**Question 1**

**Part 1:**

**a)**

**1. Structured Programming**

Structured programming is a programming paradigm that emphasizes a logical structure in the program's control flow, making it easier to understand, develop, and maintain.

**Key Characteristics**

* **Control Structures**: Utilizes sequences, selections (if/else), and iterations (loops) for managing flow.
* **Top-down Design**: Programs are designed by breaking down complex tasks into simpler, smaller functions or modules.
* **Modularity**: Code is organized into independent modules, promoting reusability and easier management.
* **Single Entry and Exit Points**: Functions have a single entry and exit, enhancing clarity and control flow.
* **Data Hiding**: Promotes encapsulation by restricting access to internal states of functions or modules.

**Principles**

* **Structured Control Flow**: Encourages the use of structured programming constructs rather than unstructured control statements like goto.
* **Separation of Concerns**: Each module or function should have a specific responsibility, making the code easier to understand and maintain.
* **Top-down Approach**: Focuses on defining the main function first and then detailing sub-functions.
* **Clear Documentation**: Emphasizes the importance of comments and documentation to explain the purpose of functions and modules.
* **Iterative Improvement**: Encourages refining and enhancing code through successive iterations and testing.

**Areas of Application**

* Primarily used in procedural programming languages like C, Pascal, and Ada.
* Suitable for applications that can be broken down into smaller, well-defined functions, such as data processing and system programming.

**2. Object-oriented Programming (OOP)**

Object-oriented programming is a programming paradigm based on the concept of "objects," which can contain data in the form of fields (attributes) and code in the form of methods (functions).

**Key Characteristics**

* **Encapsulation**: Combines data and methods that operate on that data within classes, restricting access to internal states.
* **Inheritance**: Allows new classes to inherit properties and methods from existing classes, promoting code reuse.
* **Polymorphism**: Enables methods to act differently based on the object invoking them, allowing for flexible code.
* **Abstraction**: Simplifies complex systems by modeling classes based on essential characteristics, hiding unnecessary details.
* **Message Passing**: Objects communicate through messages, allowing for interaction and collaboration.

**Principles**

* **DRY (Don't Repeat Yourself)**: Encourages code reusability to reduce redundancy in code.
* **SOLID Principles**: A set of design principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) for improving software design.
* **Composition over Inheritance**: Prefers composing objects with desired functionality over relying solely on class inheritance.
* **Interface Design**: Focuses on defining clear interfaces for classes to promote loose coupling and high cohesion.
* **Design Patterns**: Utilizes proven design patterns to solve common problems in software design.

**Areas of Application**

* Widely used in software development for complex systems, such as GUI applications, games, and enterprise software.
* Commonly implemented in languages like Java, C++, Python, and C#.

**3. Functional Programming**

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state or mutable data.

**Key Characteristics**

* **First-Class Functions**: Treats functions as first-class citizens, allowing them to be passed as arguments, returned from other functions, and assigned to variables.
* **Immutability**: Emphasizes immutable data structures, leading to fewer side effects and easier reasoning about code.
* **Higher-Order Functions**: Functions that can take other functions as arguments or return them, enabling powerful abstractions.
* **Pure Functions**: Focuses on functions that produce the same output for the same input without side effects.
* **Declarative Style**: Concentrates on what to solve rather than how to solve it, leading to more concise and expressive code.

**Principles**

* **Recursion**: Utilizes recursion as a primary mechanism for iteration, avoiding traditional looping constructs.
* **Function Composition**: Encourages combining simple functions to build more complex functions, enhancing modularity.
* **Referential Transparency**: Ensures expressions can be replaced with their values without changing the program's behavior.
* **Avoiding Side Effects**: Promotes writing functions that do not alter any state or produce observable side effects.
* **Lazy Evaluation**: Delays evaluation of expressions until their values are needed, improving efficiency and enabling infinite data structures.

**Areas of Application**

* Used in scenarios where data transformation is key, such as data analysis, concurrent programming, and web development.
* Commonly found in languages like Haskell, Scala, and in functional aspects of JavaScript and Python.

**Conclusion**

Each of these programming paradigms offers unique definitions, principles, and characteristics suited for different types of problems and applications. Understanding these paradigms allows developers to choose the best approach for their specific needs, leading to more effective and maintainable code.

**b)**

### Explained below are reasons why Object-oriented programming (OOP) is preferred for reusable and modular software, each followed by a brief explanation.

**Encapsulation:** Encapsulation groups data and methods that operate on that data within objects. This hides the internal state of the object and only exposes necessary interfaces. By restricting direct access to the object's data, developers can prevent unintended interference and maintain control over how data is accessed and modified.

**Inheritance:** Inheritance allows a new class (subclass) to inherit attributes and behaviors from an existing class (superclass). This promotes code reuse by enabling developers to extend existing functionality without rewriting code. For instance, if a base class defines common behaviors, subclasses can add or modify those behaviors as needed.

**Polymorphism:** Polymorphism enables objects of different classes to be treated as objects of a common superclass. This allows for method overriding, where a subclass can provide a specific implementation of a method already defined in its superclass. It enhances flexibility in code, allowing developers to write more generic and reusable code.

**Modularity:** OOP promotes breaking down software into discrete, self-contained modules (classes). Each class can encapsulate specific functionality, making the code easier to understand and maintain. This modularity allows teams to work on different parts of a system concurrently, facilitating parallel development.

**Abstraction:** Abstraction simplifies complex systems by focusing on the essential features and hiding unnecessary details. By defining clear interfaces, developers can interact with objects without needing to understand their internal workings. This leads to cleaner, more understandable code and enhances user experience.

**Code Reusability:** OOP promotes writing code that can be reused across different projects or modules. Developers can create generic classes and methods that can be instantiated or extended, significantly reducing duplication of effort and the likelihood of bugs. This reusability is particularly valuable for maintaining large codebases.

**Design Patterns:** OOP encourages the use of design patterns, which are proven solutions to common software design problems. By leveraging these patterns, developers can apply best practices that enhance the structure and maintainability of their code, making it easier to solve recurring problems and improve overall software quality.

**Improved Maintenance:** Because OOP promotes modularity and encapsulation, it simplifies the maintenance of software. When changes are needed, developers can often modify a single class without affecting other parts of the system. This leads to lower maintenance costs and reduced risk of introducing bugs when updating or extending functionality.

**Collaboration:** OOP facilitates collaboration among developers by allowing them to work on different classes or modules independently. Clear interfaces and encapsulated functionality mean that team members can integrate their work without needing to understand every detail of others' implementations. This leads to more efficient teamwork and faster project completion.

