

Welcome back, coffee lover! Today, you'll continue your caffeinated coding adventures and consider how best to approach errors in your program. So far, you've explored the very concept of errors in Java code and taken a dive into exceptions, specifically what they are and how to handle them.

Now, remember from the [Implementing error free code](#) lab, how entering a String for the number of servings caused an error and stopped the program? In this final lab for Module 2, you'll modify your coffee machine to tackle such errors head-on! Why? Because a single error can crash the whole program and leave users helpless. A strong programmer thinks ahead and writes code that helps avoid future problems.


So, grab your favorite mug, put on your coding hat, and get brewing!

Goal

You'll program defensively to avoid or handle unchecked exceptions using *try-catch* and *finally* constructs. You'll also throw checked exceptions using *throw* and *throws*. This will help you write robust software that can handle errors gracefully.

In this lab, you'll modify some of the code from an earlier ungraded lab, *Creating multiple classes*. Here's the plan:


- Starter classes (*Coffee*, *CoffeeMachine*, *Espresso* and *Latte*): You are given a starter class, *Coffee*, with some common coffee attributes, a Constructor, and some methods. The *CoffeeMachine* class contains the main method, which provides the user with a menu to choose the desired beverage. Based on the selected beverage (*Espresso* or *Latte*), you can use this method to create the objects and call respective methods. You previously made the *Espresso* and *Latte* subclasses which are inherited from the *Coffee* class. They define specific functionalities and details unique to each beverage type.
- Implementing exception handling: You will explore areas for improvement in the code by anticipating errors and writing suitable defensive code. First, you will learn to throw errors using the *throw* and *throws* keywords, and then you will handle them using the *try*, *catch*, and *finally* blocks.

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

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

Imagine you're setting the caffeine level for a coffee roast, and the user enters something unexpected. Instead of silently setting it to 0, let's throw an exception to alert the user about the mistake. Ready to give it a shot?

Expand the *src* folder and open the provided *Coffee* class. Go to the *setCaffeineLevel()* method. In the last else case, you'll note the caffeine level is set to 0. Let's dive in and code an exception here.

 It's time to get coding!

TODO 1: From inside the last else case, remove the statement that sets *caffeineLevelInMg* to 0. Instead, throw an *InvalidTypeException* with the following message (note: replace '*normal*' with what the user has entered):

```
invalid roast: 'normal', please select a valid roast type!
```

TODO 2: Declare that the *setCaffeineLevel()* method throws an *InvalidTypeException*.

If the roast value is not set properly, the *setCaffeineLevel()* method should now throw an exception.

Run the program from the *CoffeeMachine* class, and observe that the program crashes with the message:

```
"unreported exception com.sun.jdi.InvalidTypeException; must be caught or declared to be thrown"
```

```
// TODO 2: declare that the setCaffeineLevel() method throws a InvalidTypeException
```

```
// Method to set the caffeine level of the coffee based on the roast
```

```
public void setCaffeineLevel() {
```

```
    if (roast.equals("light")) {
```

```
        caffeineLevelInMg = 50;
```

```
    } else if (roast.equals("medium")) {
```

```
        caffeineLevelInMg = 100;
```

```
    } else if (roast.equals("dark")) {
```

```
        caffeineLevelInMg = 150;
```

```
    } else {
```

```
        // TODO 1: remove this statement & throw a "InvalidTypeException" with  
a message
```


```
        caffeineLevelInMg = 0;
```

```
}  
  
}
```

Okay, you've thrown an exception when an invalid roast type is entered. How will this improve the user experience? Consider how users will be alerted to their mistakes and prompted to correct them. This is the first step in making your program more robust.

You've just thrown your first exception. But what happens next? You need to handle this exception to make sure the coffee machine doesn't crash and burn. More specifically, you've seen that if the roast value is not set correctly, the `setCaffeineLevel()` method will throw an exception.

So, how do you fix this? You need to write code to protect your code and ensure that, in case an exception occurs, the normal flow of program execution continues. Let's write some code to catch that exception and keep the coffee brewing!

 It's time to get coding!

