



SOFTWARE DEVELOPMENT: ASSIGNMENT 1

DEPARTMENT OF COMPUTER SCIENCE

FEBRUARY 4, 2018

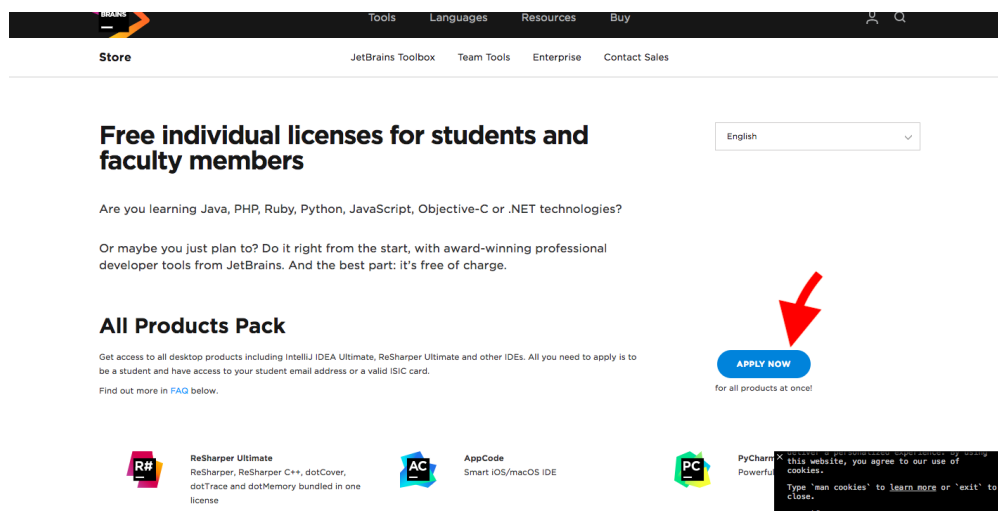
This is an individual assignment. We expect you to hand-in your code in a zip folder called *src.zip*, and your report as a latex pdf called *report.pdf*. Both should be uploaded on Absalon by **Friday, 9th of February at 3:00 PM** at the latest.

1 Exercise


1.1 Setting-up Rider

During this course you will use the Integrated Development Environment (IDE) called Rider. The first task is to install Rider on your computer and provide screenshots of the installation as proof in your report.

1. Start by visiting the site: <https://www.jetbrains.com/student/> and click on **Apply now**.



2. Fill it out



JetBrains Products for Learning

Apply with: UNIVERSITY EMAIL ADDRESS ISIC/ITC MEMBERSHIP OFFICIAL DOCUMENT

Status: ☒ I'm a student ☐ I'm a teacher

Name: First name Last name

Our software will be registered to your real name.

Email address:


Your valid university email address, e.g. john.smith@mit.edu.
I confirm that the email address provided above belongs to me.

Country / region: Denmark

☐ I have read and I accept the [JetBrains Privacy Policy](#)

[APPLY FOR FREE PRODUCTS](#)

3. Go to your email and confirm



JetBrains Products for Learning

JetBrains Educational Pack Confirmation

JetBrains Account [no_reply@jetbrains.com] Handler

Til: Sarah Maria Hyatt

1. februar 2018 10:45

• For at øge beskyttelsen af dine personlige oplysninger er noget indhold af meddelelsen blevet blokeret. Hvis du er sikker på, at denne meddelelse er fra en afsender, der er tillid til, og du vil genaktivere de blokerede funktioner, skal du [klik her](#).


Hi,

You've received this email because your email address was used for registering/updating a JetBrains Educational Pack.