**Scalability:** OOP supports scalability by making it easier to extend existing systems. New features can often be added through new classes or by extending existing ones, allowing the software to evolve without requiring a complete rewrite. This adaptability is crucial for businesses that need to respond to changing market demands or user needs over time.

**Part B**

**a)**

import java.util.Scanner;

public class NextAIInc {

private static final double MIN\_BASE\_RATE = 30000; // Minimum base hourly rate

private static final int REGULAR\_HOURS\_LIMIT = 48; // Regular hours limit

private static final int MAX\_HOURS\_LIMIT = 72; // Maximum hours limit

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Get input from the user

System.out.print("Enter base hourly rate (UGX): ");

double baseHourlyRate = scanner.nextDouble();

System.out.print("Enter total hours worked: ");

int hoursWorked = scanner.nextInt();

// Calculate payment

calculatePayment(baseHourlyRate, hoursWorked);

}

private static void calculatePayment(double baseHourlyRate, int hoursWorked) {

// Check for minimum base hourly rate

if (baseHourlyRate < MIN\_BASE\_RATE) {

System.out.println("Error: Base hourly rate must not be below UGX 30,000.");

return;

}

// Check for maximum hours limit

if (hoursWorked > MAX\_HOURS\_LIMIT) {

System.out.println("Error: Contractors cannot work more than 72 hours per week.");

return;

}

// Calculate payment

double payment = 0;

if (hoursWorked <= REGULAR\_HOURS\_LIMIT) {

payment = baseHourlyRate \* hoursWorked; // Regular pay

} else {

int overtimeHours = hoursWorked - REGULAR\_HOURS\_LIMIT;

payment = (baseHourlyRate \* REGULAR\_HOURS\_LIMIT) + (2 \* baseHourlyRate \* overtimeHours); // Regular + overtime pay

}

**Outputs**

### Scenario 1: Valid Input (Regular Hours)

**Input:**

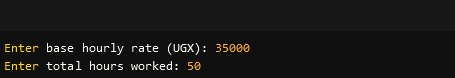
****

**Output:**

****

### Scenario 2: Valid Input (Overtime Hours)

**Input:**

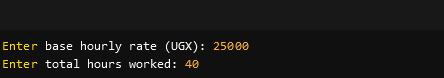
****

**Output:**

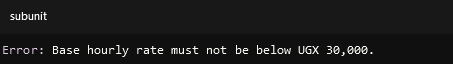
****

### Scenario 3: Base Hourly Rate Below Minimum

**Input:**

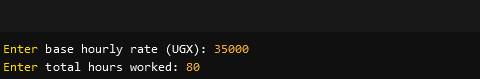
****

**Output:**

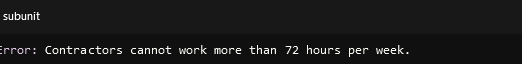
****

### Scenario 4: Hours Worked Exceed Maximum Limit

**Input:**

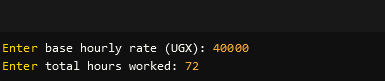
****

**Output:**

****

### Scenario 5: Edge Case (Exactly Maximum Hours Allowed)

**Input:**

****

**Output:**

****

**b)**

1. Java program thatContains a method that accepts base pay and hours worked as parameters.

import java.util.Scanner;

public class NextAIInc {

private static final double MIN\_BASE\_RATE = 30000; // Minimum base hourly rate

private static final int REGULAR\_HOURS\_LIMIT = 48; // Regular hours limit

private static final int MAX\_HOURS\_LIMIT = 72; // Maximum hours limit

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Get input from the user

System.out.print("Enter base hourly rate (UGX): ");

double baseHourlyRate = scanner.nextDouble();

System.out.print("Enter total hours worked: ");

int hoursWorked = scanner.nextInt();

// Calculate payment using the new method

double payment = calculatePayment(baseHourlyRate, hoursWorked);

// Output the payment if valid

if (payment >= 0) {

System.out.printf("Total payment for the week: UGX %.2f%n", payment);

}

}

// Method that accepts base pay and hours worked as parameters

private static double calculatePayment(double baseHourlyRate, int hoursWorked) {

// Check for minimum base hourly rate

if (baseHourlyRate < MIN\_BASE\_RATE) {

System.out.println("Error: Base hourly rate must not be below UGX 30,000.");

return -1; // Return -1 to indicate an error

}

// Check for maximum hours limit

if (hoursWorked > MAX\_HOURS\_LIMIT) {

System.out.println("Error: Contractors cannot work more than 72 hours per week.");

return -1; // Return -1 to indicate an error

}

// Calculate payment

double payment = 0;

if (hoursWorked <= REGULAR\_HOURS\_LIMIT) {

payment = baseHourlyRate \* hoursWorked; // Regular pay

} else {

int overtimeHours = hoursWorked - REGULAR\_HOURS\_LIMIT;

payment = (baseHourlyRate \* REGULAR\_HOURS\_LIMIT) + (2 \* baseHourlyRate \* overtimeHours); // Regular + overtime pay

}

return payment; // Return the calculated payment

}

}

### Scenario 1: Valid Input (Regular Hours)

**Input:**

Enter base hourly rate (UGX): 35000

Enter total hours worked: 40

**Output:**

****

### Scenario 2: Valid Input (Overtime Hours)

**Input:**

Enter base hourly rate (UGX): 35000

Enter total hours worked: 50

**Output:**

****

### Scenario 3: Base Hourly Rate Below Minimum

**Input:**

Enter base hourly rate (UGX): 25000

Enter total hours worked: 40

**Output:**

****

### Scenario 4: Hours Worked Exceed Maximum Limit

**Input:**

Enter base hourly rate (UGX): 35000

Enter total hours worked: 80

**Output:**

****

### Scenario 5: Edge Case (Exactly Maximum Hours Allowed)

**Input:**

Enter base hourly rate (UGX): 40000

Enter total hours worked: 72

**Output:**

### Scenario 5: Edge Case (Exactly Maximum Hours Allowed)

**Input:**

Enter base hourly rate (UGX): 40000

Enter total hours worked: 72

**Output:**

b.ii)

**Java program that contains a method that calculates and prints the total weekly pay for the contractor or displays the appropriate error message.**

Contractor A

import java.util.Scanner;

public class NextAIInc {

private static final double MIN\_BASE\_RATE = 30000; // Minimum base hourly rate

private static final int REGULAR\_HOURS\_LIMIT = 48; // Regular hours limit

private static final int MAX\_HOURS\_LIMIT = 72; // Maximum hours limit

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Get input from the user

System.out.print("Enter base hourly rate (UGX): ");

double baseHourlyRate = scanner.nextDouble();

System.out.print("Enter total hours worked: ");

int hoursWorked = scanner.nextInt();