- TODO 3: Staying in the Coffee class, surround the `setCaffeineLevel()` method with a *try-catch* block to handle the exception. Handle the appropriate exception.
- TODO 4: Inside the catch block, set the value of the `caffeineLevelInMg` attribute to 50 so that a nice cup of coffee is prepared.

```
// TODO 3: surround the setCaffeineLevel() method call with a try-catch  
block to handle the exception
```

```
// TODO 4: inside the catch block, set the caffeineLevelInMg to 50
```

Rerun the program with the same input and observe the difference this time. The program runs while handling the invalid input.

```
Welcome to the Coffee Machine!
```

```
Select an option to continue:
```

```
1. Espresso
```

```
2. Latte
```

```
3. Exit
```

```
Enter your choice (1, 2, or 3): 1
```

```
What Roast would you like? (light, medium, dark): normal

How many servings would you like? (a number please): 1

Grinding beans for Espresso...

Brewing your favorite Espresso...

You ordered a Espresso with a normal roast.

The caffeine level in your coffee is 50 mg.

You asked for 1 servings!

Every serving of Espresso costs 2.5$. Your total bill is 2.5$.

Welcome to the Coffee Machine!

Select an option to continue:

1. Espresso

2. Latte

3. Exit

Enter your choice (1, 2, or 3):
```

Well done! You've successfully handled the exception for the invalid roast type, ensuring the program continues running smoothly. Setting a default caffeine level made the coffee machine handle unexpected inputs and kept the coffee brewing!

Pause and check

Great progress! Before you continue let's make sure everything is working so far. Run your program and try the following:

1. Open the *CoffeeMachine* class and run the *main()* method.
2. You will be presented with the following menu in the console:

```
Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3): |
```

3. Press *1* to select *Espresso*.

4. Enter *regular* as the value for the roast type. Remember if you've completed the tasks correctly, the code should be able to handle this "exceptional scenario" where the user has entered a random value of roast type and set the caffeine level to *50*.

Expected output

Compare your output with the expected output. Observe that the user has entered *regular* as the roast type and the code makes adjustments accordingly ensuring the program runs correctly.

```
Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3): 1
What Roast would you like? (light, medium, dark): regular
How many servings would you like? (a number please): 1

Grinding beans for Espresso...

Brewing your favorite Espresso...

You ordered a Espresso with a regular roast.
The caffeine level in your coffee is 50 mg.
You asked for 1 servings!
Every serving of Espresso costs 2.5$. Your total bill is 2.5$.

Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3):
```


Troubleshooting

If your output doesn't match the expected output, you can try the following steps:

1. Check whether the `setCaffeineLevel()` method has been called inside the `try` block from `TODO 3` in the `Coffee` class.
2. Check whether the `caffeineLevelInMg` variable has been set to 50 in the `catch` block from `TODO 4` in the `Coffee` class.
3. Check whether the `InvalidTypeException` thrown in `TODO 1` is the same exception that is being caught in the `catch` block in `TODO 3`.

Now, it's time for your next challenge. When creating an Espresso, if the user enters 0 or a negative number of shots, you should throw an exception to let them know they need at least one shot.

Open the provided *Espresso* class and go to the constructor. If the value of *numberOfShots* is less than or equal to 0, then the machine creates a coffee but sets the bill as \$0.0. Throw an exception and ask the user to enter a valid number.

 It's time to get coding!

- TODO 5: Inside the constructor, before setting the value of *numberOfShots*, add an if-else statement to check whether the value of *numberOfShots* is less than or equal to 0. If it is less than or equal to 0, then throw an *ArithmeticException* with the following message:

please select at least 1 serving!

- TODO 6: Declare that the constructor throws an *ArithmeticException*. If the value of the *numberOfShots* attribute is less than or equal to 0, the constructor will now throw an *ArithmeticException*.

```
public class Espresso extends Coffee {  
  
    int numberOfShots;  
  
    // TODO 6: declare that the constructor throws a ArithmeticException  
  
    public Espresso(String name, String roast, double price, int numberOfShots) {  
  
        super(name, roast, price);  
  
        // TODO 5: check whether at least 1 serving has been selected, if not,  
        throw an exception with a message  
  
        this.numberOfShots = numberOfShots;  
  
    }  
}
```

Open and run the program in the *CoffeeMachine* class. Select Espresso, enter "normal" for the roast value, and enter any number for the number of servings.

