```
 <<interface>>
BinaryOperator

apply(T a, T b): T
```

1. Three ways to implement an interface.

The java.util.function.BinaryOperator interface is shown at right. **`T`** is a type parameter.

1.1 Define a class named **`Adder`** that implements this interface and adds two Double values and returns the result.

`public class Adder ________________________________________ {`

`}`

1.2 Define an *anonymous class* that implements this interface and produces one object named **`adder`**. As before, it adds the arguments and returns the result.

`BinaryOperator<Double> adder =`

1.3 Define a *lambda expression* named **`adder`** that does the same thing as 4.2

`BinaryOperator<Double> adder =`

1.4 Suppose we compute 1.0+2.0 by calling `adder.accept(1.0,2.0).`

This works because of Java's "autoboxing". How many Double objects are created?

1.5 Creating objects just to perform 1.0+2.0 is inefficient. Find an interface in the Java API that can do the same thing without using objects for the arguments or result. Write a Lambda for "adder" using this interface:

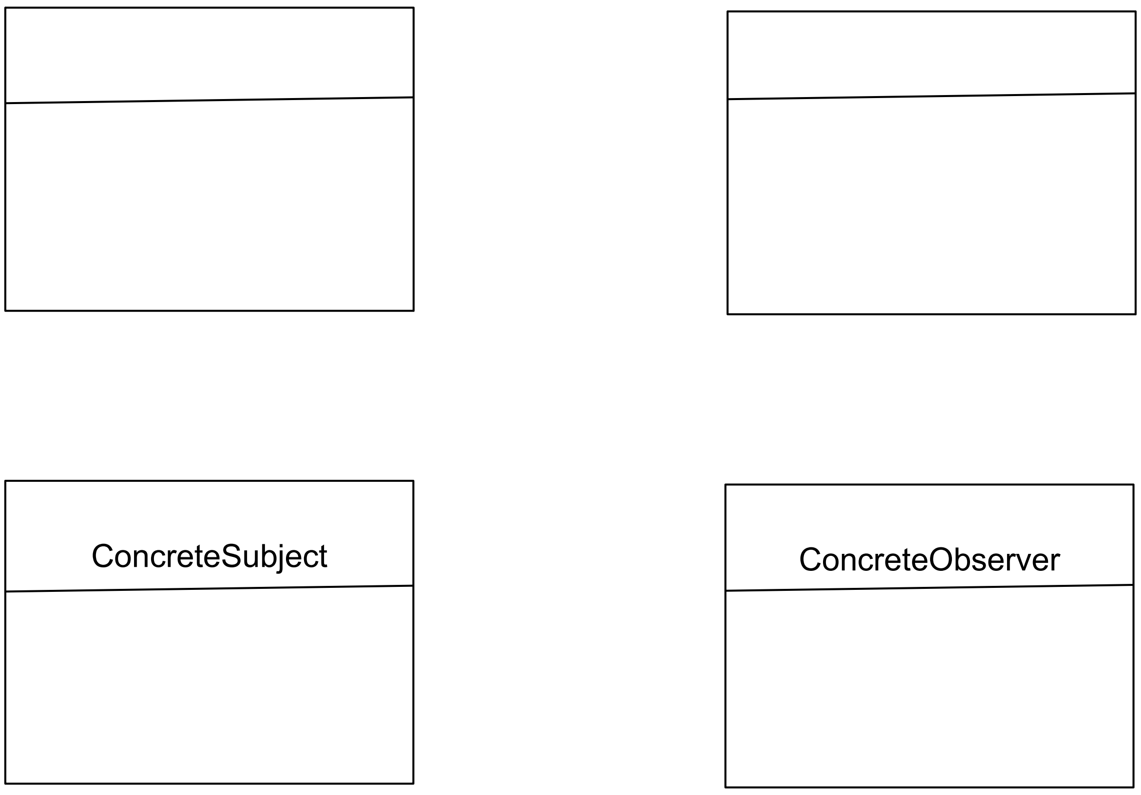
______________________________ **`adder`** =

2. Observer Pattern - complete the UML class diagram for the observer pattern in Java.

a) Add these items to the correct boxes in diagram: Observer, Observable, <<interface>>, addObserver(Observer), notifyObservers(), setChanged(), update(Observable, Object).

b) Draw UML arrows for association, implements, dependency, and inheritance. *Use correct notation.*

Assume that ConcreteObserver does *not* store a reference (attribute) to ConcreteSubject.



3. High Cohesion and Single Responsibility.

```
public class BankAccount {
     private String accountNumber;
     private Customer accountOwner;
     private Money balance;
     private TransactionLedger ledger = Bank.getTransactionLedger();
     . . .

     public void deposit(Money amount) {
          // deposit money and record transaction
     }
     public boolean withdraw(Money amount) {
          // perform withdraw and record transaction
     }
     public Money getBalance() {
          // compute and return the account balance
     }
     public boolean isActive() {
          // return true if account is active
     }
     public void printStatement(OutputStream out, LocalDate month) {
          // print monthly statement to the output stream
     }
}
```

3.1 Classes should usually strive for high cohesion by making all the methods related to the same general responsibility. Using this principle, which of the above methods does **not** belong in the BankAccount class? Justify your answer.