// Calculate payment using the method that accepts parameters

double payment = calculatePayment(baseHourlyRate, hoursWorked);

// Output the payment if valid

if (payment >= 0) {

System.out.printf("Total payment for the week: UGX %.2f%n", payment);

}

}

// Method that accepts base pay and hours worked as parameters

private static double calculatePayment(double baseHourlyRate, int hoursWorked) {

// Check for minimum base hourly rate

if (baseHourlyRate < MIN\_BASE\_RATE) {

System.out.println("Error: Base hourly rate must not be below UGX 30,000.");

return -1; // Return -1 to indicate an error

}

// Check for maximum hours limit

if (hoursWorked > MAX\_HOURS\_LIMIT) {

System.out.println("Error: Contractors cannot work more than 72 hours per week.");

return -1; // Return -1 to indicate an error

}

// Calculate payment

double payment = 0;

if (hoursWorked <= REGULAR\_HOURS\_LIMIT) {

payment = baseHourlyRate \* hoursWorked; // Regular pay

} else {

int overtimeHours = hoursWorked - REGULAR\_HOURS\_LIMIT;

payment = (baseHourlyRate \* REGULAR\_HOURS\_LIMIT) + (2 \* baseHourlyRate \* overtimeHours); // Regular + overtime pay

}

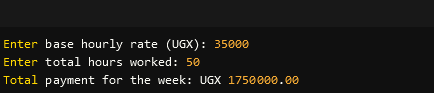
return payment; // Return the calculated payment

}

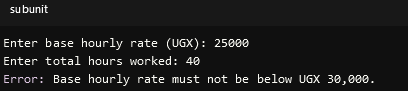
}

### Outputs

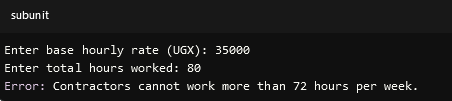
* For valid input:



For invalid input (base rate too low):



For invalid input (hours too high):



**Contractor B**

Given the input for **Contractor B** with a base pay of UGX 20,000/hour and hours worked at 40 hours, the Java program will check the base pay against the minimum required rate of UGX 30,000.

**Input for Contractor B**

* **Base Pay**: UGX 20,000/hour
* **Hours Worked**: 40 hours

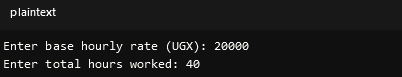
**Expected Outcome**

Since the base pay is below the minimum required rate, the program will display the following error message:

****

### How the Program Processes This Input

When you run the program and input the values:

****

The calculatePayment method will perform the following checks:

1. **Minimum Base Rate Check**:
   * It will check if baseHourlyRate < MIN\_BASE\_RATE (i.e., 20000 < 30000). This condition is true.
2. **Error Message Display**:
   * The program will print the error message:

****

**Return Value**: The method returns -1 to indicate an error.

**Main Method Handling**: The main method checks the return value. Since it is -1, it does not print any payment amount.

The program correctly identifies the error due to the base pay being below the minimum required rate and provides the appropriate error message, as expected.

**Contractor C**

For **Contractor C**, with a base pay of UGX 35,000/hour and hours worked at 96 hours, the Java program will check the hours worked against the maximum allowed limit of 72 hours.

**Input for Contractor C**

* **Base Pay**: UGX 35,000/hour
* **Hours Worked**: 96 hours

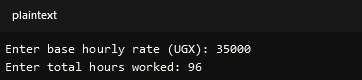
**Expected Outcome**

Since the hours worked exceed the maximum limit, the program will display the following error message:

****

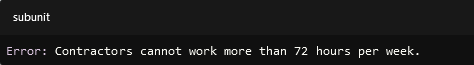
### How the Program Processes This Input

When you run the program with these values:

****

The calculate Payment method will perform the following checks:

1. **Minimum Base Rate Check**: It checks if baseHourly Rate < MIN\_BASE\_RATE (i.e., 35000 < 30000). This condition is false, so it proceeds to the next check.
2. **Maximum Hours Limit Check**: It checks if hours Worked > MAX\_HOURS\_LIMIT (i.e., 96 > 72). This condition is true.
3. **Error Message Display**: The program will print the error message:

****

**Return Value**: The method returns -1 to indicate an error.

**Main Method Handling**: The main method checks the return value. Since it is -1, it does not print any payment amount.

The program correctly identifies the error due to the hours worked exceeding the maximum allowed limit and provides the appropriate error message, as expected.

**Question two**

A Java program that implements a class called TriathlonResults with the specified attributes for participants in the triathlon.

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

this.name = name;

this.participantId = participantId;

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter for Name

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

// Getter and Setter for Participant ID

public String getParticipantId() {

return participantId;

}

public void setParticipantId(String participantId) {

this.participantId = participantId;

}

// Getter and Setter for Swimming Time

public int getSwimmingTime() {

return swimmingTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

// Getter and Setter for Cycling Time

public int getCyclingTime() {

return cyclingTime;

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

// Getter and Setter for Running Time

public int getRunningTime() {

return runningTime;

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

// Getter for Total Time

public int getTotalTime() {

return totalTime;

}

// Method to calculate total time

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

// Method to display participant details

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nSwimming Time: %d minutes%nCycling Time: %d minutes%nRunning Time: %d minutes%nTotal Time: %d minutes%n",

name, participantId, swimmingTime, cyclingTime, runningTime, totalTime);

}

// Main method for testing

public static void main(String[] args) {

// Create participants

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

TriathlonResults bob = new TriathlonResults("Bob", "P002", 20, 35, 25);

TriathlonResults charlie = new TriathlonResults("Charlie", "P003", 30, 50, 30);

TriathlonResults diana = new TriathlonResults("Diana", "P004", 28, 42, 18);

// Display details for each participant

alice.displayDetails();

System.out.println();

bob.displayDetails();

System.out.println();

charlie.displayDetails();

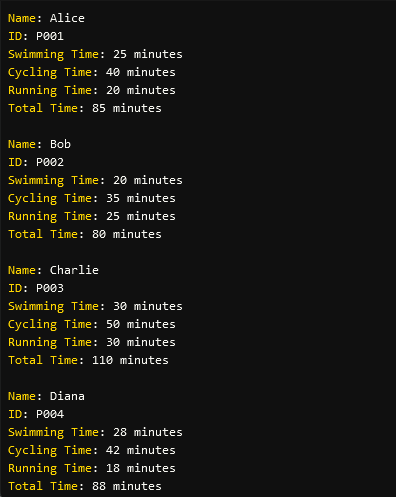
System.out.println();

diana.displayDetails();

