

Remember the zoo project you completed in Course 1? Now, you're going to take it to the next level. In this lab, you'll update your zoo program to save the details of animals like the tiger, dolphin, and penguin into files. This means that the next time you run the application, you'll be able to pull up all that saved information and continue right where you left off. Pretty neat.

You'll focus on reusing as much of your existing code as possible while also setting it up for maximum reusability in the future. To do this, you'll make some updates to the current classes and add some new code to save and retrieve information from the file.


## Goal

Update the existing *Tiger*, *Penguin*, and *Dolphin* classes to make them serializable, enabling their state to be saved to a file. Then, you'll override the *toString()* method in each class, allowing you to display the deserialized data on the screen after reading it from the file.

You'll then write methods to save each animal's data into separate files—one for the tiger, one for the penguin, and one for the dolphin. After that, you'll create a method to retrieve and display this data from the files, ensuring that the saved information can be easily accessed.

Finally, you'll add two new options to the main menu: one to save all the animal data to their respective files, and another to retrieve and display the data from those files, making the program more interactive and user-friendly.

Ready to dive in? Let's get started!

 Note: When you encounter this icon, it's time to get into your integrated development environment, or IDE, and start coding!

In your lab environment, open IntelliJ by double-clicking on the icon.

When the IntelliJ IDE opens, you will be brought to the starter code. The *main* method of the *main* class is placed in a file named *Main.java*. The starter code includes the following files:

- *Animal.java*
- *Dolphin.java*
- *Eat.java*
- *Main.java*
- *Penguin.java*
- *Swim.java*

- *Tiger.java*
- *Walk.java*

For this lab, you will work with *Tiger.java*, *Penguin.java*, *Dolphin.java*, and *Main.java*.


Here is a quick breakdown of the tasks:

- Task 1 focuses on making the *Tiger*, *Penguin*, and *Dolphin* classes serializable.
- Task 2 will guide you through overriding the *toString()* method in the *Tiger*, *Penguin*, and *Dolphin* classes, allowing you to display the properties in human-readable form after deserialization.
- Task 3 focuses on writing the logic to save the data of the objects into separate files.
- Task 4 involves writing the logic to read the data from the files *tiger.txt*, *penguin.txt*, and *dolphin.txt* and display it on the screen.
- Task 5 takes you through adding a new option in the main menu that saves the objects for all three animal classes. This will call a method that saves the *Tiger*, *Penguin*, and *Dolphin* objects into their respective files.
- Task 6 involves adding a new option in the main menu to read and display the data from the saved files *tiger.txt*, *penguin.txt*, and *dolphin.txt* on the screen.

You need to serialize the *Tiger*, *Penguin*, and *Dolphin* classes by implementing the *Serializable* interface.

Why do you need to serialize these classes? Serializing allows you to convert them into a byte array, which lets you save the state of your objects - a key step in building a program that can store and retrieve data efficiently.

Before you get coding, think about why serialization is important. How will serializing these classes help you save the state of your zoo's animals? Take a moment to consider how this will tie into the bigger picture of your project.

 TODO 1: Make the classes serializable by implementing a *Serializable* interface.

1.a Make *Tiger.java* serializable by implementing the *Serializable* interface.

```
/**
```

```
* TODO 1.a: Implement serializable interface for class Tiger
```

```
*/
```

1.b Make *Penguin.java* serializable by implementing the *Serializable* interface.

```
/**
```


```
* TODO 1.b: Implement serializable interface for class Penguin  
  
*/
```

1.c Make *Dolphin.java* serializable by implementing the *Serializable* interface.  
/\*\*