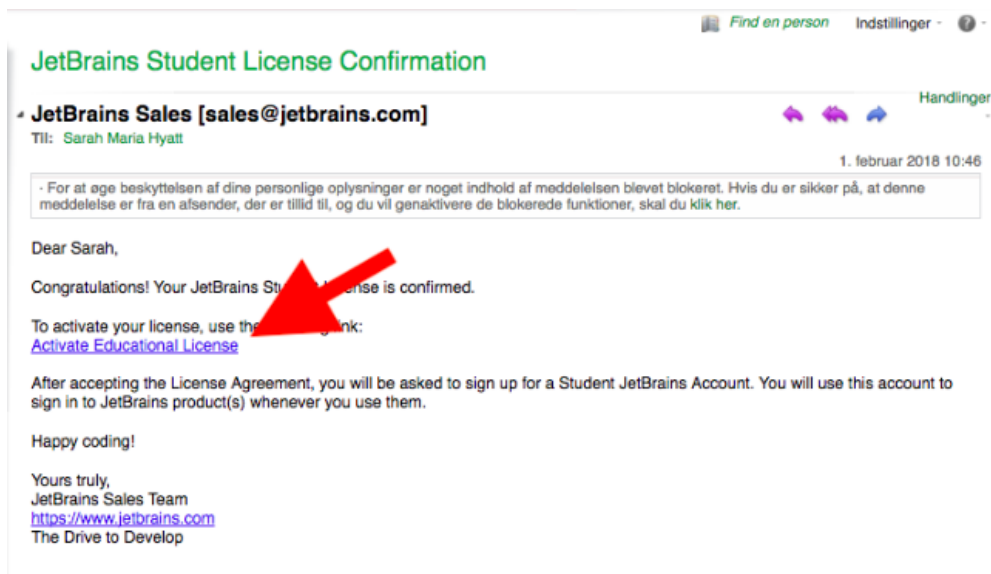
Please follow this link to confirm your intention:

[Confirm Request](#)

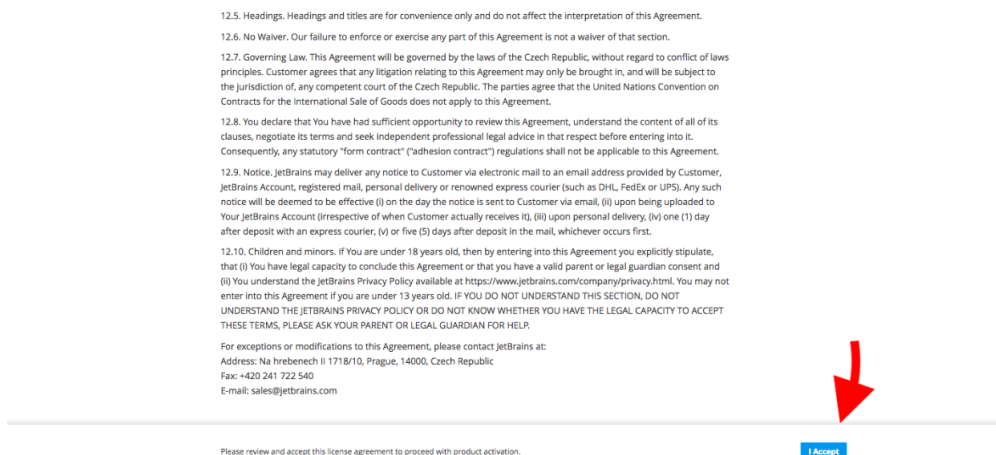
Yours truly,
JetBrains Team
<https://www.jetbrains.com>
The Drive to Develop