}

}

**Output**

****

**b)**

A derived class Elite Participant that inherits from the Triathlon Results class and adds a new attribute for the sponsor's name:

// Base class

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

this.name = name;

this.participantId = participantId;

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter methods

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getParticipantId() {

return participantId;

}

public void setParticipantId(String participantId) {

this.participantId = participantId;

}

public int getSwimmingTime() {

return swimmingTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

public int getCyclingTime() {

return cyclingTime;

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

public int getRunningTime() {

return runningTime;

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

public int getTotalTime() {

return totalTime;

}

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nSwimming Time: %d minutes%nCycling Time: %d minutes%nRunning Time: %d minutes%nTotal Time: %d minutes%n",

name, participantId, swimmingTime, cyclingTime, runningTime, totalTime);

}

}

// Derived class

class EliteParticipant extends TriathlonResults {

private String sponsorName; // New attribute for sponsor's name

// Constructor

public EliteParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime, String sponsorName) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

this.sponsorName = sponsorName;

}

// Getter and Setter for Sponsor Name

public String getSponsorName() {

return sponsorName;

}

public void setSponsorName(String sponsorName) {

this.sponsorName = sponsorName;

}

// Overriding displayDetails method

@Override

public void displayDetails() {

super.displayDetails(); // Call the base class method

System.out.printf("Sponsor Name: %s%n", sponsorName); // Add sponsor information

}

// Main method for testing

public static void main(String[] args) {

// Create participants

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

EliteParticipant bob = new EliteParticipant("Bob", "P002", 20, 35, 25, "Fitness Inc.");

TriathlonResults charlie = new TriathlonResults("Charlie", "P003", 30, 50, 30);

EliteParticipant diana = new EliteParticipant("Diana", "P004", 28, 42, 18, "Health Corp.");

// Display details for each participant

alice.displayDetails();

System.out.println();

bob.displayDetails();

System.out.println();

charlie.displayDetails();

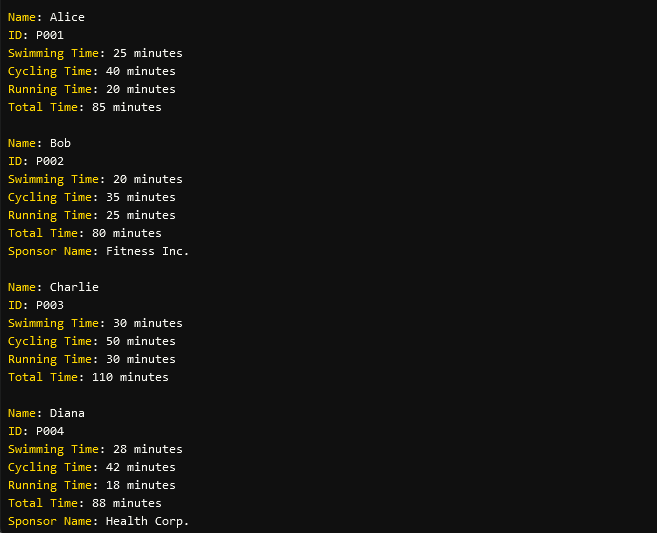
System.out.println();

diana.displayDetails();

}

}

**Output**



**c)**

**Java coe that creates a derived class called BeginnerParticipant that inherits from the TriathlonResults class without adding any additional attributes**

// Base class

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

this.name = name;

this.participantId = participantId;

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter methods

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getParticipantId() {

return participantId;

}

public void setParticipantId(String participantId) {

this.participantId = participantId;

}

public int getSwimmingTime() {

return swimmingTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

public int getCyclingTime() {

return cyclingTime;

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

public int getRunningTime() {

return runningTime;

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

public int getTotalTime() {

return totalTime;

}

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nSwimming Time: %d minutes%nCycling Time: %d minutes%nRunning Time: %d minutes%nTotal Time: %d minutes%n",

name, participantId, swimmingTime, cyclingTime, runningTime, totalTime);

}

}

// Derived class for Elite Participants

class EliteParticipant extends TriathlonResults {

private String sponsorName; // New attribute for sponsor's name

// Constructor

public EliteParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime, String sponsorName) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

this.sponsorName = sponsorName;

}

// Getter and Setter for Sponsor Name

public String getSponsorName() {

return sponsorName;

}

public void setSponsorName(String sponsorName) {

this.sponsorName = sponsorName;

}

// Overriding displayDetails method

@Override

public void displayDetails() {

super.displayDetails(); // Call the base class method

System.out.printf("Sponsor Name: %s%n", sponsorName); // Add sponsor information

}

}

// Derived class for Beginner Participants

class BeginnerParticipant extends TriathlonResults {

// Constructor

public BeginnerParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

}

// displayDetails method inherits from TriathlonResults

}

// Main method for testing

class Main {

public static void main(String[] args) {

// Create participants

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

EliteParticipant bob = new EliteParticipant("Bob", "P002", 20, 35, 25, "Fitness Inc.");

BeginnerParticipant charlie = new BeginnerParticipant("Charlie", "P003", 30, 50, 30);

BeginnerParticipant diana = new BeginnerParticipant("Diana", "P004", 28, 42, 18);

// Display details for each participant

alice.displayDetails();

System.out.println();

bob.displayDetails();

System.out.println();

charlie.displayDetails();

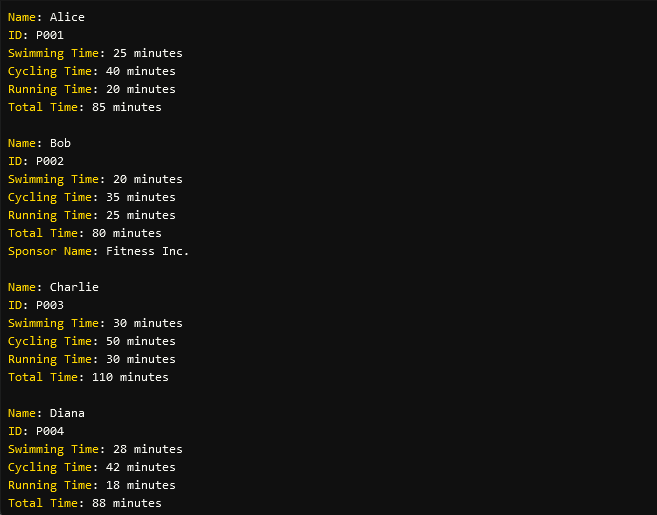
System.out.println();

diana.displayDetails();

}

}

**Output**



**d)**

**Java code that** implements polymorphism in the TriathlonResults class hierarchy, we can modify the displayDetails() method in the base class to show generic details (name, ID, and total time) and then extend this method in the EliteParticipant subclass to include sponsor information.

