**1.Add JComboBox for Account Types**

a. **Declare JComboBox and AccountType Field:**

Add these lines to the class fields section, just above the MAXACCOUNTS statement:

java

Copy code

private JComboBox<AccountType> accountTypes;

private AccountType actType = AccountType.SAVINGS;

b. **Set Up JComboBox:**

Add the following code where specified (under the code setting up displayJButton but before displayJLabel setup):

java

Copy code

// set up accountTypes combo box

accountTypes = new JComboBox<AccountType>(AccountType.values());

accountTypes.setBounds(16, 238, 176, 24);

inputDetailJPanel.add(accountTypes);

accountTypes.addActionListener(

new ActionListener() // anonymous inner class

{

// event handler called when the combo box selection changes

public void actionPerformed(ActionEvent event) {

actType = (AccountType) accountTypes.getSelectedItem();

}

}

); // end call to addActionListener

**2. Update Application Window Size**

a. **Update Window Size:**

Find the code that sets the size of the window in the createUserInterface method, which may look like this:

java

Copy code

setSize(670, 308);

Change it to:

java

Copy code

setSize(670, 340);

b. **Update Panel Size:**

Find the code that sets up inputDetailJPanel, which may look like this:

java

Copy code

inputDetailJPanel.setBounds(16, 16, 208, 250);

Update the height to:

java

Copy code

inputDetailJPanel.setBounds(16, 16, 208, 280);

**3. Update JTextArea Size**

a. **Update JTextArea Height:**

Find the code that sets up displayJTextArea, which may look like this:

java

Copy code

displayJTextArea.setBounds(16, 300, 208, 200);

Increase the height to:

java

Copy code

displayJTextArea.setBounds(16, 300, 208, 245);

2.

### ****Update the**** myAccounts ****Array Declaration****

You need to update the myAccounts array to use the superclass AbstractBankAccount instead of the Account class. This allows you to store both SavingsAccount and CreditAccount objects in the array.

**Original Declaration:**

java

Copy code

private Account[] myAccounts;

**Updated Declaration:**

Assuming MAXACCOUNTS is a constant defining the maximum number of accounts:

java

Copy code

private AbstractBankAccount[] myAccounts = new AbstractBankAccount[MAXACCOUNTS];

### b. ****Update Account Creation Logic****

You need to modify the createAccountJButtonActionPerformed method to handle different account types. This involves adding an if statement that checks the selected account type and creates the appropriate subclass of AbstractBankAccount.

**Update createAccountJButtonActionPerformed Method:**

Find the section of the code where new accounts are added, likely after the finally block, and update it as follows:

1. **Identify and Extract Account Type:**

You have a JComboBox for selecting the account type, which is stored in actType.

1. **Update the Account Creation Logic:**

Here is a simplified example of how you might update the createAccountJButtonActionPerformed method:

private void createAccountJButtonActionPerformed(ActionEvent event) {

// Assume there are JTextFields for account number and initial balance

String accountNumber = accountNumberTextField.getText();

double initialBalance = Double.parseDouble(initialBalanceTextField.getText());

// Check the selected account type

AbstractBankAccount newAccount = null;

if (actType == AccountType.SAVINGS) {

newAccount = new SavingsAccount(accountNumber, initialBalance);

} else if (actType == AccountType.CREDIT) {

newAccount = new CreditAccount(accountNumber, initialBalance);

}

// Add the new account to the array if space is available

boolean accountAdded = false;

for (int i = 0; i < myAccounts.length; i++) {

if (myAccounts[i] == null) {

myAccounts[i] = newAccount;

accountAdded = true;

break;

}

}

// Show a message if account creation was successful or not

if (accountAdded) {

JOptionPane.showMessageDialog(this, "Account created successfully.");

} else {

JOptionPane.showMessageDialog(this, "Account could not be created. Maximum number of accounts reached.");

}

}

### ****Create an Enum Class**** BikeUses

Create a new enum class named BikeUses with the specified values:

java

Copy code

public enum BikeUses {

OFF\_ROAD,

TRACK,

ROAD,

DOWNHILL,

TRAIL

}

### b. ****Update Interfaces and Subclasses****

#### i. **Update** MountainParts **Interface**

Replace the original terrain declaration with the BikeUses enum value for off\_road.

**Original Interface:**

java

Copy code

public interface MountainParts {

// Comment out the old line if present

// String TERRAIN = "off\_road";

BikeUses TERRAIN = BikeUses.OFF\_ROAD;

String getSuspension();

void setSuspension(String newValue);

String getType();

void setType(String newValue);

}

#### ii. **Update** RoadParts **Interface**

Similarly, replace the terrain declaration with the BikeUses enum value for track.