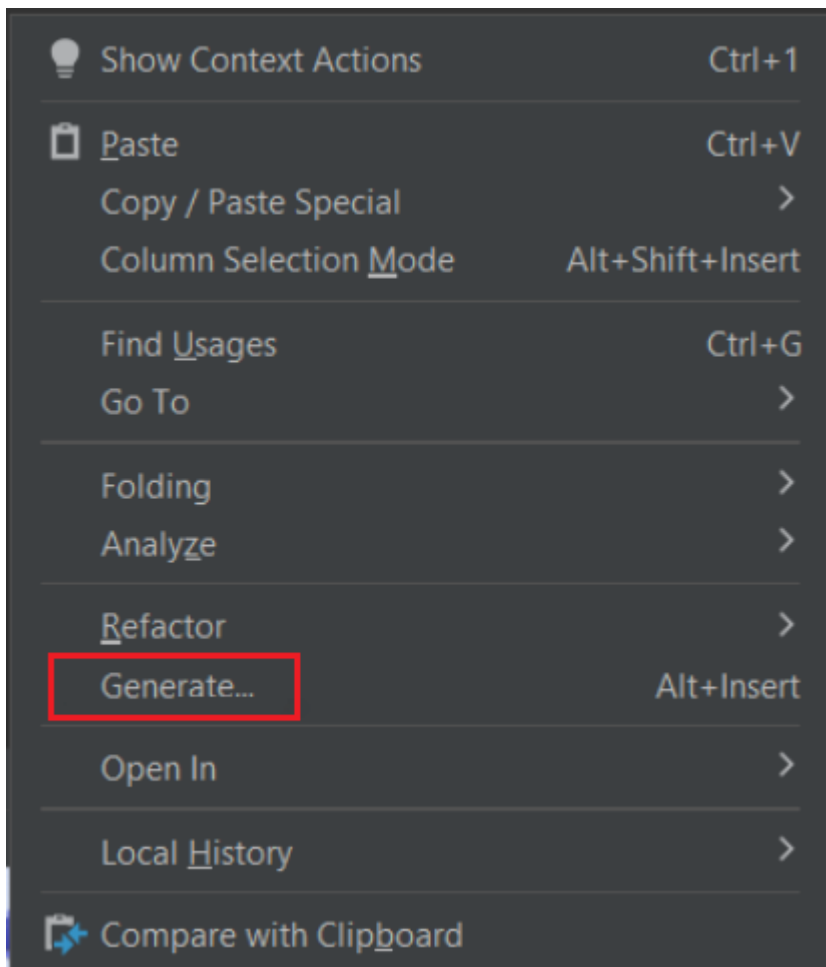
```
* TODO 1.c: Implement serializable interface for class Dolphin  
  
*/
```

Great, you've made the classes serializable. Now, imagine the scenario where you're saving these animal states. How will this serialization affect the way data is stored and retrieved later? Picture the flow of data from your program to the file and back.

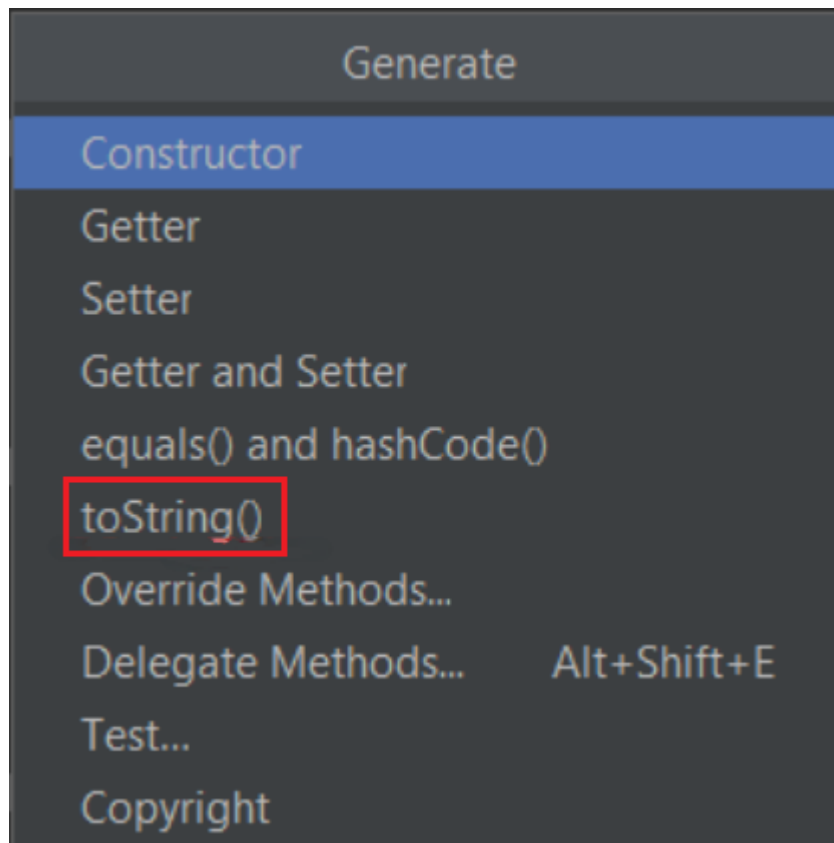
Now, what if you wanted to see the object's data after it has been deserialized from a file? To do this, you must override the *toString()* method in each class: *Tiger.java*, *Penguin.java*, and *Dolphin.java*. This is important because it lets you visually confirm that your data has been correctly restored, helping you debug and validate your program.