// Base class

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

this.name = name;

this.participantId = participantId;

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter methods

public String getName() {

return name;

}

public String getParticipantId() {

return participantId;

}

public int getTotalTime() {

return totalTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

// Display generic details

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nTotal Time: %d minutes%n",

name, participantId, totalTime);

}

}

// Derived class for Elite Participants

class EliteParticipant extends TriathlonResults {

private String sponsorName; // New attribute for sponsor's name

// Constructor

public EliteParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime, String sponsorName) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

this.sponsorName = sponsorName;

}

// Overriding displayDetails method to include sponsor information

@Override

public void displayDetails() {

super.displayDetails(); // Call the base class method

System.out.printf("Sponsor Name: %s%n", sponsorName); // Add sponsor information

}

}

// Derived class for Beginner Participants

class BeginnerParticipant extends TriathlonResults {

// Constructor

public BeginnerParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

}

// displayDetails method inherits from TriathlonResults

}

// Main class for testing

class Main {

public static void main(String[] args) {

// Create participants

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

EliteParticipant bob = new EliteParticipant("Bob", "P002", 20, 35, 25, "Fitness Inc.");

BeginnerParticipant charlie = new BeginnerParticipant("Charlie", "P003", 30, 50, 30);

BeginnerParticipant diana = new BeginnerParticipant("Diana", "P004", 28, 42, 18);

// Display details for each participant

alice.displayDetails();

System.out.println();

bob.displayDetails();

System.out.println();

charlie.displayDetails();

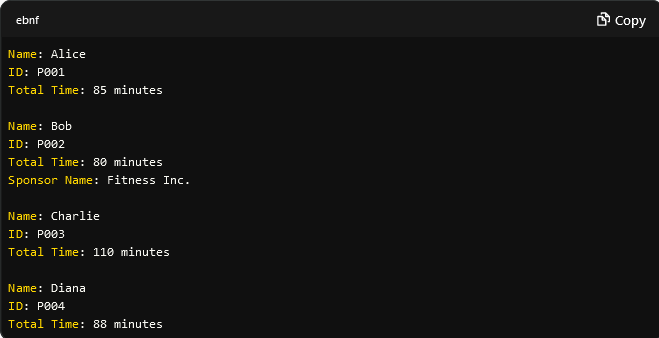
System.out.println();

diana.displayDetails();

}

}

**Output**

****

**e).i)**

Java program to include a method that calculates the total time for each participant by summing their times for swimming, cycling.

// Base class

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

this.name = name;

this.participantId = participantId;

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter methods

public String getName() {

return name;

}

public String getParticipantId() {

return participantId;

}

public int getTotalTime() {

return totalTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

// Method to calculate total time

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

// Method to get individual times

public void displayTimes() {

System.out.printf("Swimming Time: %d minutes%nCycling Time: %d minutes%nRunning Time: %d minutes%n",

swimmingTime, cyclingTime, runningTime);

}

// Display generic details

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nTotal Time: %d minutes%n",

name, participantId, totalTime);

}

}

// Derived class for Elite Participants

class EliteParticipant extends TriathlonResults {

private String sponsorName; // New attribute for sponsor's name

// Constructor

public EliteParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime, String sponsorName) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

this.sponsorName = sponsorName;

}

// Overriding displayDetails method to include sponsor information

@Override

public void displayDetails() {

super.displayDetails(); // Call the base class method

System.out.printf("Sponsor Name: %s%n", sponsorName); // Add sponsor information

}

}

// Derived class for Beginner Participants

class BeginnerParticipant extends TriathlonResults {

// Constructor

public BeginnerParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

}

}

// Main class for testing

class Main {

public static void main(String[] args) {

// Create participants

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

EliteParticipant bob = new EliteParticipant("Bob", "P002", 20, 35, 25, "Fitness Inc.");

BeginnerParticipant charlie = new BeginnerParticipant("Charlie", "P003", 30, 50, 30);

BeginnerParticipant diana = new BeginnerParticipant("Diana", "P004", 28, 42, 18);

// Display details for each participant

alice.displayDetails();

alice.displayTimes();

System.out.println();

bob.displayDetails();

bob.displayTimes();

System.out.println();

charlie.displayDetails();

charlie.displayTimes();

System.out.println();

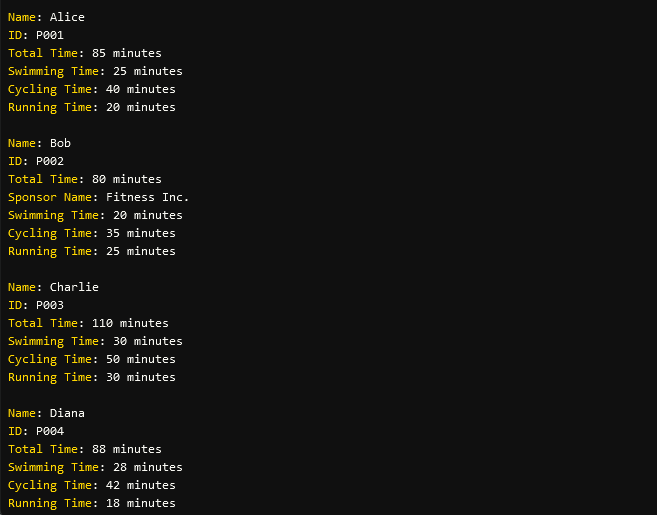
diana.displayDetails();

diana.displayTimes();

}

}

**Output**

****

**e.ii.i)**

**Java code to determine and print the participant with the fastest total time:**

// Base class

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

this.name = name;

this.participantId = participantId;

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter methods

public String getName() {

return name;

}

public String getParticipantId() {

return participantId;

}

public int getTotalTime() {

return totalTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

// Method to calculate total time

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

// Method to get individual times

public void displayTimes() {

System.out.printf("Swimming Time: %d minutes%nCycling Time: %d minutes%nRunning Time: %d minutes%n",

swimmingTime, cyclingTime, runningTime);

}

// Display generic details

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nTotal Time: %d minutes%n",

name, participantId, totalTime);

}

}

// Derived class for Elite Participants

class EliteParticipant extends TriathlonResults {

private String sponsorName; // New attribute for sponsor's name

// Constructor

public EliteParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime, String sponsorName) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

this.sponsorName = sponsorName;

}

// Overriding displayDetails method to include sponsor information

@Override

public void displayDetails() {

super.displayDetails(); // Call the base class method

System.out.printf("Sponsor Name: %s%n", sponsorName); // Add sponsor information

}

}

// Derived class for Beginner Participants

class BeginnerParticipant extends TriathlonResults {

// Constructor

public BeginnerParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

}

}

// Main class for testing

class Main {

public static void main(String[] args) {

// Create participants

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

EliteParticipant bob = new EliteParticipant("Bob", "P002", 20, 35, 25, "Fitness Inc.");

