Lab 02 Computer Science 1027B Summer 2025

Introduction

The purpose of these labs is to provide practice problems for you to get used to the concepts being covered in class. It is very important that you do these labs on your own without relying on available AI tools. Questions on exams are often very similar to exercises presented in these labs, so understanding them and solving these problems gives you a huge advantage for the exams. Make sure that you complete both the code as well as provide written answers to the blue questions within the boxes.

When you have concluded this lab, you should submit the following files to OWL Brightspace in a zip file:

SpaceExplorer.java (code written for Exercise 01)

MarsExplorer.java (code written for Exercise 01)

MoonExplorer.java (code written for Exercise 01)

AsteroidExplorer.java (code written for Exercise 01)

HealthDepletionException.java (code written for Exercise 02)

ItemNotFoundException.java (code written for Exercise 02)

GameCharacter.java (code written for Exercise 02)

AdventureGame.java (code written for Exercise 02)

SimpleDivision.java (code written for Exercise 03)

written_answers.txt (written answers to questions in all exercises)

Topics Covered

- Classes
- Interfaces
- Exceptions
- Exception Handling

Lab 02 Computer Science 1027B Summer 2025

Exercise 01

Exploring Interfaces

You are part of the Space Exploration Agency (SEA)! Exciting! You are tasked with designing a system for different types of space explorers. Each space explorer can perform various activities such as exploring planets, collecting samples, and sending reports.

All explorers should adhere to a common interface but may have different ways of performing their tasks.

First, you should create an interface called '**SpaceExplorer**' with the following abstract methods:

- void explore()
- void collectSamples()
- String sendReport()

Now, let's create three different classes that implement the '**SpaceExplorer**' interface. Each class should have a unique way of performing the tasks defined in the interface.

MarsExplorer

explore() prints the String "MarsExplorer is exploring the red planet!"

collectSamples() prints the String "MarsExplorer is collecting Martian soil samples!"

sendReport() returns the String "MarsExplorer: Exploration and sample collection on Mars complete!"

MoonExplorer

explore() prints the String "MoonExplorer is exploring the lunar surface!"

collectSamples() prints the String "MoonExplorer is collecting moon rocks!"

sendReport() returns the String "MoonExplorer: Exploration and sample collection on the Moon complete!"

AsteroidExplorer

explore() prints the String "MoonExplorer is navigating the asteroid belt!"

collectSamples() prints the String "AsteroidExplorer is collecting asteroid dust samples!"

sendReport() returns the String "Asteroid Explorer: Exploration and sample collection on asteroids complete!"

Q 01.1

Reflect on the design of the SpaceExplorer interface and its implementations in the MarsExplorer, MoonExplorer, and AsteroidExplorer classes. What are the advantages and potential disadvantages of using interfaces this way?

Great! Now that that's done, let's write a default method in the **SpaceExplorer** interface. We'll call it prepareForMission and it won't return anything (think about what the return type should be in this case). The method should print the String "Preparing for the mission. Checking all systems and supplies".

Create a new file called **SpaceExplorationAgency.java**, and paste the following code into the file:

```
public class SpaceExplorationAgency {
    public static void main(String[] args) {
        SpaceExplorer[] explorers = {
            new MarsExplorer(),
            new MoonExplorer(),
            new AsteroidExplorer()
        };

        for (int i = 0; i < explorers.length; i++) {
            SpaceExplorer explorer = explorers[i];
            explorer.prepareForMission();
            explorer.explore();
            explorer.collectSamples();
            System.out.println(explorer.sendReport());
            System.out.println();
        }
    }
}
```

Q 01.2

Why is it beneficial to store the different explorer objects ('MarsExplorer', 'MoonExplorer', and 'AsteroidExplorer') in an array of the 'SpaceExplorer' type instead of creating separate arrays for each explorer type?

Q 01.3

Compile and execute SpaceExplorationAgency.java. Copy and paste your output.

Exercise 02

Congratulations! You got a new job working with a popular video game developer. They are working on developing an adventure game where a character can perform actions such as attacking and using items. These actions might encounter specific problems, such as the character's health dropping to zero or attempting to use an item that doesn't exist. You need to handle these problems using exceptions to ensure the system can manage errors gracefully.