**Original Interface:**

java

Copy code

public interface RoadParts {

// Comment out the old line if present

// String TERRAIN = "track";

BikeUses TERRAIN = BikeUses.TRACK;

String getHandlebars();

void setHandlebars(String newValue);

String getFrame();

void setFrame(String newValue);

}

#### iii. **Update** toString() **Method in Subclasses**

Ensure the toString() methods in your bike subclasses display the terrain information. For example:

**For MountainBike Subclass:**

java

Copy code

public class MountainBike extends Bike implements MountainParts {

private String suspension;

private String frameSize;

@Override

public String toString() {

return "Oracle Cycles\n" +

"This bike has Bull Horn handlebars on a Hardtail frame with 27 gears.\n" +

"It has a dropper seat with Maxxis tyres.\n" +

"This mountain bike is a Pro bike and has a RockShox XC32 suspension and a frame size of " + frameSize + ".\n" +

"This bike is best for " + TERRAIN.toString().toLowerCase();

}

@Override

public String getSuspension() {

return suspension;

}

@Override

public void setSuspension(String newValue) {

this.suspension = newValue;

}

@Override

public String getType() {

return "Mountain Bike";

}

@Override

public void setType(String newValue) {

// Method implementation

}

}

**For RoadBike Subclass:**

java

Copy code

public class RoadBike extends Bike implements RoadParts {

private String handlebars;

private String frame;

@Override

public String toString() {

return "Oracle Cycles\n" +

"This bike has " + handlebars + " handlebars on a " + frame + " frame with 27 gears.\n" +

"It has a dropper seat with Maxxis tyres.\n" +

"This road bike is a Pro bike and is best for " + TERRAIN.toString().toLowerCase();

}

@Override

public String getHandlebars() {

return handlebars;

}

@Override

public void setHandlebars(String newValue) {

this.handlebars = newValue;

}

@Override

public String getFrame() {

return frame;

}

@Override

public void setFrame(String newValue) {

this.frame = newValue;

}

}

### iv. ****Run and Test****

After making these changes, compile and run your program to ensure that:

* The MountainParts and RoadParts interfaces use the BikeUses enum correctly.
* The toString() methods in the bike subclasses correctly display the terrain information as specified.

This should ensure that the bike project now properly integrates the enum for terrain types and displays the relevant information correctly.

4o mini

**a. Create the Generic Class Cuboid**

First, create a generic class Cuboid that accepts only numeric types using the Number class. This will allow you to handle different numeric types like Double and Integer.

**Cuboid.java:**

java

Copy code

public class Cuboid<T extends Number> {

private T length;

private T breadth;

private T height;

// Constructor

public Cuboid(T length, T breadth, T height) {

this.length = length;

this.breadth = breadth;

this.height = height;

}

// Setters

public void setLength(T length) {

this.length = length;

}

public void setBreadth(T breadth) {

this.breadth = breadth;

}

public void setHeight(T height) {

this.height = height;

}

// Getters

public T getLength() {

return length;

}

public T getBreadth() {

return breadth;

}

public T getHeight() {

return height;

}

// Method to return all dimensions

@Override

public String toString() {

return "Length: " + length + ", Breadth: " + breadth + ", Height: " + height;

}

// Method to calculate volume

public double getVolume() {

return length.doubleValue() \* breadth.doubleValue() \* height.doubleValue();

}

}

**b. Create a Driver Class for Double Cuboid**

**DoubleCuboidDriver.java:**

java

Copy code

public class DoubleCuboidDriver {

public static void main(String[] args) {

// Instantiate a Cuboid with Double

Cuboid<Double> c1 = new Cuboid<>(1.3, 2.2, 2.0);

// Display dimensions

System.out.println("Cuboid with Double dimensions:");

System.out.println(c1);

// Display volume

System.out.println("Volume: " + c1.getVolume());

}

}

**c. Create a Driver Class for Integer Cuboid**

**IntegerCuboidDriver.java:**

java

Copy code

public class IntegerCuboidDriver {

public static void main(String[] args) {

// Instantiate a Cuboid with Integer

Cuboid<Integer> c2 = new Cuboid<>(1, 2, 3);

// Display dimensions

System.out.println("Cuboid with Integer dimensions:");

System.out.println(c2);

// Display volume

System.out.println("Volume: " + c2.getVolume());

}

}

**d. Modify the Cuboid Class**

The Cuboid class already uses Number as the bound for the generic type, which allows for numeric operations. The getVolume() method calculates the volume using doubleValue() to ensure that the result is computed as a double.

**e. Display the Result of getVolume()**

In the driver classes provided in steps b and c, the getVolume() method is called on the c1 and c2 objects, and the results are displayed.

**Summary**

* **Generic Cuboid Class**: Handles dimensions with any numeric type and computes the volume.
* **Driver Classes**: Instantiate Cuboid objects with Double and Integer, display dimensions, and calculate volume.

These steps should help you create and test the Cuboid class with different numeric types.