BeginnerParticipant charlie = new BeginnerParticipant("Charlie", "P003", 30, 50, 30);

BeginnerParticipant diana = new BeginnerParticipant("Diana", "P004", 28, 42, 18);

// Store participants in an array for easy access

TriathlonResults[] participants = {alice, bob, charlie, diana};

// Display details for each participant

for (TriathlonResults participant : participants) {

participant.displayDetails();

participant.displayTimes();

System.out.println();

}

// Determine the participant with the fastest total time

TriathlonResults fastestParticipant = participants[0]; // Start with the first participant

for (TriathlonResults participant : participants) {

if (participant.getTotalTime() < fastestParticipant.getTotalTime()) {

fastestParticipant = participant; // Update if a faster participant is found

}

}

// Print the fastest participant's details

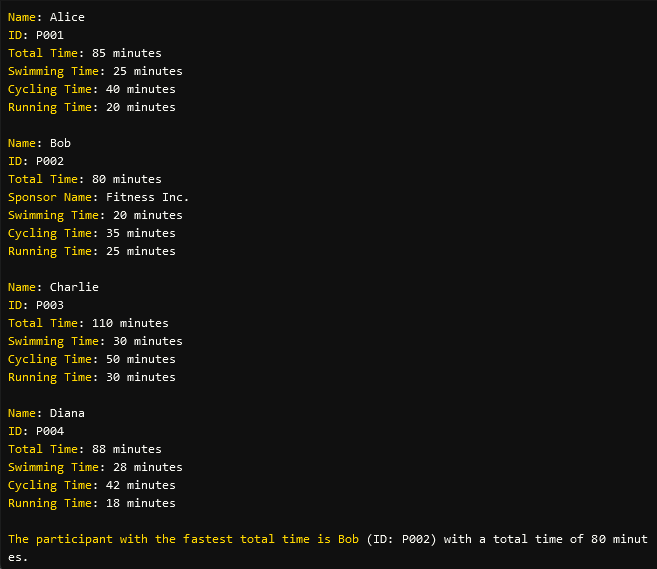
System.out.printf("The participant with the fastest total time is %s (ID: %s) with a total time of %d minutes.%n",

fastestParticipant.getName(), fastestParticipant.getParticipantId(), fastestParticipant.getTotalTime());

}

}

**Output**

****

**e.ii.ii)**

**Java code to determine and print the participant with the second fastest total time.**

import java.util.Arrays;

import java.util.Comparator;

// Base class

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

this.name = name;

this.participantId = participantId;

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter methods

public String getName() {

return name;

}

public String getParticipantId() {

return participantId;

}

public int getTotalTime() {

return totalTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

// Method to calculate total time

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

// Method to get individual times

public void displayTimes() {

System.out.printf("Swimming Time: %d minutes%nCycling Time: %d minutes%nRunning Time: %d minutes%n",

swimmingTime, cyclingTime, runningTime);

}

// Display generic details

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nTotal Time: %d minutes%n",

name, participantId, totalTime);

}

}

// Derived class for Elite Participants

class EliteParticipant extends TriathlonResults {

private String sponsorName; // New attribute for sponsor's name

// Constructor

public EliteParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime, String sponsorName) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

this.sponsorName = sponsorName;

}

// Overriding displayDetails method to include sponsor information

@Override

public void displayDetails() {

super.displayDetails(); // Call the base class method

System.out.printf("Sponsor Name: %s%n", sponsorName); // Add sponsor information

}

}

// Derived class for Beginner Participants

class BeginnerParticipant extends TriathlonResults {

// Constructor

public BeginnerParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

}

}

// Main class for testing

class Main {

public static void main(String[] args) {

// Create participants

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

EliteParticipant bob = new EliteParticipant("Bob", "P002", 20, 35, 25, "Fitness Inc.");

BeginnerParticipant charlie = new BeginnerParticipant("Charlie", "P003", 30, 50, 30);

BeginnerParticipant diana = new BeginnerParticipant("Diana", "P004", 28, 42, 18);

// Store participants in an array for easy access

TriathlonResults[] participants = {alice, bob, charlie, diana};

// Display details for each participant

for (TriathlonResults participant : participants) {

participant.displayDetails();

participant.displayTimes();

System.out.println();

}

// Sort participants based on total time

Arrays.sort(participants, Comparator.comparingInt(TriathlonResults::getTotalTime));

// Print the fastest participant's details

System.out.printf("The participant with the fastest total time is %s (ID: %s) with a total time of %d minutes.%n",

participants[0].getName(), participants[0].getParticipantId(), participants[0].getTotalTime());

// Print the second fastest participant's details

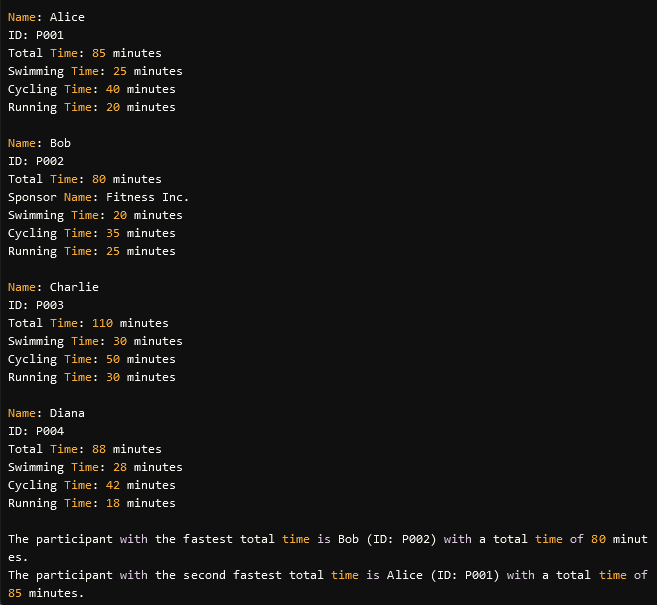
System.out.printf("The participant with the second fastest total time is %s (ID: %s) with a total time of %d minutes.%n",

participants[1].getName(), participants[1].getParticipantId(), participants[1].getTotalTime());

}

}

**Output**

****

**e.ii.iii)**

import java.util.Arrays;

import java.util.Comparator;

// Base class

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

this.name = name;

this.participantId = participantId;

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter methods

public String getName() {

return name;

}

public String getParticipantId() {

return participantId;