Rerun the *CoffeeMachine* class. Select *Espresso* and enter 0 or any negative integer for the number of servings.

Observe that the program crashes with the message "please select at least 1 serving!".

```
Welcome to the Coffee Machine!
```

```
Select an option to continue:
```

```
1. Espresso
```

```
2. Latte
```

```
3. Exit
```

```
Enter your choice (1, 2, or 3): 1
```

```
What Roast would you like? (light, medium, dark): normal
```

```
How many servings would you like? (a number please): 0
```

```
Exception in thread "main" java.lang.ArithmeticException: please select at least 1  
serving!
```

```
    at Espresso.<init>(Espresso.java:10)
```

```
    at CoffeeMachine.main(CoffeeMachine.java:39)
```

Great, you've thrown an exception for invalid shot numbers in the *Espresso* constructor. Why is this important? Think about how this prevents the creation of an invalid *espresso* object and guides the user to enter a valid number of shots.

Fantastic! You've successfully thrown an exception for invalid shot numbers. But what if it happens in the *main* method? You must catch and handle it there, too, so your coffee machine remains user-friendly and robust.

Open the provided *CoffeeMachine* class and, as your programmer buddy reminds you...


Remember

The try-catch block can be written without a finally block

```
try () {  
    // ... some statements  
}  
catch (ExceptionType exceptionVariable) {  
    // .... statements to handle exception  
}
```

The *finally* block is always executed irrespective of whether an exception has occurred

```
try () {  
    // ... some statements  
}  
catch (ExceptionType exceptionVariable) {  
    // .... statements to handle exception  
}  
finally {  
    // ... these statements will always execute  
}
```

 It's time to get coding!

- TODO 7: Inside the *main* method, where the *Espresso* object (*myEspresso*) is being initialized, surround the initialization statement with a *try-catch* block.
- TODO 8: Declare the *myEspresso* object before the *try* block and initialize it to null.
- TODO 9: Inside the *catch* block, catch the appropriate exception, ask the user for the number of shots, and store the result in *numberOfShots*. If an invalid value is entered twice, the program *will* crash because the same exception will occur.
- TODO 10: Add a *finally* block after the *catch* block, and initialize the *myEspresso* object again.

```
// TODO 7: surround the myEspresso object with a try-catch block to handle the  
ArithmeticException.
```

```
Espresso myEspresso = new Espresso(espressoName, espressoRoast, espressoPrice,  
numberOfShots);
```

```
// TODO 8: declare the myEspresso object before the try block and set it to null.

// TODO 9: inside the catch block, ask the user to enter number of shots and store
it in numberOfShots

// TODO 10: add a finally block, and initialize the myEspresso object again
```

Rerun the program with the same input and observe the difference. The program runs while handling the invalid input.

Welcome to the Coffee Machine!

Select an option to continue:

1. Espresso

2. Latte

3. Exit

Enter your choice (1, 2, or 3): 1

What Roast would you like? (light, medium, dark): normal

How many servings would you like? (a number please): 0

How many servings would you like? (a number please): 1

Grinding beans for Espresso...

Brewing your favorite Espresso...

You ordered a Espresso with a normal roast.

The caffeine level in your coffee is 50 mg.

You asked for 1 servings!

Every serving of Espresso costs 2.5\$. Your total bill is 2.5\$.

Welcome to the Coffee Machine!

Select an option to continue:

1. Espresso

2. Latte

3. Exit

Enter your choice (1, 2, or 3):

You've handled the exception in the main method, too, ensuring that users are prompted to correct their input if they enter an invalid number of shots.

Great job so far! Now, shift focus to the *Latte*. What does a latte have that an espresso doesn't? That's right - milk! Now, if a user enters an invalid milk type, you should throw an exception to prompt them for a valid one. This way, you ensure every cup of latte meets the proper milky standards.

