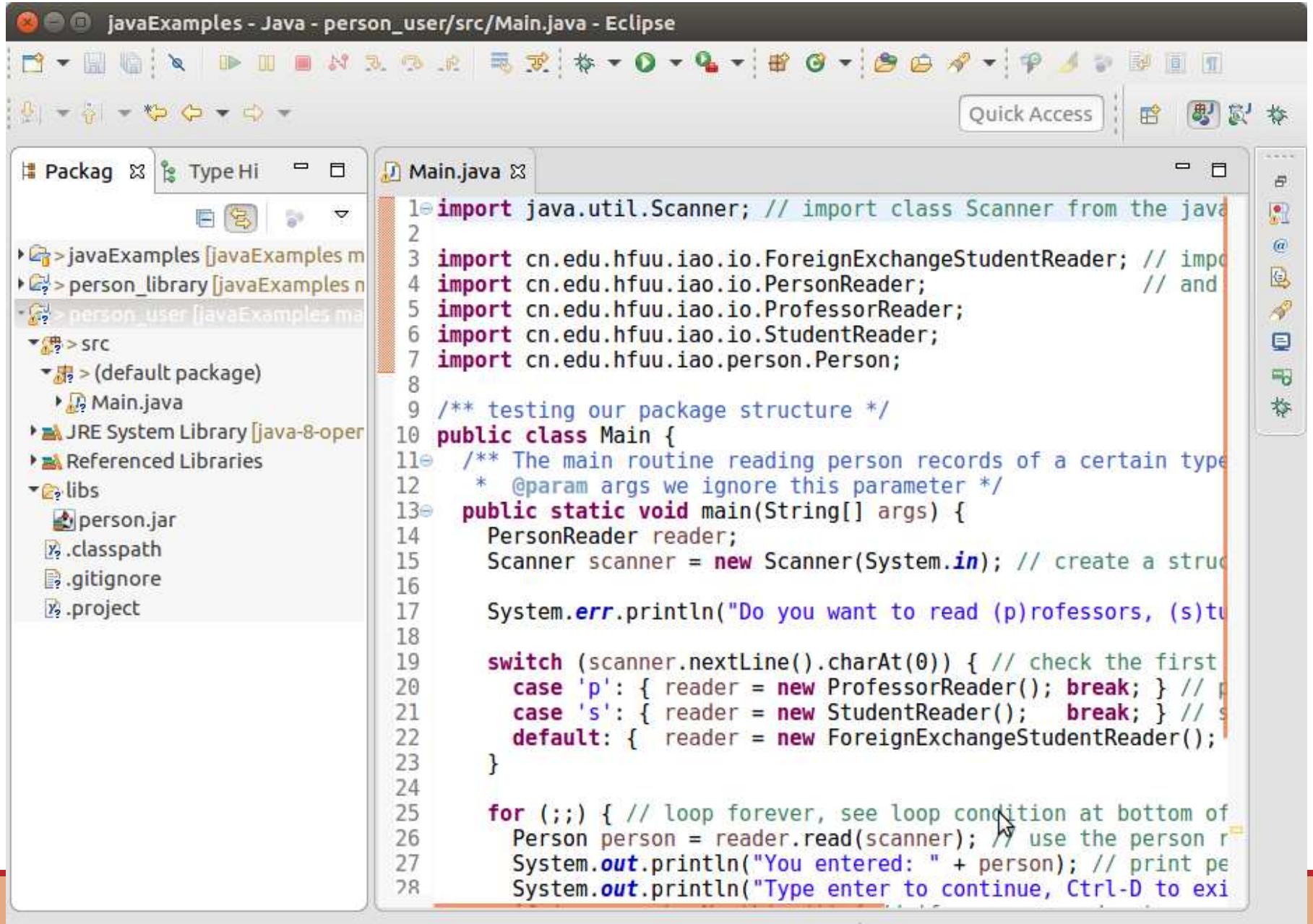
Using a Library



Using a Library

- 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 `OK`
- We click `OK`
- And our project compiles, the compiler errors disappear.