3.2 What other classes is BankAccount *associated with*?

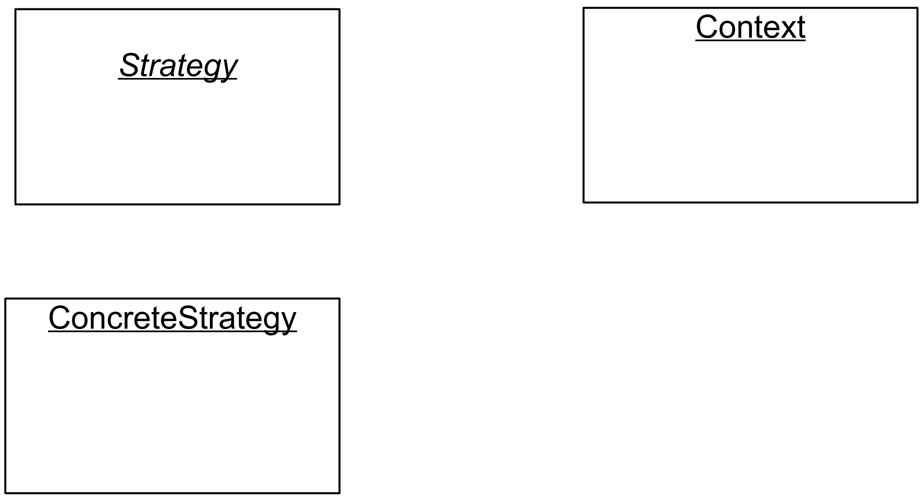
3.3 What other classes does BankAccount *depend on?* (Ignore classes in the Java API.)

4.1 In what kind of situation might the *Strategy Pattern* be useful? What the important features of an application design problem that suggest using the Strategy Pattern?

4.2 In the Strategy Pattern there are parts called the *Strategy* and *Context*. Suppose that an actual Strategy implementation is named ConcreteStrategy. Draw a UML class diagram showing:

a) relationships between the parts

b) the important methods



4.3 Another design pattern, called the *State Pattern*, is very similar to Strategy. The purpose of the State Pattern is to change an object's behavior based on its *state*. The State Pattern can greatly simplify classes whose behavior depends on state. A summary of the State Pattern is on the class home page.

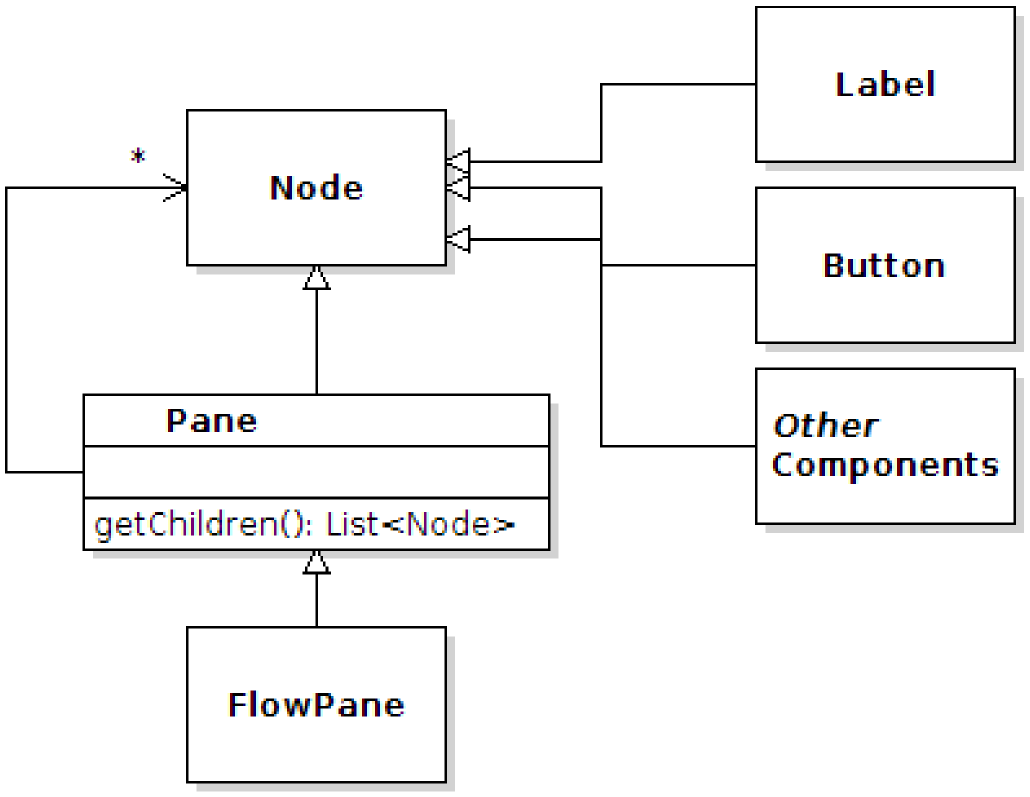
5. Draw a **sequence diagram** for this code.

```
class ItemTest {

    public void testAddItem( ) {

            Item item = new Item("111");

            item.setQuantity( 3 );

            Sale sale = new Sale( );

            sale.addItem( item );

    }
```

6. In JavaFX, a Pane is a Node that contains other Nodes. (The superclass for Pane is actually Parent, and Parent is a subclass of Node, but that is not important here.)

We can add nodes to a Pane using code like:

```
Pane pane = new Pane();
pane.getChildren().addAll( button, label, textfield, table );
```



6.1 Can we put a Pane inside another Pane? Why or why not? Give a reason.

```
// Is this possible?
Pane box = new VBox( );
FlowPane pane1 = new FlowPane();
FlowPane pane2 = new FlowPane();
box.getChildren().addAll( pane1, pane2, new Label("Hi htere") );
```

6.2 What *Design Pattern* does this design use?