 TODO 2: Override the *toString()* method in each class. Start by following these steps:

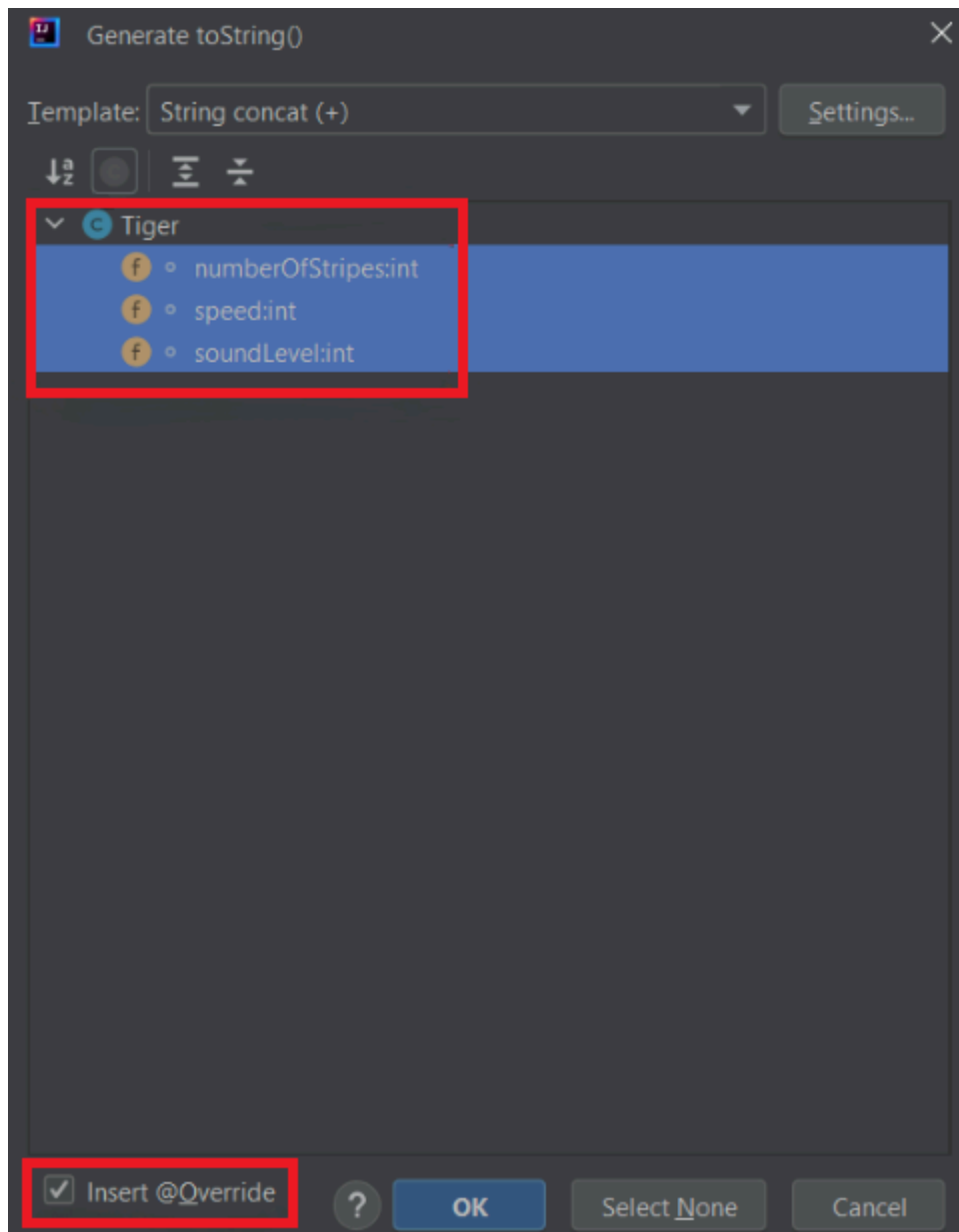
Step 1: In IntelliJ, right-click within the class file and, in the dropdown menu, click on Generate.