}

public int getTotalTime() {

return totalTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

// Method to calculate total time

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

// Method to get individual times

public void displayTimes() {

System.out.printf("Swimming Time: %d minutes%nCycling Time: %d minutes%nRunning Time: %d minutes%n",

swimmingTime, cyclingTime, runningTime);

}

// Display generic details

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nTotal Time: %d minutes%n",

name, participantId, totalTime);

}

}

// Derived class for Elite Participants

class EliteParticipant extends TriathlonResults {

private String sponsorName; // New attribute for sponsor's name

// Constructor

public EliteParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime, String sponsorName) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

this.sponsorName = sponsorName;

}

// Overriding displayDetails method to include sponsor information

@Override

public void displayDetails() {

super.displayDetails(); // Call the base class method

System.out.printf("Sponsor Name: %s%n", sponsorName); // Add sponsor information

}

}

// Derived class for Beginner Participants

class BeginnerParticipant extends TriathlonResults {

// Constructor

public BeginnerParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

}

}

// Main class for testing

class Main {

public static void main(String[] args) {

// Create participants

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

EliteParticipant bob = new EliteParticipant("Bob", "P002", 20, 35, 25, "Fitness Inc.");

BeginnerParticipant charlie = new BeginnerParticipant("Charlie", "P003", 30, 50, 30);

BeginnerParticipant diana = new BeginnerParticipant("Diana", "P004", 28, 42, 18);

// Store participants in an array for easy access

TriathlonResults[] participants = {alice, bob, charlie, diana};

// Sort participants based on total time

Arrays.sort(participants, Comparator.comparingInt(TriathlonResults::getTotalTime));

// Display sorted results

System.out.println("Results sorted by total time:");

for (TriathlonResults participant : participants) {

participant.displayDetails();

participant.displayTimes();

System.out.println();

}

// Print the fastest participant's details

System.out.printf("The participant with the fastest total time is %s (ID: %s) with a total time of %d minutes.%n",

participants[0].getName(), participants[0].getParticipantId(), participants[0].getTotalTime());

// Print the second fastest participant's details

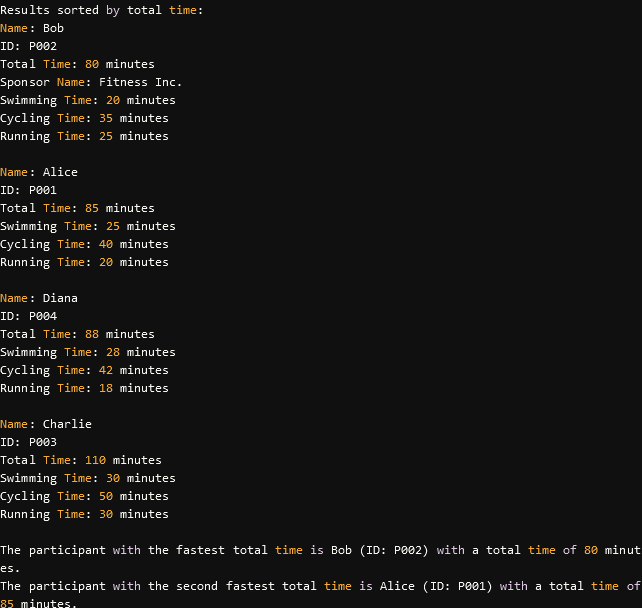
System.out.printf("The participant with the second fastest total time is %s (ID: %s) with a total time of %d minutes.%n",

participants[1].getName(), participants[1].getParticipantId(), participants[1].getTotalTime());

}

}

**Output**

****

**f)**

Java code to include functionality to handle scenarios where two participants have the same total time, ensuring that both are properly ranked.

import java.util.Arrays;

import java.util.Comparator;

// Base class

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

this.name = name;

this.participantId = participantId;

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter methods

public String getName() {

return name;

}

public String getParticipantId() {

return participantId;

}

public int getTotalTime() {

return totalTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

// Method to calculate total time

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

// Method to get individual times

public void displayTimes() {

System.out.printf("Swimming Time: %d minutes%nCycling Time: %d minutes%nRunning Time: %d minutes%n",

swimmingTime, cyclingTime, runningTime);

}

// Display generic details

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nTotal Time: %d minutes%n",

name, participantId, totalTime);

}

}

// Derived class for Elite Participants

class EliteParticipant extends TriathlonResults {

private String sponsorName; // New attribute for sponsor's name

// Constructor

public EliteParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime, String sponsorName) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

this.sponsorName = sponsorName;

}

// Overriding displayDetails method to include sponsor information

@Override

public void displayDetails() {

super.displayDetails(); // Call the base class method

System.out.printf("Sponsor Name: %s%n", sponsorName); // Add sponsor information

}

}

// Derived class for Beginner Participants

class BeginnerParticipant extends TriathlonResults {

// Constructor

public BeginnerParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

}

}

// Main class for testing

class Main {

public static void main(String[] args) {

// Create participants

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

EliteParticipant bob = new EliteParticipant("Bob", "P002", 20, 35, 25, "Fitness Inc.");

BeginnerParticipant charlie = new BeginnerParticipant("Charlie", "P003", 30, 50, 30);

BeginnerParticipant diana = new BeginnerParticipant("Diana", "P004", 28, 42, 18);

// Adding a participant with the same total time as Alice

TriathlonResults eve = new TriathlonResults("Eve", "P005", 25, 40, 20); // Same total time as Alice

// Store participants in an array for easy access

TriathlonResults[] participants = {alice, bob, charlie, diana, eve};

// Sort participants based on total time

Arrays.sort(participants, Comparator.comparingInt(TriathlonResults::getTotalTime));

// Display sorted results

System.out.println("Results sorted by total time:");

for (TriathlonResults participant : participants) {

participant.displayDetails();

participant.displayTimes();

System.out.println();

}

// Print the fastest participant's details

System.out.printf("The participant with the fastest total time is %s (ID: %s) with a total time of %d minutes.%n",

participants[0].getName(), participants[0].getParticipantId(), participants[0].getTotalTime());

// Print the second fastest participant's details

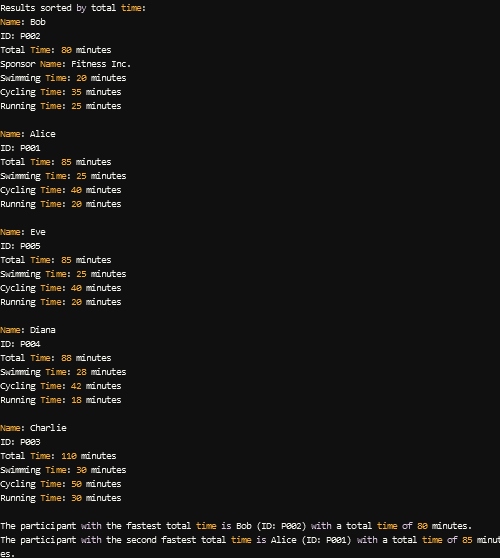
System.out.printf("The participant with the second fastest total time is %s (ID: %s) with a total time of %d minutes.%n",

participants[1].getName(), participants[1].getParticipantId(), participants[1].getTotalTime());