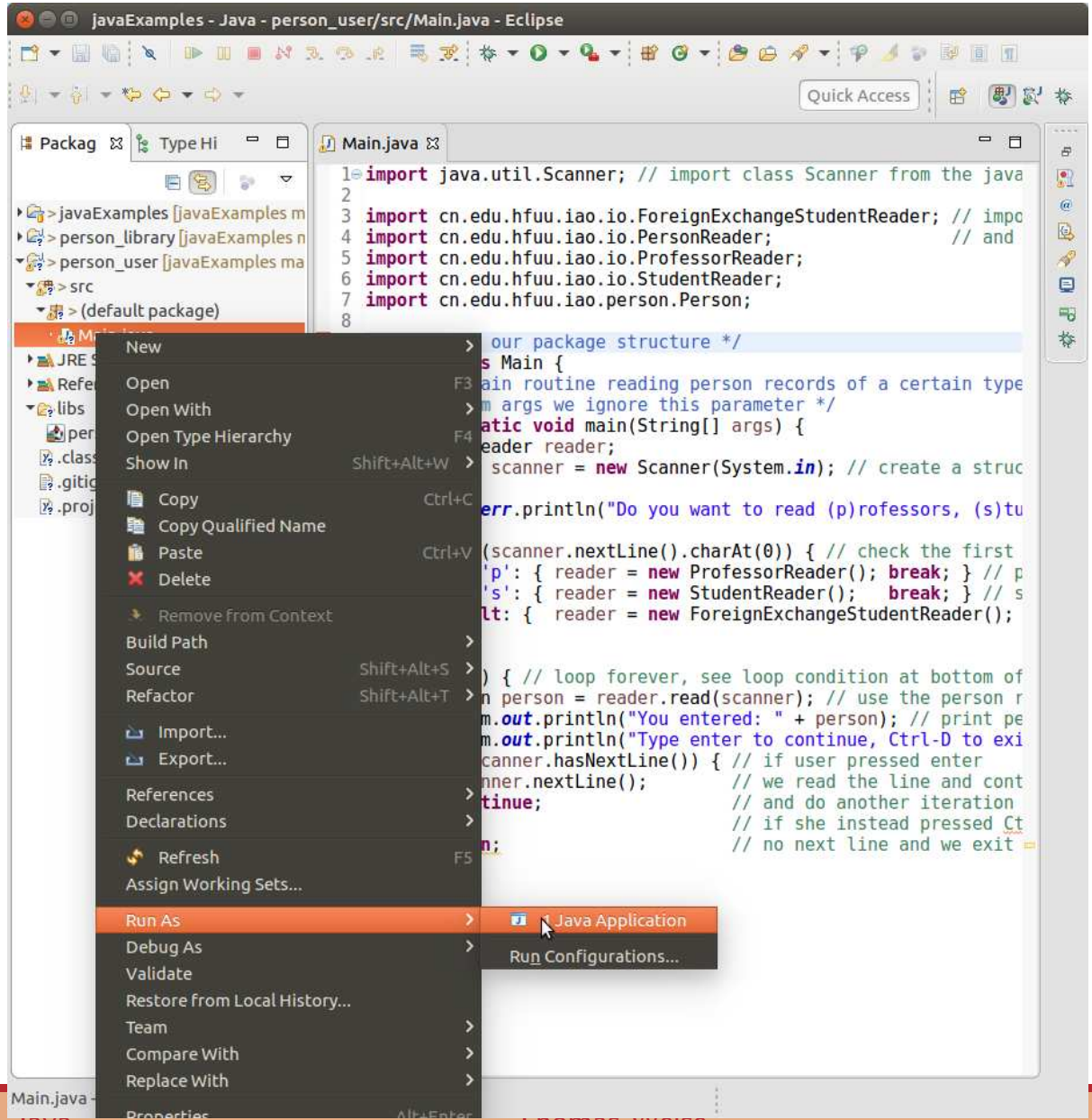
Using a Library



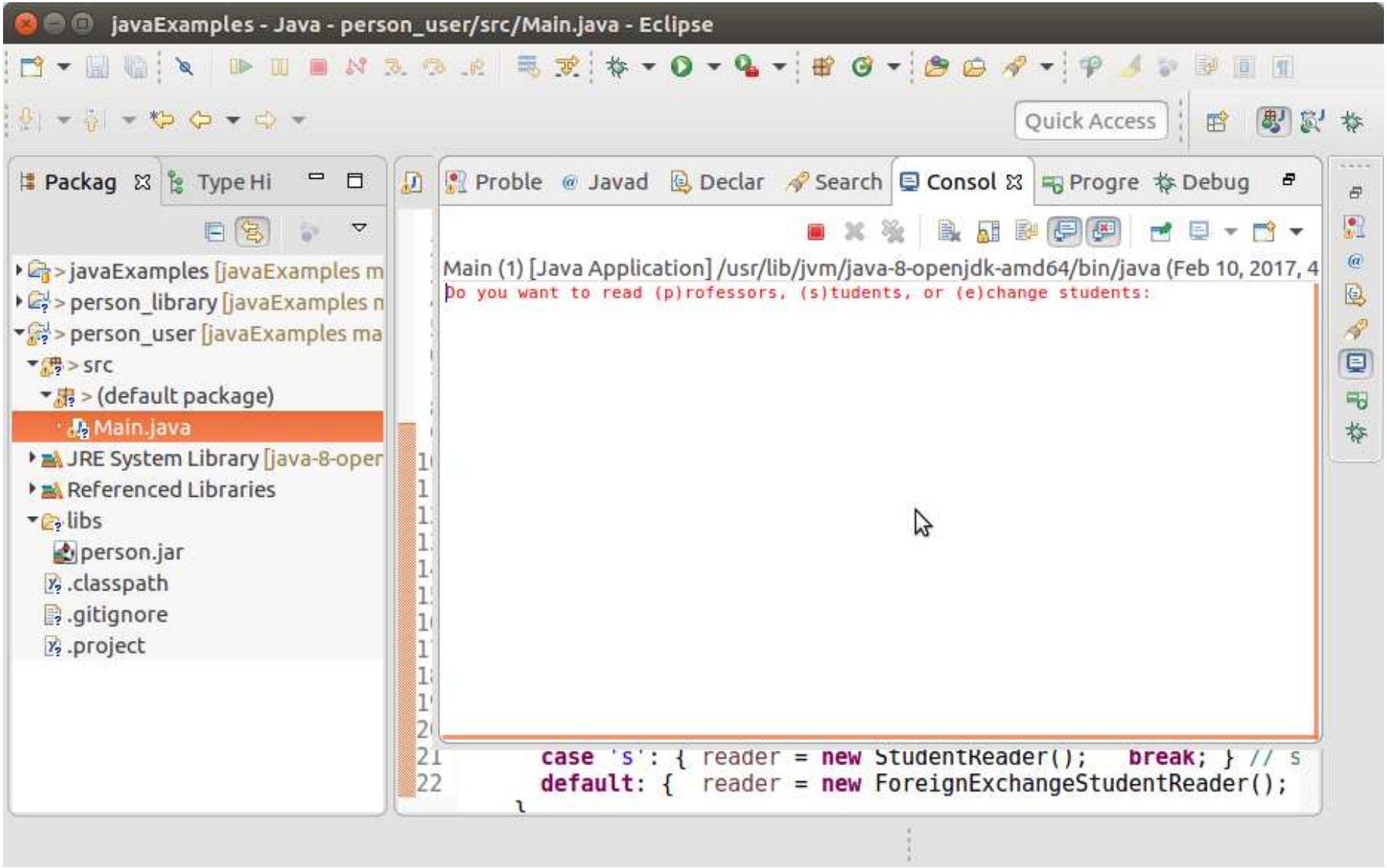
Using a Library

- 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 `OK`
- We click `OK`
- And our project compiles, the compiler errors disappear.
- We can now run it, and it will run beautifully

Using a Library



Using a Library



Using Libraries outside Eclipse

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

Using Libraries outside Eclipse

- 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

Using Libraries outside Eclipse

- 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

Using Libraries outside Eclipse

- 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)

Using Libraries outside Eclipse

- 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):
  1. open your terminal, `cd` into `src` folder

Using Libraries outside Eclipse

- 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):
  1. open your terminal, `cd` into `src` folder
  2. type in `javac -d ../bin -cp .:../libs/person.jar Main.java` and press return

Using Libraries outside Eclipse

- 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):
  1. open your terminal, `cd` into `src` folder
  2. type in `javac -d ../bin -cp .:../libs/person.jar Main.java` and press return
  3. 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

Using Libraries outside Eclipse

- 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)

Using Libraries outside Eclipse

- 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):
  1. open your terminal, `cd` into `bin` folder

Using Libraries outside Eclipse

- 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):
  1. open your terminal, `cd` into `bin` folder
  2. type in `java -cp .:../libs/person.jar Main` and press return

Using Libraries outside Eclipse

- 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):
  1. open your terminal, `cd` into `bin` folder
  2. type in `java -cp .:../libs/person.jar Main` and press return
  3. 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 JARs