Open the provided `Latte` class and go to the constructor. If the value of *milkType* is anything except "whole", "skim", "almond", and "oat", you know the user has entered the wrong milk type. Throw an exception and ask the user to enter a valid milk type.



It's time to get coding!

- TODO 11: Add an if-else statement to check whether the value of *milkType* is anything except "whole", "skim", "almond", and "oat". If it is, then throw an `IllegalArgumentException` with the following message:

please select a valid milk type!

- TODO 12: Declare that the constructor throws an `IllegalArgumentException`. If the value of the *milkType* is anything except "whole", "skim", "almond", and "oat", the constructor now throws an `IllegalArgumentException`.

```
public class Latte extends Coffee {

    String milkType;

    String syrupFlavor;

    // TODO 12: declare that the constructors throws a InvalidTypeException

    public Latte(String name, String roast, double price, String milkType, String
syrupFlavor) {

        super(name, roast, price);

        // TODO 11: check whether a valid milkType is selected, if not throw an
exception with a message
```

```
        this.milkType = milkType;

        this.syrupFlavor = syrupFlavor;

    }
}
```

Run the program from the *CoffeeMachine* class, select *Latte*, and enter any string value for milk type except “whole”, “skim”, “almond”, and “oat”. The program crashes with the message:

please select a valid milk type!

```
Welcome to the Coffee Machine!
```

```
Select an option to continue:
```

```
1. Espresso
```

```
2. Latte
```

```
3. Exit
```

```
Enter your choice (1, 2, or 3): 2
```

```
What Roast would you like? (light, medium, dark): light
```

```
What milk type would you like? (whole, skim, almond, oat): 2%
```

```
Would you like syrup? (yes/ no): no
```

```
Exception in thread "main" java.lang.IllegalArgumentException: please select a
valid milk type!
```

```
    at Latte.<init>(Latte.java:10)
```

```
    at CoffeeMachine.main(CoffeeMachine.java:94)
```

Nice job! You've ensured that only valid milk types are accepted when making a latte. By throwing an exception for invalid inputs, you help users provide the correct information, ensuring their latte is made just right. This coffee machine just gets better and better!

Pause and check

Great progress! Before you continue let's make sure everything is working so far. Run your program and try the following:


1. Open the *CoffeeMachine* class and run the *main()* method.
2. You will be presented with the following menu in the console:

```
Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3): |
```

3. Press 2 to select Latte.
4. Enter a *String* for the value of milk type.
5. For the milk type, try entering anything except “whole”, “skim”, “almond”, and “oat”. If you’ve completed all the tasks correctly, the code should understand this is an issue and throw an error.

Great, you've handled exceptions in the *Latte* constructor. But, like before, you must also catch and handle these in the *main* method to keep things running smoothly. Let's ensure your coffee machine can handle any hiccups with grace!

Open the *CoffeeMachine* class and get stuck in!

 It's time to get coding!

- TODO 13: Inside the main method, where the *Latte* object (*myLatte*) is being initialized, surround the initialization statement with a *try-catch* block.
- TODO 14: Declare the *myLatte* object before the try block and set its value to null.
- TODO 15: Inside the *catch* block, catch the appropriate exception, ask the user for milk type, and store it again.
- TODO 16: Add a *finally* block after the *catch* block, and initialize the *myLatte* object again.

```
// TODO 13: surround the myLatte object with a try-catch block to handle the
IllegalArgumentException.
```

```
Latte myLatte = new Latte(latteName, latteRoast, lattePrice, milkType,
syrupFlavor);
```

```
// TODO 14: declare the myLatte object before the try block and set it to null.

// TODO 15: inside the catch block, ask the user to enter milkType again

// TODO 16: add a finally block, and initialize the myLatte object again
```

Run the program with the same input and observe the difference this time. The program runs while handling the invalid input.

Welcome to the Coffee Machine!

Select an option to continue:

1. Espresso

2. Latte

3. Exit

Enter your choice (1, 2, or 3): 2

What Roast would you like? (light, medium, dark): normal

What milk type would you like? (whole, skim, almond, oat): 2%

Would you like syrup? (yes/ no): no

What milk type would you like? (whole, skim, almond, oat): whole

Grinding beans for Latte...