Start by writing two custom exception classes (each custom exception should be in its own file).

- **HealthDepletionException**
- **ItemNotFoundException**

Each exception class should simply extend "Exception" and have a constructor that accepts a message String.

Next, create a class called **GameCharacter** with the following methods:

- void attack() throws HealthDepletionException
- void useItem(String item) throws ItemNotFoundException

The class should have:

- A private instance variable for health, an int initialized to 50
- A private instance variable for items, an ArrayList of Strings. When a GameCharacter object is created, add a "Potion" and a "ThrowingStar" to the inventory.

I'll give this to you the field declarations and constructor start off with:

```
import java.util.ArrayList;

public class GameCharacter {
    private int health = 50;
    private ArrayList<String> inventory = new ArrayList<>();

    public GameCharacter() {
        inventory.add("Potion");
        inventory.add("ThrowingStar");
    }
}
```

Now, we will want to write two public methods to start:

- **void attack()** which decreases health by 10 and throws a **'HealthDepletionException'** with the message "Character's health has depleted!" if health drops to or below zero. Otherwise, print "Character attacks! Health is now: <health>".
- **void useItem(String item)** which removes the item from the inventory and throws **ItemNotFoundException** with the message "Item not found in character's inventory!" if the item doesn't exist. Otherwise, remove the item and print "Character used: <item>".

To present a clearer idea of what this class should look like, consider this template:

```java
import java.util.ArrayList;

public class GameCharacter {
    private int health = 50;
    private ArrayList<String> inventory = new ArrayList<>();

    public GameCharacter() {
        inventory.add("Potion");
        inventory.add("ThrowingStar");
    }

    public void attack() throws HealthDepletionException {
        // your code here
    }

    public void useItem(String item) throws ItemNotFoundException {
        // your code here
        // you may want to use the contains() method in this method!
        // (look back into the week 1 slides at ArrayList methods)
    }
}
```

Q 02.1

Why might we write 'throws <exception name>' at the end of a method signature? What does it signify?

Now, let's write a main method. Copy and paste the following code into a file named **AdventureGame.java**. Now, let's take a look:

```java
public class AdventureGame {
    public static void main(String[] args) {
        GameCharacter character = new GameCharacter();

        character.attack();
        character.attack();
        character.attack();
        character.attack();
        character.attack();
        character.attack(); // this should trigger a HealthDepletionException

        character.useItem("Branch"); // this should trigger a ItemNotFoundException
    }
}
```

Q 02.2

What happens when we try to compile and execute the above AdventureGame class? What is missing from the above main method? Hint: It has to do with checked versus unchecked exceptions.

Adjust the above main method so that the following is printed when AdventureGame.java is compiled and executed (hint: consider where try-catch blocks might be needed because of checked exceptions):

Character attacks! Health is now: 40

Character attacks! Health is now: 30

Character attacks! Health is now: 20

Character attacks! Health is now: 10

Character's health has depleted!

Item not found in character's inventory!

Great work! The video game company thanks you and celebrates your efforts. This is going to win Game of the Year for sure!

Exercise 03

Let's create a class called 'SimpleDivision' with a single public static method:

- divide(int a, int b)

That returns the result of dividing int a by int b.

In this method, use a try-catch-finally block to handle potential ArithmeticException (division by zero).

In the catch block, print the message 'Error: Division by zero is not allowed' and return a default value of 0.

In the finally block, print the message 'Division operation is complete' to indicate that the division operation is complete.

After writing the functionality for the divide method in the following template, run the file.

```
public class SimpleDivision {
    public static int divide(int a, int b) {
        // your code here
    }

    public static void main(String[] args) {
        int result1 = SimpleDivision.divide(10, 2);
        System.out.println("Result: " + result1);

        int result2 = SimpleDivision.divide(10, 0); // This should trigger an ArithmeticException
        System.out.println("Result: " + result2);

        int result3 = SimpleDivision.divide(10, 5);
        System.out.println("Result: " + result3);
    }
}
```

The output should be:

Division operation is complete

Result: 5

Error: Division by zero is not allowed

Division operation is complete

Result: 0

Division operation is complete

Result: 2

Q 03.1

Why do we not need to create an instance of SimpleDivision to call the divide method on?