- Executable `jar` archives are just the same as normal `jar` archives

Executable JARs

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

Executable JARs

- 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*

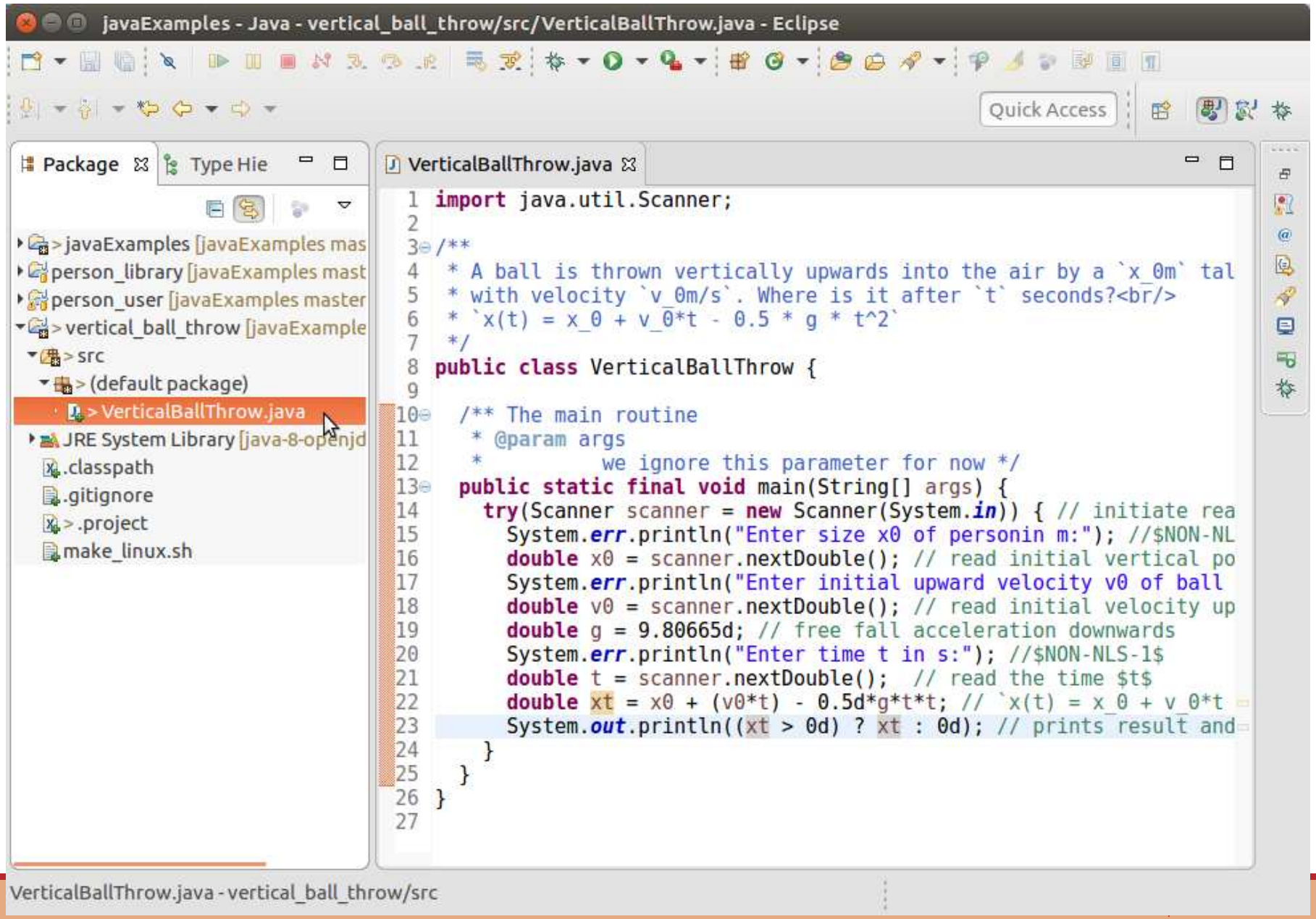
Listing: Vertical Ball Throw with Console I/O

```
import java.util.Scanner;

/**
 * A ball is thrown vertically upwards into the air by a x0m tall person
 * with velocity v0m/s. Where is it after t seconds?<br/>
 * x(t) = x0 + v0 * t - 0.5 * g * t^2
 */
public class VerticalBallThrow {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  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 x0 of person in m:"); //$NON-NLS-1$
      double x0 = scanner.nextDouble(); // read initial vertical position x0
      System.err.println("Enter initial upward velocity v0 of ball in m/s:"); //$NON-NLS-1$
      double v0 = scanner.nextDouble(); // read initial velocity upwards v0
      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) = x0 + v0 * t - 0.5 * g * t^2
      System.out.println((xt > 0d) ? xt : 0d); // prints result and makes sure the ball stops at
          ground
    }
  }
}
```