Brewing your favorite Latte...

You ordered a Latte with a normal roast.

The caffeine level in your coffee is 50 mg.

Your latte has whole milk and no flavor.

Your total bill is 3.5\$.

Welcome to the Coffee Machine!

Select an option to continue:

1. Espresso

2. Latte

3. Exit

Enter your choice (1, 2, or 3):

Like before, you've ensured that the *main* method can also handle improper user inputs. Now, it can gracefully handle invalid milk type inputs. By prompting the user to re-enter the correct milk type, you maintain a smooth flow in the application (or coffee machine)!

Excellent work handling all those exceptions! Now it's time to put your application to the test. Run through some scenarios to make sure everything works as expected, and your coffee machine is ready for prime time. Ready, set, test!

1. Run your code using the IDE.

Make sure the exception in the `setCaffeineLevel()` method is handled correctly so that when a correct roast level is not set (except "light", "medium", or "roast"), it is set to the default value of "light" roast (caffeine level is set to 50). That way, the coffee machine will ensure a coffee is always prepared.

```
Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3): 1
What Roast would you like? (light, medium, dark): normal
How many servings would you like? (a number please): 1

Grinding beans for Espresso...

Brewing your favorite Espresso...

You ordered a Espresso with a normal roast.
The caffeine level in your coffee is 0 mg.
You asked for 1 servings!
Every serving of Espresso costs 2.5$. Your total bill is 2.5$.

Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3): |
```

Make sure the exception in the main method of `CoffeeMachine` class, while declaring the `myEspresso` object, is handled properly so that when a correct value of `numberOfShots` is not set (0 or negative), the machine will let the user enter it again.


```
Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3): 1
What Roast would you like? (light, medium, dark): normal
How many servings would you like? (a number please): 0
How many servings would you like? (a number please): 1

Grinding beans for Espresso...

Brewing your favorite Espresso...

You ordered a Espresso with a normal roast.
The caffeine level in your coffee is 50 mg.
You asked for 1 servings!
Every serving of Espresso costs 2.5$. Your total bill is 2.5$.

Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3):
```

Make sure the exception in the main method of `CoffeeMachine` class, while declaring the `myLatte` object, is handled properly so that when the correct `milkType` is not set (except "whole", "skim", "almond", and "oat"), the machine will let the user enter it again.

```
Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3): 2
What Roast would you like? (light, medium, dark): normal
What milk type would you like? (whole, skim, almond, oat): 2%
Would you like syrup? (yes/ no): no
What milk type would you like? (whole, skim, almond, oat): whole

Grinding beans for Latte...

Brewing your favorite Latte...

You ordered a Latte with a normal roast.
The caffeine level in your coffee is 50 mg.
Your latte has whole milk and no flavor.
Your total bill is 3.5$.

Welcome to the Coffee Machine!
Select an option to continue:
1. Espresso
2. Latte
3. Exit
Enter your choice (1, 2, or 3):
```

2. Carefully input test values.

3. Compare your program's output to the expected results provided in the screenshots

You've run through various scenarios to test the program. What have you observed? Think about how each exception-handling mechanism you've implemented contributes to the overall robustness of the program. How does this testing help ensure that users will have a smooth experience?

Congratulations, Java master brewer! In this lab, you have successfully programmed defensively to protect your code from unchecked exceptions.

Here's a quick recap of what you've achieved in this lab:

1. Improved user input handling: You threw exceptions for invalid inputs, guiding users to provide correct information and ensuring your coffee machine produces only the best brews.
2. Ensured smooth program flow: By handling exceptions gracefully, you kept the program running smoothly, even when users made mistakes.
3. Enhanced software robustness: Your defensive programming techniques have made the coffee machine robust and reliable, preventing crashes and improving the overall user experience.

It all boils down to “Expect the unexpected”. Writing code this way creates fewer bugs later and saves you hours of debugging. Plus, it makes sure your coffee machine keeps brewing delicious cups without a hitch. Keep up the great work and happy coding!