Step 2: In the new menu, select toString().



Step 3: In the new window, you'll see that all the attributes from that class included in the *toString()* method are selected by default. Click OK.



Now you can continue with your coding!

💻 It's time to get coding!

2.a Override the *toString()* method in *Tiger.java*.

```
/**  
 * TODO 2.a: Override the toString method display the deserialized content  
 * after reading the file  
 */
```

2.b Override the *toString()* method in *Penguin.java*.

```
/**  
  
 * TODO 2.b: Override the toString method display the deserialized content  
  
 * after reading the file  
  
 */
```

## 2.c Override the *toString()* method in *Dolphin.java*.


```
/**  
  
 * TODO 2.c: Override the toString method display the deserialized content  
  
 * after reading the file  
  
 */
```


With the *toString()* method in place, you can see exactly what's inside your objects after deserialization. How do you think this will help debug or confirm that your data has been correctly retrieved?

## Pause and check

Before starting Task 3, compile and run your program to ensure your existing code functions as expected. You can also experiment with your zoo program by adding different animals and updating their details using the menu options. Observe the output in the console - are the animal details displayed and updated correctly? Now's a good chance to identify and fix bugs in your animal classes or menu logic before moving on to persistent storage in the next tasks.

Here, you'll write method to save your objects into files. This is where your program starts interacting with the file system, turning your in-memory data into something that can be stored and accessed later. You're bringing your zoo to life!

 It's time to get coding!

 **TODO 3:** Open *Main.java* and write a public static method named *writeObjectsToFile* with parameters *Tiger* object, *Penguin* object, and *Dolphin* object and return type void.

3.a Save the *Tiger* object into a file named *tiger.txt*

3.b Save the *Penguin* object into a file named *penguin.txt*

3.c Save the *Dolphin* object into a file named *dolphin.txt*.

```
/**
```

```
 * TODO 3: Write a method named writeObjectsToFile and pass Tiger, Penguin and
Dolphin to be saved onto a file.
```

```
 * TODO 3.a: Save the state of Tiger to output tiger.txt file
```

```
 * TODO 3.b: Save the state of Penguin to output penguin.txt file
```

```
 * TODO 3.c: Save the state of Dolphin to output dolphin.txt file
```


```
 * */
```


## Tip

When handling file operations, include a try-catch block to manage any potential exceptions. In the catch block, you can use the *printStackTrace()* method to output any errors that occur.

Well done! You've just enabled your program to save objects to files, ensuring that the data is stored and available for future use. This is a big leap toward making your program fully functional and realistic.

Next, you'll write method to read your objects back from files. This step is critical because it completes the data cycle—ensuring that what you save can be accurately retrieved.

 It's time to get coding!