Making an Executable JAR Archive

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

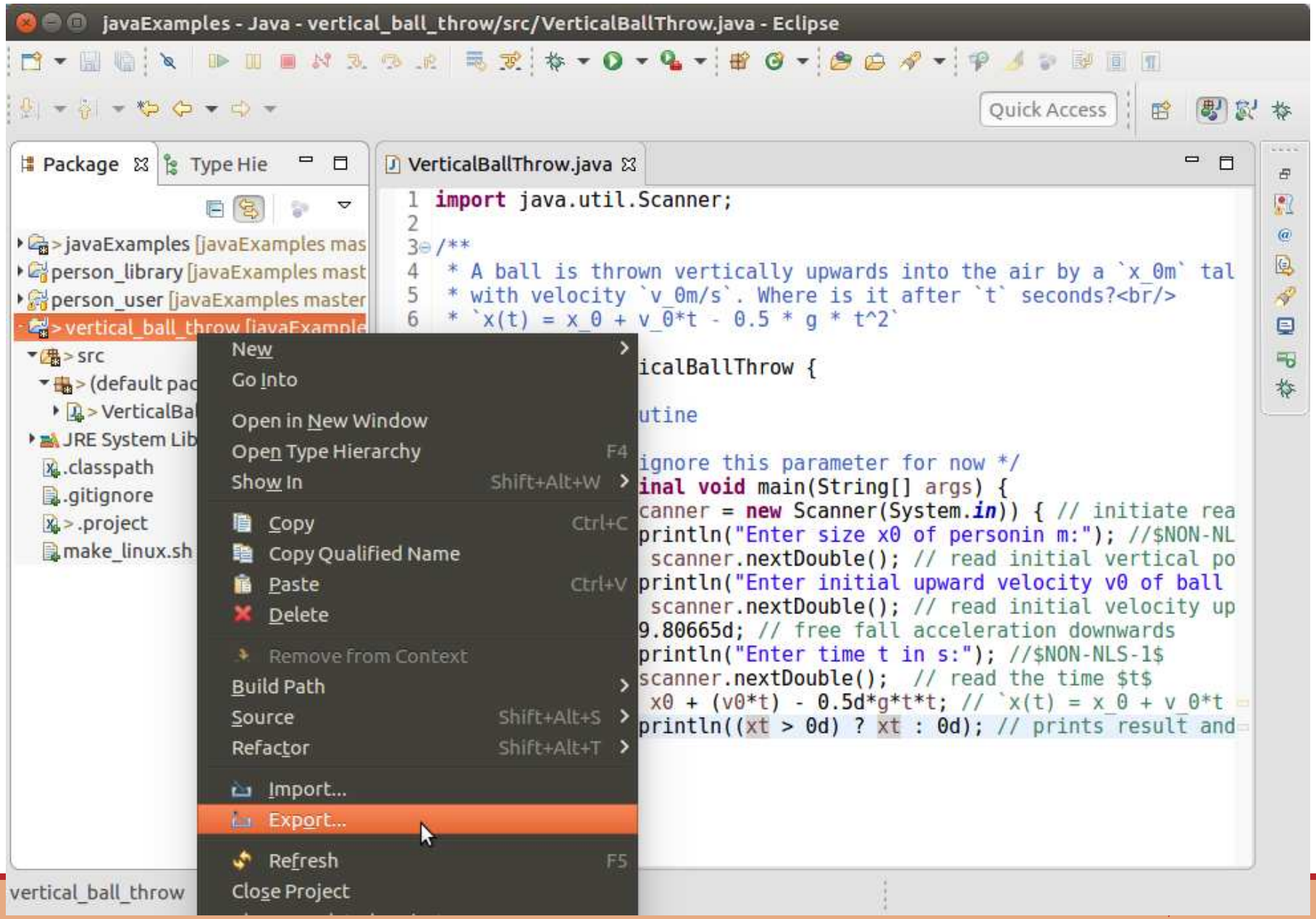
Making an Executable JAR Archive



Making an Executable JAR Archive

- 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...`

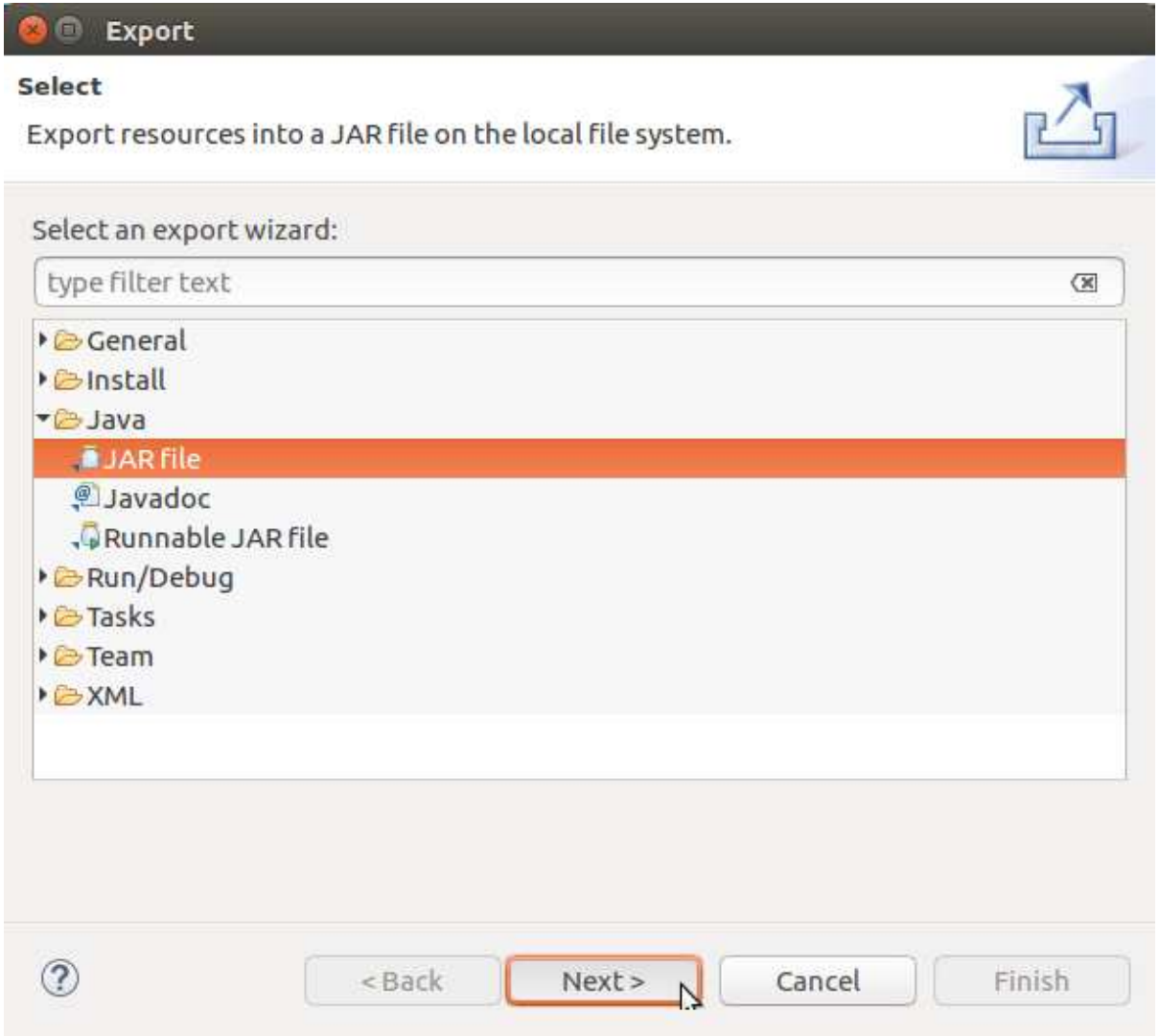
Making an Executable JAR Archive