}

}

**Output**

****

**g)**

Java Program to ensure the program outputs meaningful error messages if any times are missing or invalid (e.g., negative times).

import java.util.Arrays;

import java.util.Comparator;

// Base class

public class TriathlonResults {

// Private attributes

private String name;

private String participantId;

private int swimmingTime; // in minutes

private int cyclingTime; // in minutes

private int runningTime; // in minutes

private int totalTime; // in minutes

// Constructor

public TriathlonResults(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

if (name == null || name.isEmpty()) {

throw new IllegalArgumentException("Participant name cannot be null or empty.");

}

if (participantId == null || participantId.isEmpty()) {

throw new IllegalArgumentException("Participant ID cannot be null or empty.");

}

this.name = name;

this.participantId = participantId;

// Validate input times

if (swimmingTime < 0 || cyclingTime < 0 || runningTime < 0) {

throw new IllegalArgumentException("Times cannot be negative.");

}

setSwimmingTime(swimmingTime);

setCyclingTime(cyclingTime);

setRunningTime(runningTime);

calculateTotalTime();

}

// Getter and Setter methods

public String getName() {

return name;

}

public String getParticipantId() {

return participantId;

}

public int getTotalTime() {

return totalTime;

}

public void setSwimmingTime(int swimmingTime) {

if (swimmingTime < 0) {

throw new IllegalArgumentException("Swimming time cannot be negative.");

}

this.swimmingTime = swimmingTime;

calculateTotalTime(); // Recalculate total time

}

public void setCyclingTime(int cyclingTime) {

if (cyclingTime < 0) {

throw new IllegalArgumentException("Cycling time cannot be negative.");

}

this.cyclingTime = cyclingTime;

calculateTotalTime(); // Recalculate total time

}

public void setRunningTime(int runningTime) {

if (runningTime < 0) {

throw new IllegalArgumentException("Running time cannot be negative.");

}

this.runningTime = runningTime;

calculateTotalTime(); // Recalculate total time

}

// Method to calculate total time

private void calculateTotalTime() {

this.totalTime = swimmingTime + cyclingTime + runningTime;

}

// Method to get individual times

public void displayTimes() {

System.out.printf("Swimming Time: %d minutes%nCycling Time: %d minutes%nRunning Time: %d minutes%n",

swimmingTime, cyclingTime, runningTime);

}

// Display generic details

public void displayDetails() {

System.out.printf("Name: %s%nID: %s%nTotal Time: %d minutes%n",

name, participantId, totalTime);

}

}

// Derived class for Elite Participants

class EliteParticipant extends TriathlonResults {

private String sponsorName; // New attribute for sponsor's name

// Constructor

public EliteParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime, String sponsorName) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

this.sponsorName = sponsorName;

}

// Overriding displayDetails method to include sponsor information

@Override

public void displayDetails() {

super.displayDetails(); // Call the base class method

System.out.printf("Sponsor Name: %s%n", sponsorName); // Add sponsor information

}

}

// Derived class for Beginner Participants

class BeginnerParticipant extends TriathlonResults {

// Constructor

public BeginnerParticipant(String name, String participantId, int swimmingTime, int cyclingTime, int runningTime) {

super(name, participantId, swimmingTime, cyclingTime, runningTime);

}

}

// Main class for testing

class Main {

public static void main(String[] args) {

try {

// Create participants with validation

TriathlonResults alice = new TriathlonResults("Alice", "P001", 25, 40, 20);

EliteParticipant bob = new EliteParticipant("Bob", "P002", 20, 35, 25, "Fitness Inc.");

BeginnerParticipant charlie = new BeginnerParticipant("Charlie", "P003", 30, 50, 30);

BeginnerParticipant diana = new BeginnerParticipant("Diana", "P004", 28, 42, 18);

// Adding a participant with the same total time as Alice

TriathlonResults eve = new TriathlonResults("Eve", "P005", 25, 40, 20); // Same total time as Alice

// Store participants in an array for easy access

TriathlonResults[] participants = {alice, bob, charlie, diana, eve};

// Sort participants based on total time

Arrays.sort(participants, Comparator.comparingInt(TriathlonResults::getTotalTime));

// Display sorted results

System.out.println("Results sorted by total time:");

for (TriathlonResults participant : participants) {

participant.displayDetails();

participant.displayTimes();

System.out.println();

}

// Print the fastest participant's details

System.out.printf("The participant with the fastest total time is %s (ID: %s) with a total time of %d minutes.%n",

participants[0].getName(), participants[0].getParticipantId(), participants[0].getTotalTime());

// Print the second fastest participant's details

System.out.printf("The participant with the second fastest total time is %s (ID: %s) with a total time of %d minutes.%n",

participants[1].getName(), participants[1].getParticipantId(), participants[1].getTotalTime());

} catch (IllegalArgumentException e) {

// Catch and print the error message

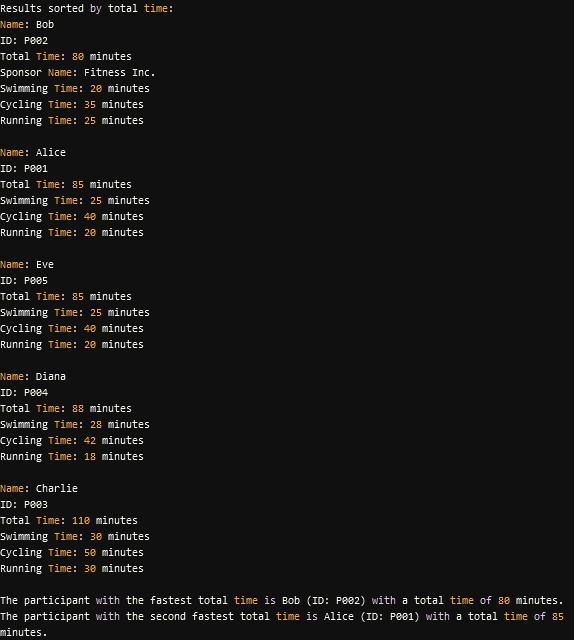
System.err.println("Error: " + e.getMessage());

}

}

}

**Output**

****

If an invalid time or input is provided, such as:

****

The output is:

****