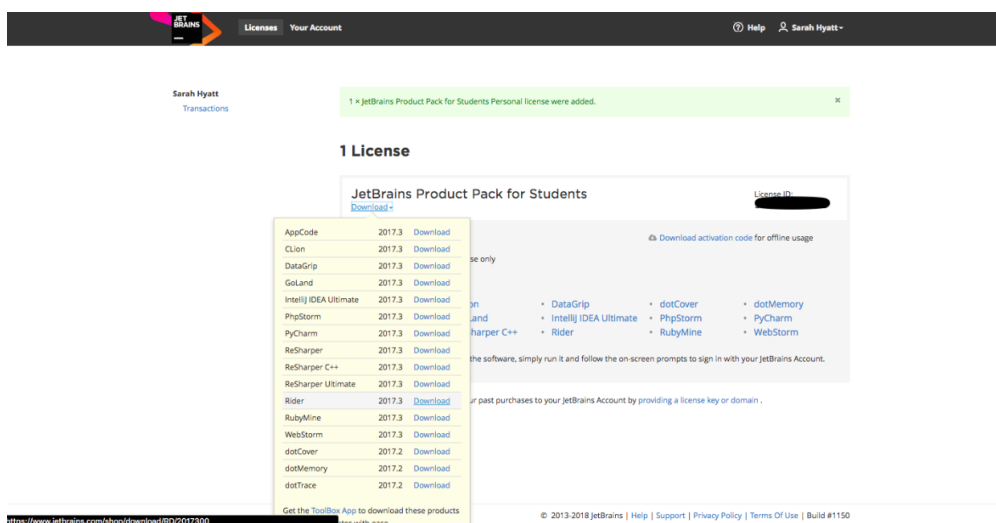
4. Go back to your email and activate

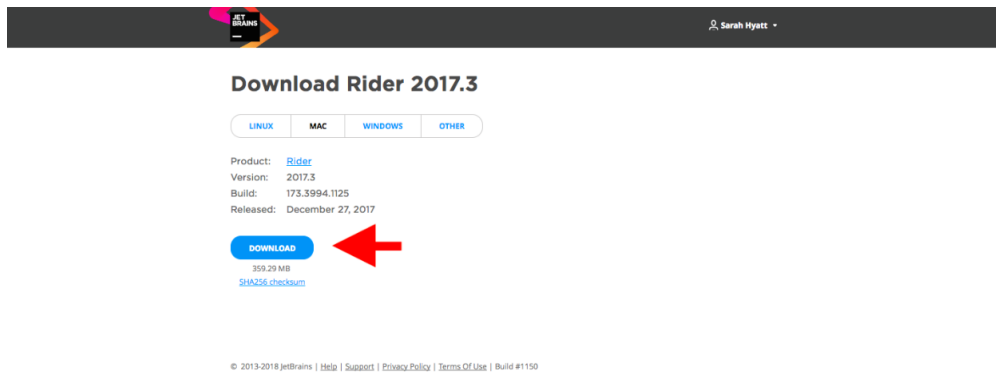


5. Accept the terms (you have no choice)

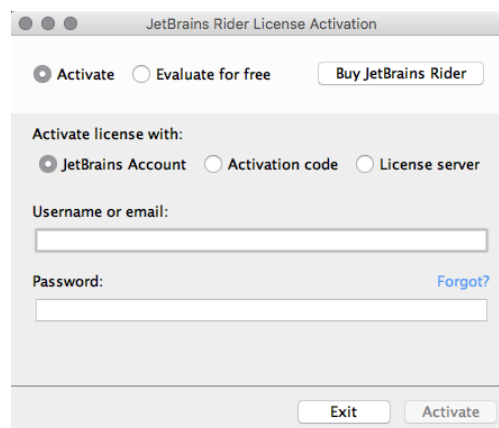


6. Download Rider





7. Register and get started



1.2 Importing DIKU's C# Style to Rider

1. Open: <https://git.dikunix.dk/su18/Guides/blob/master/>
2. Read it, learn it and follow it when you code
3. Download the style guide
4. Open Rider and click on **File -> Import settings** and choose the downloaded file

Deliverable: Add a screen-shot of Rider opened on your computer to your report.

2 Hello World (10%)

With Rider installed and DIKU's code formatting policy implemented, we are ready to implement our first small project in C#.

1. Start up Rider.
2. On the first screen, click "New Solution".

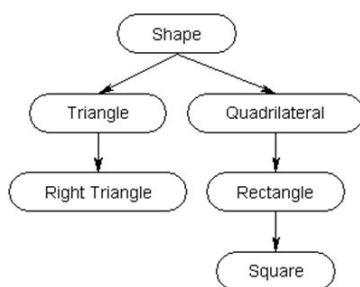
3. In the sidebar, choose "Console Application", and name your solution "Assignment1" with "HelloWorld" as the project name. Pick a suitable location for your project and tick the box to "Put solution and project in the same directory". Click "Create".

Now, take a brief minute to explore and familiarize yourself with the interface of Rider.

1. On the left, in the Solution Explorer, you will find your newly created solution "Assignment1" as the root folder. Below this, your project "HelloWorld" should exist with the two subfolders "References" and "Properties", as well as the file "Program.cs" inside. Open this file.
2. This file contains the special "Main" method, which acts as the entry point of our program. A project may contain several files linked together by internal references, but the Main method will be the first to run.
3. Notice also the lines of code surrounding our Main method. In particular, the method is placed within the class "Program", which in turn resides in the namespace "HelloWorld". These names will come in handy later, when we start to work on larger projects requiring multiple classes and namespaces.
4. For now, simply add the line `Console.WriteLine("Hello World!");`
(*hotkey cw* → TAB)
5. Click the green play button in the top right corner to run your code.
6. Provide a screen-shot of the contents output in your console.