Making an Executable JAR Archive

- 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`

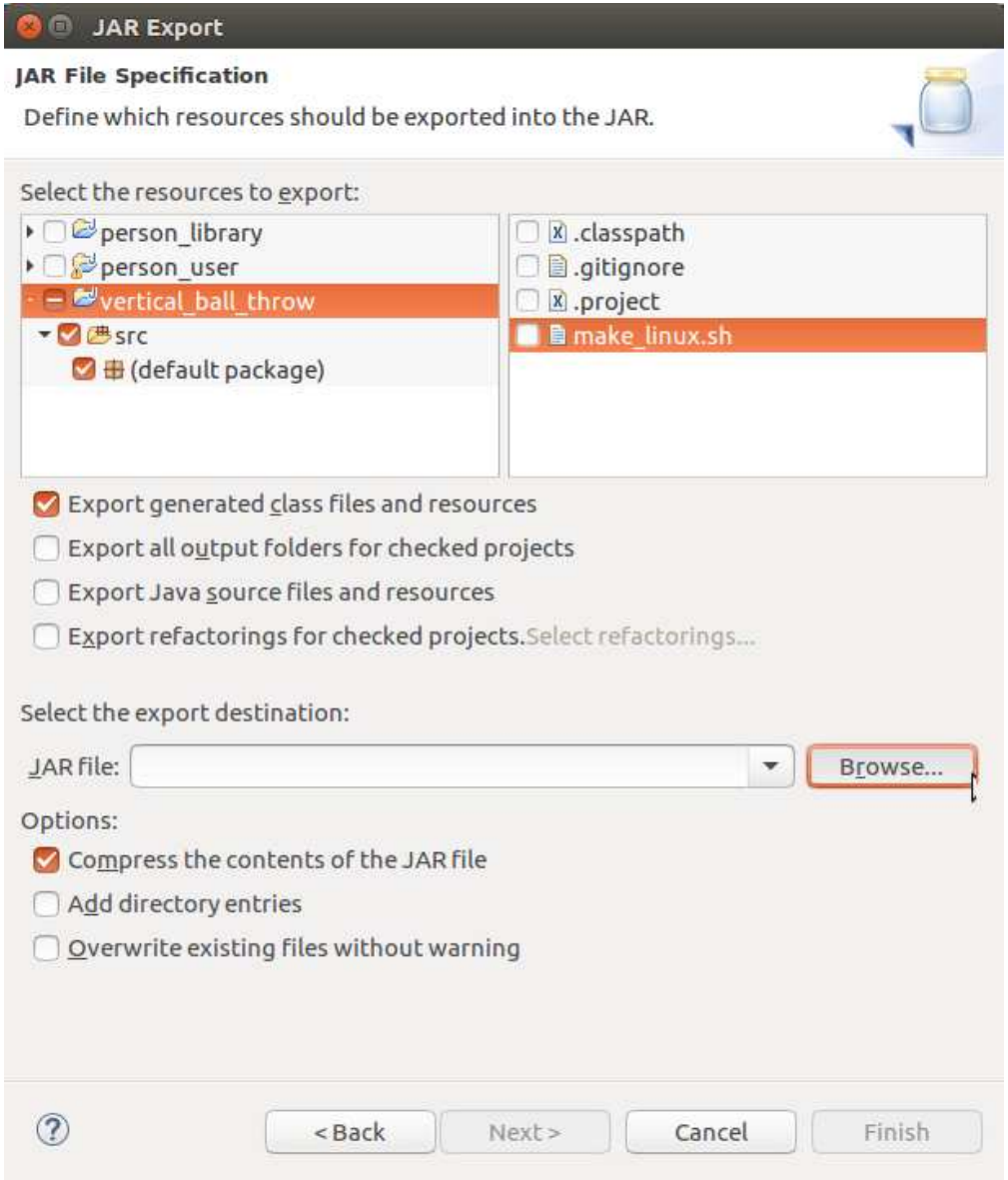
Making an Executable JAR Archive



Making an Executable JAR Archive

- 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

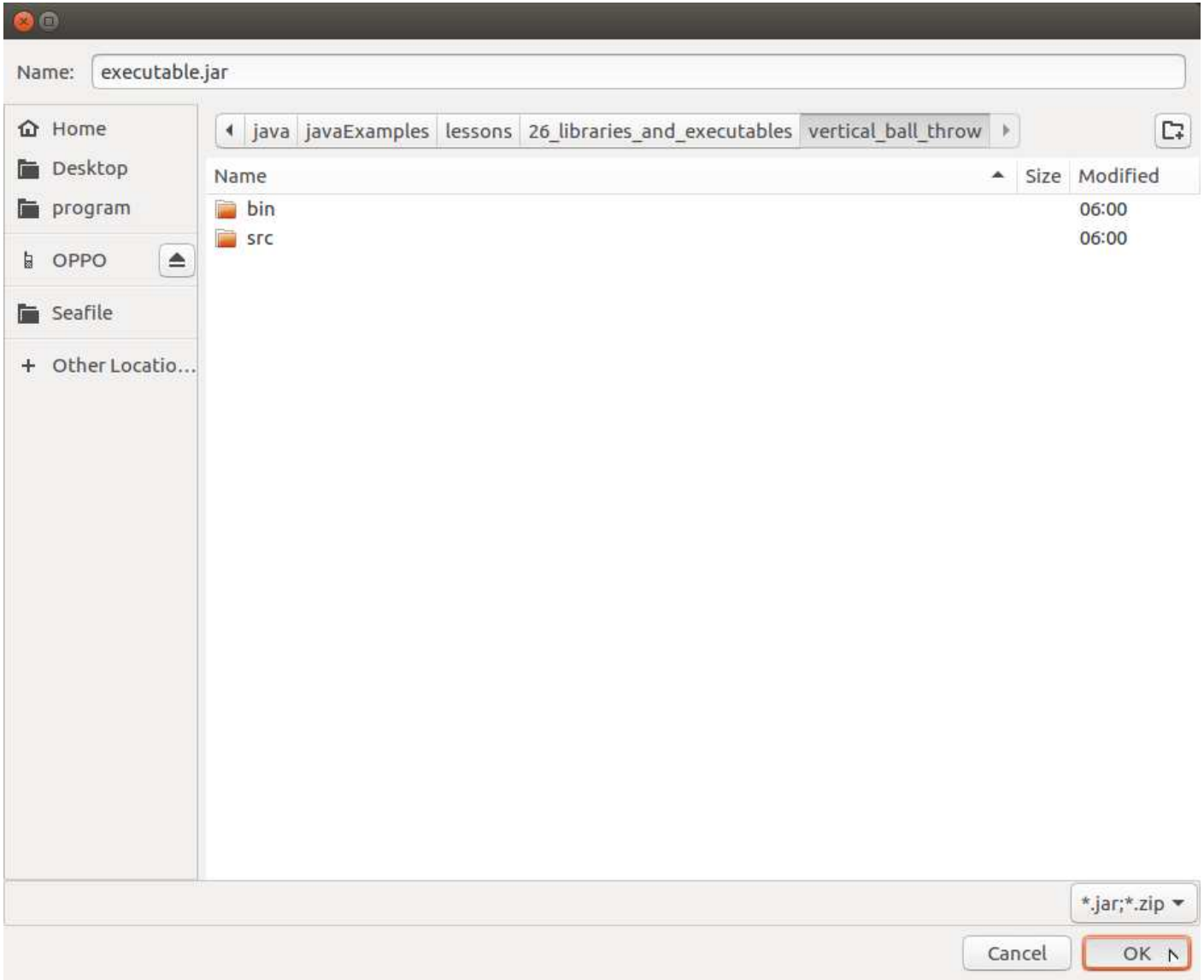
Making an Executable JAR Archive