 TODO 4: In *Main.java*, write a public static method named *readObjectsFromFile* with no parameters and return type void.

4.a Read the file name *tiger.txt* and print the object state.

4.b Read the file name *penguin.txt* and print the object state.

4.c Read the file name *dolphin.txt* and print the object state.

```
/**
```



```

* TODO 4: Read the file tiger.txt, penguin.txt and dolphin.txt

* TODO 4.a: Print the save state of Tiger from the file tiger.txt

* TODO 4.b: Print the save state of Penguin from the file penguin.txt

* TODO 4.c: Print the save state of Penguin from the file dolphin.txt

*/


```


## Tip

When handling file operations, include a try-catch block to manage any potential exceptions. In the catch block, you can use the *printStackTrace()* method to output any errors that occur.

Excellent! You've successfully enabled your program to read and restore data from files. This means your zoo's data is no longer temporary—it's saved and can be accessed anytime.

Now, you'll add a menu option to save your animal data to files. This makes your program interactive, dynamic, and user-friendly. Other users can now easily save their work with a simple command.

 It's time to get coding!

 TODO 5: In *Main.java*, add a new case in the main menu - for example, case number 4 - to save the objects into separate files by calling the method *writeObjectsToFile*, which you wrote in Task 3.

```

/**

* TODO 5: Introduce case 4 to call the writeObjectsToFile method to save the

* object state of the animal into the file

*/


```


## Tip

When adding new cases to the switch statement, be sure to include a *break*; statement to properly exit each case.

You've added the option to save your objects to files through the menu. How does this fit into the overall functionality of your program? Consider how this feature will be used in real-world scenarios and what could go wrong.

Now, you'll add a menu option to read data from files. This will let users retrieve and display previously saved data, making your program even more versatile and user-focused. You're adding the final touches. Almost there!

 It's time to get coding!

 **TODO 6:** In *Main.java*, add a new case in the main menu - for example, case number 5 - to read the objects from separate files by calling the method *readObjectsFromFile*, which you wrote in Task 4.

```
/**
```

```
 * TODO 6: Introduce case 5 to call the readObjectsFromFile method to
```

```
 * fetch the object state of the animal from the file to display on screen
```

```
 */
```

## Tip

When adding new cases to the switch statement, be sure to include a *break*; statement to properly exit each case.

You've now completed the loop by allowing users to read and display data from files. Your program is now fully functional and interactive—nice work!

If you encounter errors while running the code, here's some tips to help you troubleshoot:

**Compilation Errors:** These errors occur when the compiler cannot understand your code. This often happens because of typos, missing semicolons, or incorrect use of Java syntax. To find the error, carefully review the error messages; these usually point to the specific line of code causing the issue.

**Runtime Errors:** These errors occur while the program is running. Examples include *NullPointerException* (trying to access an object that hasn't been initialized),

*InputMismatchExceptions* (user entering incorrect data types), or logical errors in your code. IntelliJ has a built-in debugger you can use to identify the error source.

Logical Errors: These errors are the hardest to pinpoint, as the program might run without crashing but produce unexpected results. A good approach is to use print statements at various points in your code to check the values of variables and the flow of execution. Doing this can help you pinpoint where the logic is going wrong.

After you have tested and confirmed your project functionality, if you are happy with your results you can now submit your assignment for grading and feedback!

To submit your assignment, look for and click the blue “Submit Assignment” button in the top right corner of your lab environment. This sends your code off to the grader for evaluation. When you submit your assignment, the grader is designed to check how well your code performs its intended functions based on the instructions for this assignment. Think of it as a thorough code review!

You can submit as many times as you’d like! Use the grader as a partner to help improve your code with every submission.

Here’s the crucial part: the grader needs to compile and run your application in order to work. If it can’t get past this stage, your grade will unfortunately be a 0. If you submit your work and the grade is a 0, there is a good chance that your hard work couldn’t compile - not necessarily that your logic or functionality is wrong. So double-check that everything compiles and runs correctly in the IDE before submitting.