Deliverable: Add the screen-shot of your result displayed on the console to your report.

3 Classes & Object-Oriented Programming in C#



In this exercise you will learn how to work with classes. On the left is a visual example of how a structure for classes may look like.

This task is to build a bank, inheriting from a company, with customers and employees.

3.1 Introducing DIKUBank

Let's start out by creating a bank to generate some much-needed revenue for DIKU!

1. Open Assignment1 in Rider and create a new project, choose a "Console Application" and call it **DIKUBank**.

2. Create an abstract class **Company**, containing a field for the name of the company.
3. Create another class **Bank** (remember to have only one class per file). Let it inherit the name field from **Company** and add in a new private field for the bank balance containing some initial start capital of 20000000 DIKU Dollars defined in the constructor.
4. Test your program by creating a Bank instance in your main method and print the Bank instance to the console (the output is probably not very meaningful to you yet).

3.2 Adding People

Now it's time to add a little bit of life to our world!

1. Create a new abstract class called **Person** with public fields for name, age, and gender.
2. Inheriting from this abstract class, create two new classes **Customer** and **Employee**.
 - All customers should have a name and an account balance. Consider which access modifiers make the most sense for each individual field.
 - All employees should have a name, an account balance, and a count of the total amount of hours worked.
3. Finally, in your main method, create two customers called Sarah and Anders, and a single employee called Boris. Print the instances to your console and verify that no exceptions are thrown.

3.3 Adding Methods

With people added to our world, we should also give them something to do!

1. Both the employee and customer class should be able to deposit and withdraw money.
2. The employee class should be able to work, adding the hours worked to a personal work summary.
3. Test the methods by first depositing 40 DIKU Dollars to Sarah's account, printing the resulting account balance to the console, and then withdrawing 13 DIKU Dollars from Sarah's account, again printing the resulting balance to the console.
4. Test the methods some more by making the employee, Boris, work for 40 hours. Print out the total work hours for Boris.

Hint: You can use this syntax to get the account balance of the customers

```
public int getSomeField {  
    get { return someField; }  
}
```

3.4 Adding People to the Bank

Let's get our bank into business by hiring some staff and making some customers.

1. Give the employees a job position.
2. Create a list of employees and another list of customers in the bank.
 - Hint: In C#, a list can be created using the syntax:
`private List<someType> someFieldName;`
3. In your main method, add two or more employees, Fritz and Lone, to the bank.
4. In your main method, print out the names of every customer, and the name and position of every employee currently registered in the banks lists of customers and employees.

3.5 Adding Value

Finally, let's do some accounting to make sure everyone is paid and the bank is making money!

1. Create a dictionary in the bank with an hourly rate corresponding to each job position, Boris should earn 100 per hours, Fritz 50 per hour and Lone 500 per hour. You can decide for yourself which job positions they should have.
 - Hint: In C#, a dictionary can be created using the syntax:
`private Dictionary<someKeyType, someValueType> someFieldName;`
2. Create another dictionary in the bank, documenting the hourly amount of value added to the bank for each type of job position.
3. Make Fritz work 60 hours and Lone 80 hours.
4. In the bank, create an accounting method as follows:
 - For every employee, get their individual job position and hours worked.
 - Using their job position, find out how much total value they have add to the bank. Add this amount to the bank balance.
 - Using their job position, find out which hourly rate they qualify for and pay out a salary proportionate to the hours worked.
5. Print out the bank balance before and after paying all the employees.
 - Hint: use a get-method to get the bank balance before paying.

Deliverable: Add a screen-shot of the output from exercises 3.1-3.5.

4 Discussion

1. Create a simple diagram of the relational structure of the code implemented above.¹ How are the different classes connected, and why do you think they are structured in this way? (You don't have to fulfil any formal requirements in your diagram as long as the overall structure is clear.)
2. Reflect on the principles of object-oriented programming used in the previous exercise: Did you use any? Which ones? Where?
3. What benefits and drawbacks do you see in using these principles to structure your code?
4. Do you see any ways to improve or expand the implementation above? Try to use some of the theory from the book/lecture to motivate your proposals.

Deliverable: Add the theoretical discussion and your diagram to the report.

¹You can use <http://www.draw.io/>