Making an Executable JAR Archive

- 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`

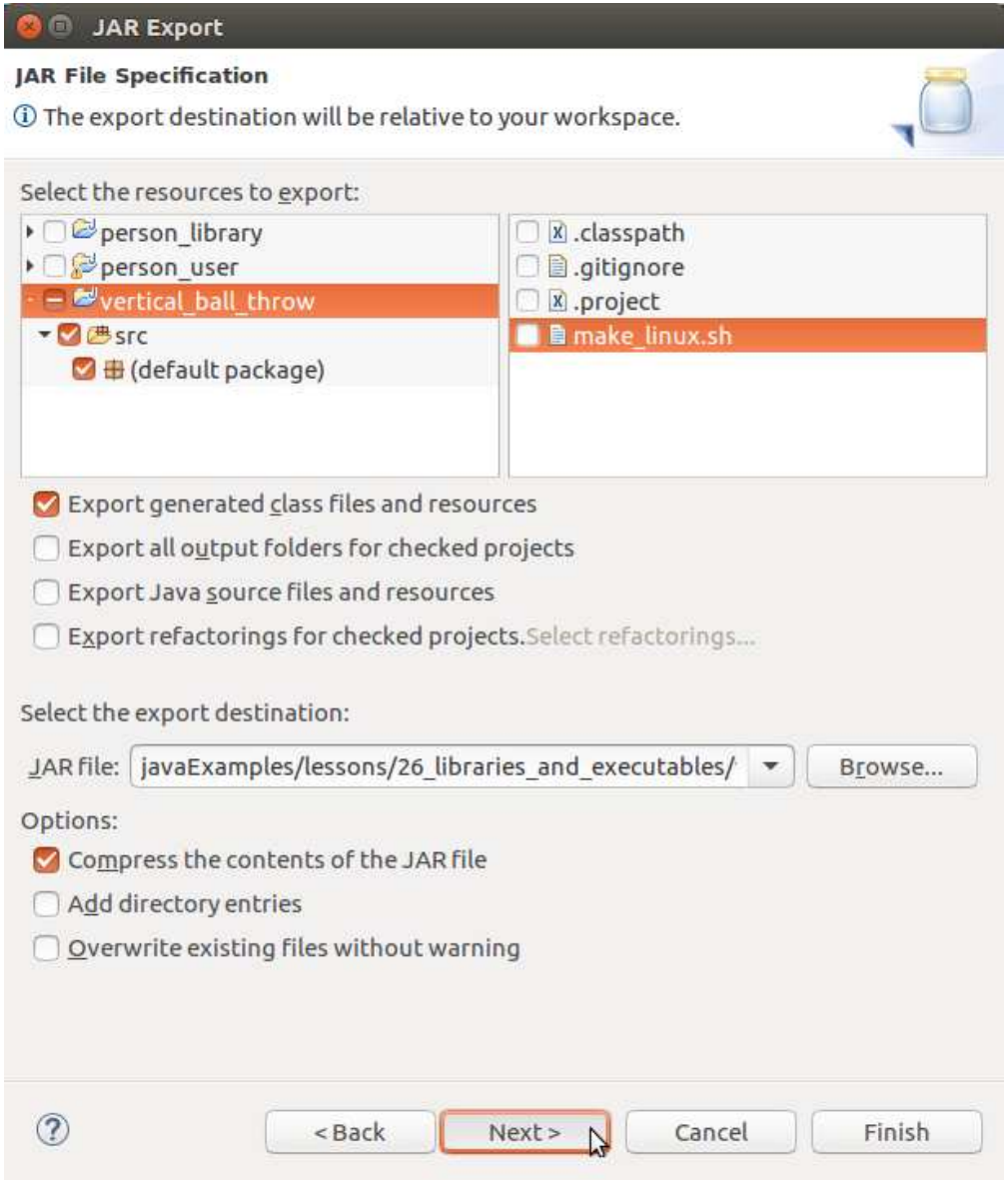
Making an Executable JAR Archive



Making an Executable JAR Archive

- 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`

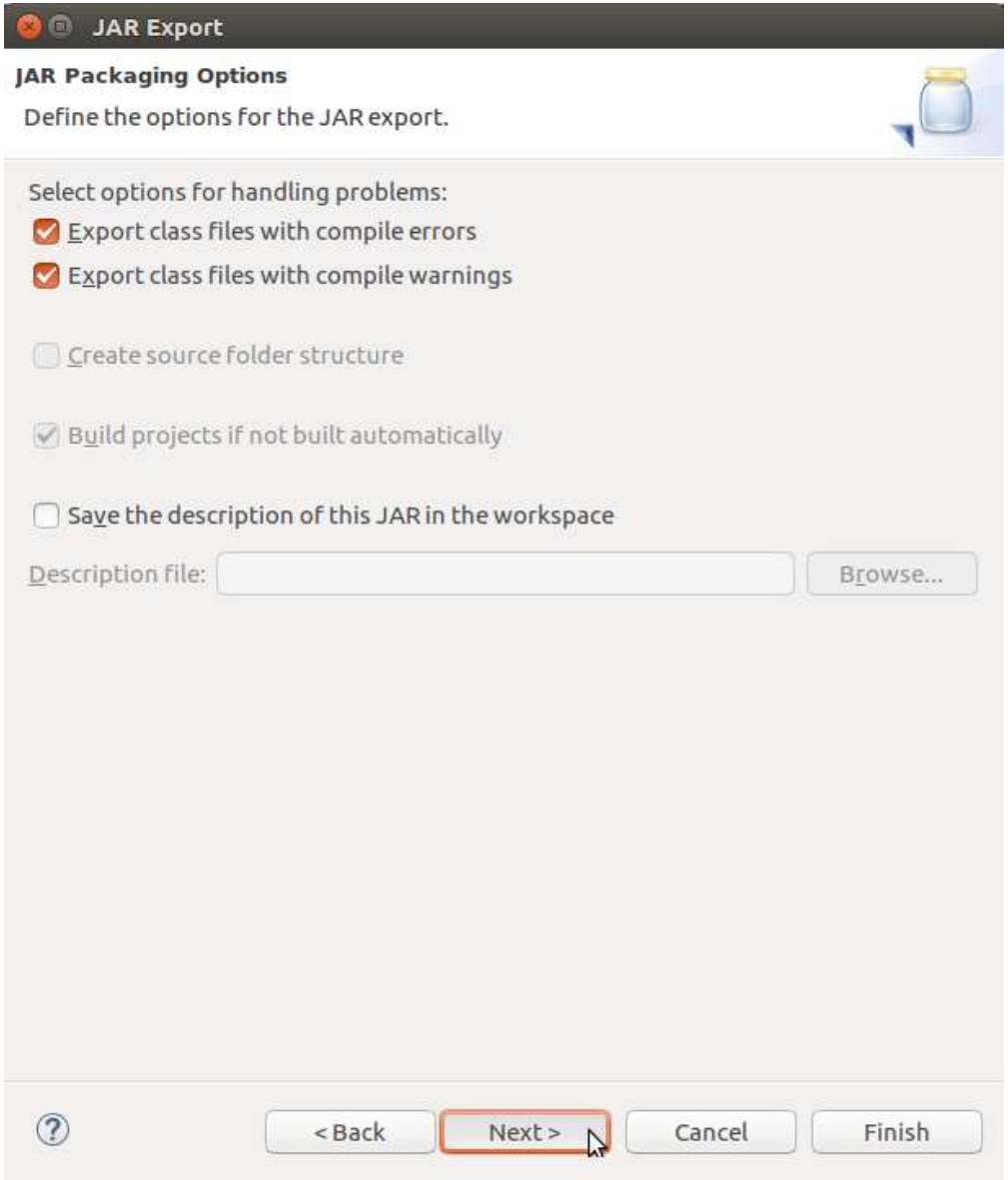
Making an Executable JAR Archive



Making an Executable JAR Archive

- 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`

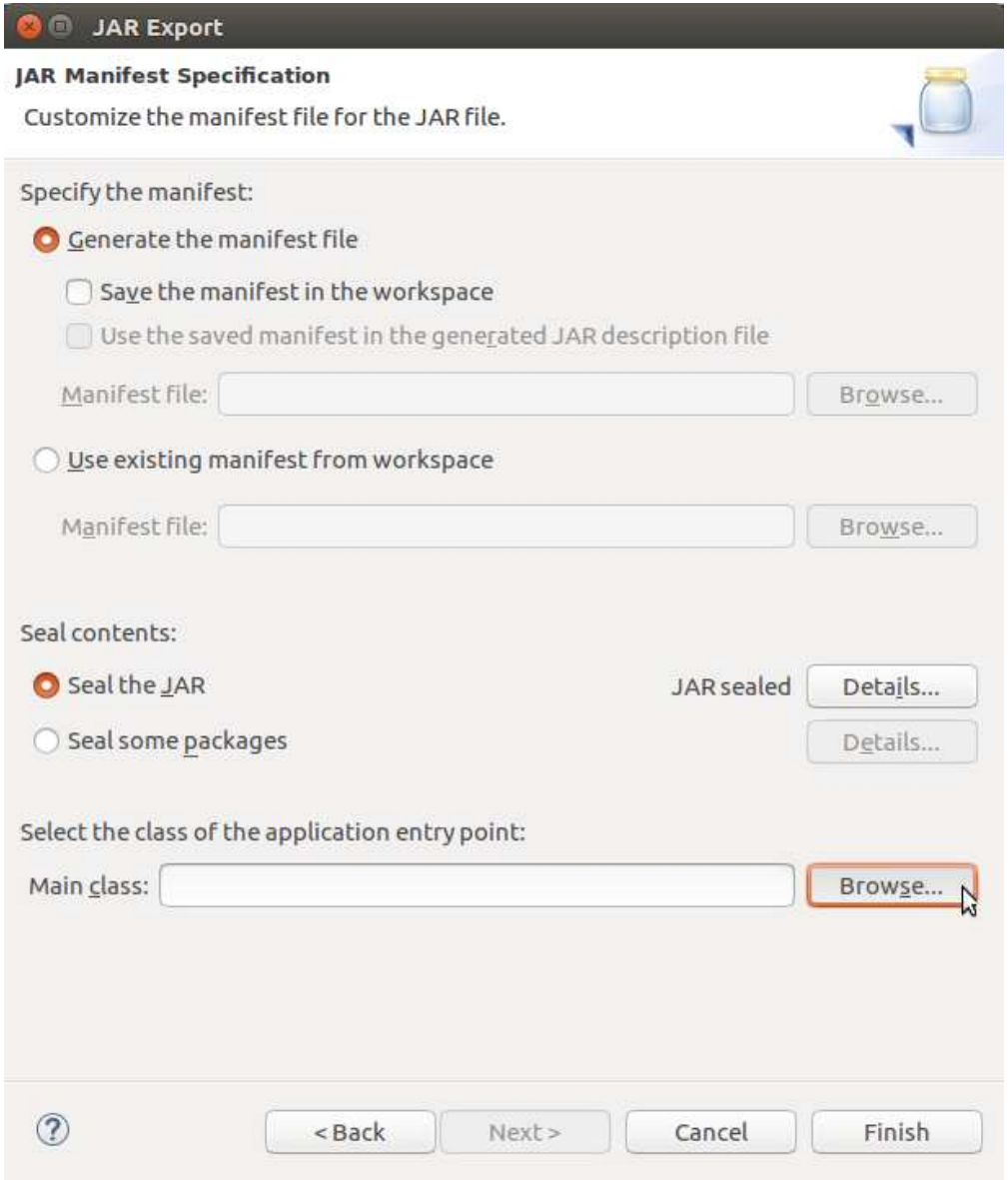
Making an Executable JAR Archive



Making an Executable JAR Archive

- 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 class` and therefore click `Browse...` after the corresponding input field

Making an Executable JAR Archive



Making an Executable JAR Archive

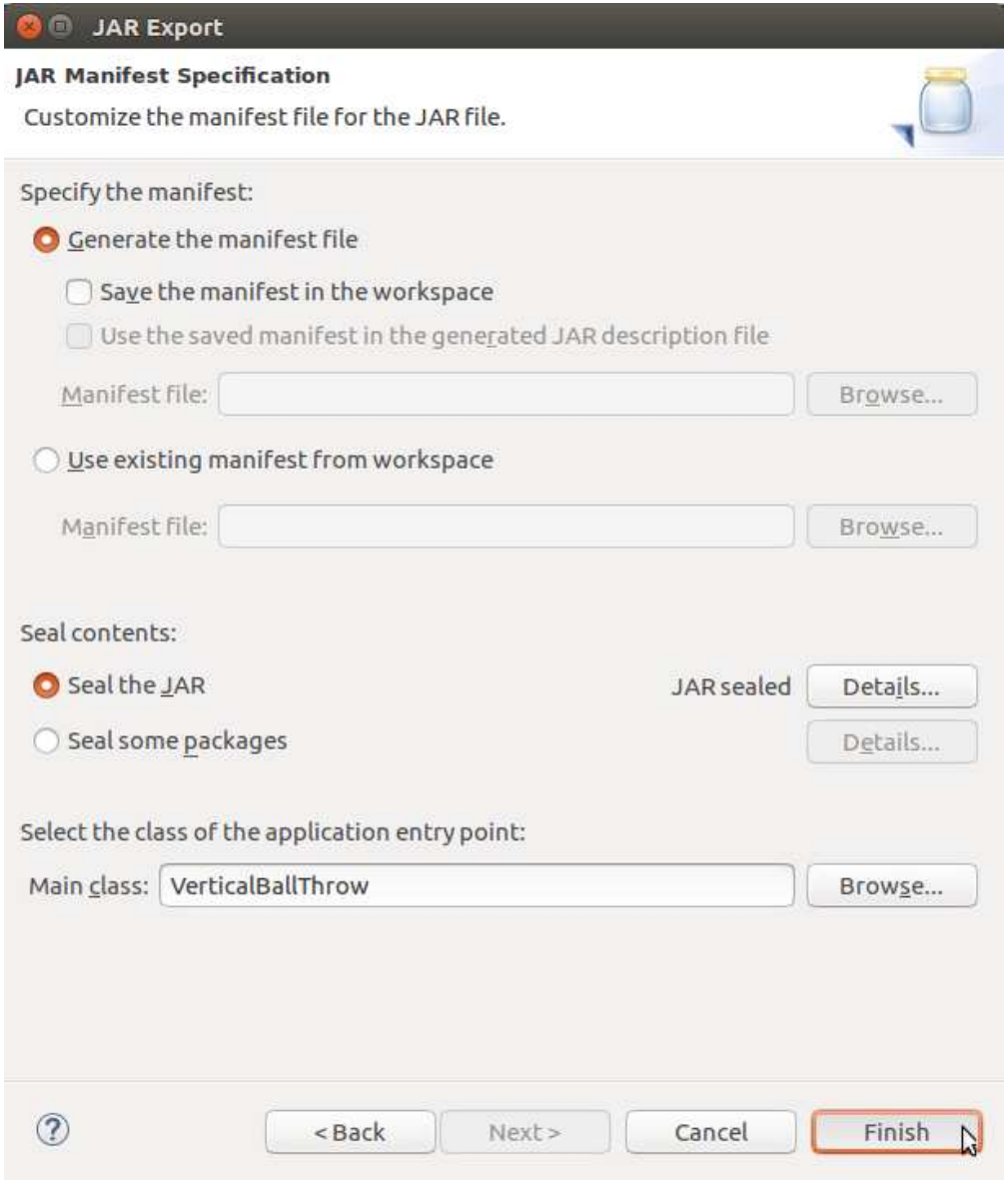
- We first make an Eclipse project called `vertical_throw` with our class `VerticalBallThrow` in it
- We again right-click on the project and then click `Export...`
- In the export wizzard, we choose `Java` and then `JAR file` and press `Next`
- In the next screen, we hit the `Browse` button
- We choose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

Making an Executable JAR Archive

Making an Executable JAR Archive

- 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 class` 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

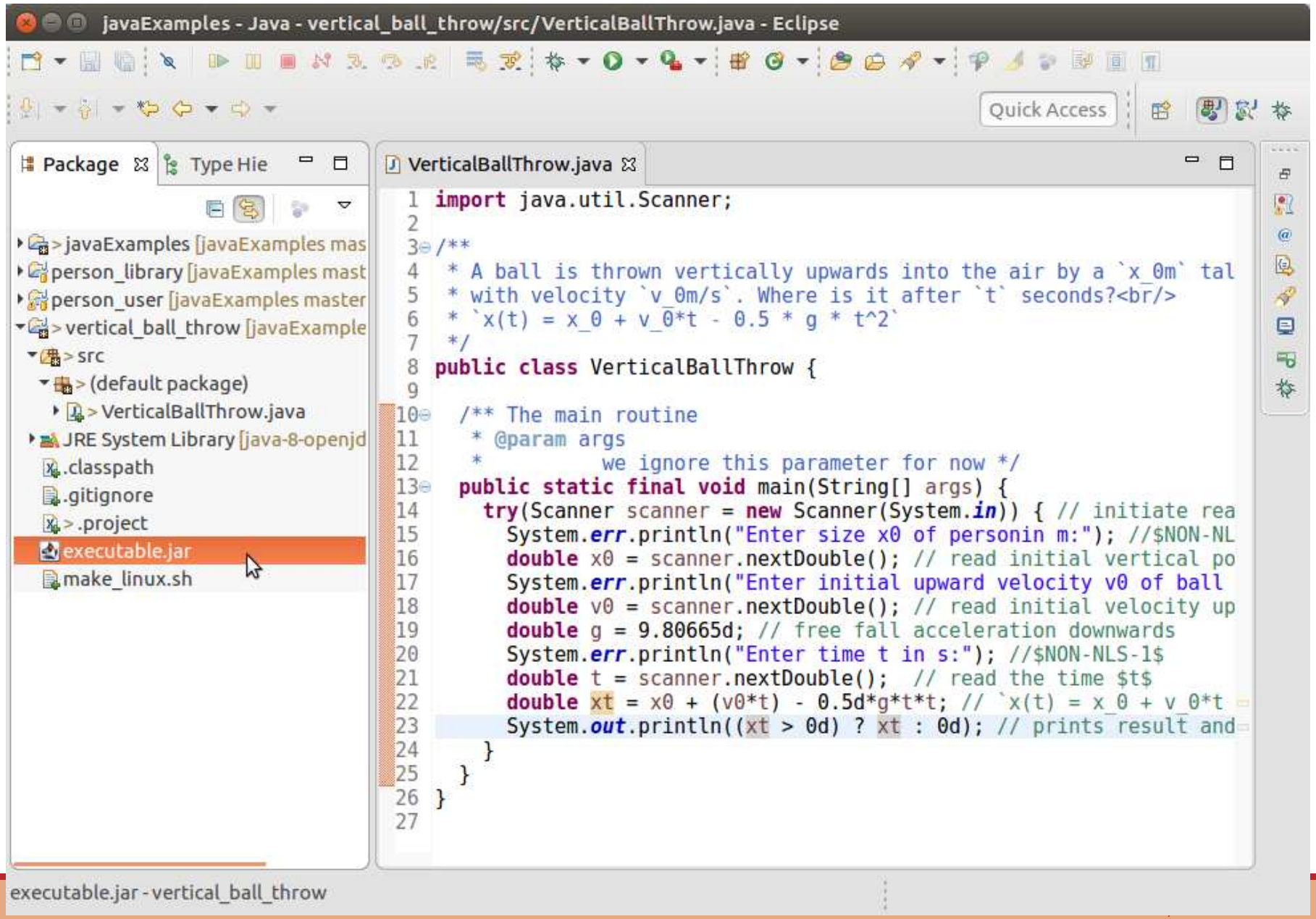
Making an Executable JAR Archive



Making an Executable JAR Archive

- 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 class` 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...

Making an Executable JAR Archive



Executing JARs

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

Executing JARs

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

Executing JARs

- `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

Executing JARs

- `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

Executing JARs

- `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!

Executing JARs

- `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

Executing JARs

- `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

Summary

- 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

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog