Program 01:

We have two classes named `Test` and `Handling`.

For the `Handling` class:

- **Instance Variables:**
  - No instance variables.
- **Methods:**
  1. **`tiMethod(int)` : void : static**
     - In the **try block**, print "10000" and call the `ti1Method(int)` method, passing the same parameter as the `tiMethod(int)` method. After that, print "Coders".
     - In the **catch block**, accept the exception as a parameter and print the message of the exception.
  2. **`ti1Method(int)` : void : static**
     - The `ti1Method(int)` method must throw an exception.
     - In the **try block**, check an arithmetic operation (like division) using the passed parameter. If the division operation works correctly, print "Completed".
     - For example, if the parameter value is `0`, it should throw an `ArithmeticException`. This must be handled with a custom exception message.
     - If the parameter value is something like `10`, it should throw an exception as well.
     - In the **catch block**, handle the `ArithmeticException` and print an appropriate message.
     - In the **finally block**, print the message "Finally".

For the `Test` class:

- **Instance Variables:**
  - No instance variables.
- **Methods:**
  - Use the `Test` class to test your solution's classes and methods with test cases like `0`, `1`, `10`, and `15` as input.

Expected Input and Output for Program 01:

Test Case 1: Input = 0

- **Input:** 0
- **Output:**

```
10000
ArithmeticException: / by zero
Finally
Coders
```

Test Case 2: Input = 1

- **Input:** 1
- **Output:**

  ```
  10000
  Completed
  Finally
  Coders
  ```

Test Case 3: Input = 10

- **Input:** 10
- **Output:**

  ```
  10000
  Completed
  Finally
  Coders
  ```

Test Case 4: Input = 15

- **Input:** 15
- **Output:**

  ```
  10000
  Completed
  Finally
  Coders
  ```

Program 02:

We have four classes: `CarStopped`, `CarPuncture`, `CarHeat`, and `CarTest`.

For `CarStopped`:

- Extends `Exception`.
- This class is used to raise an exception if any reason, except for puncture or heat, causes the car to stop.
- **Instance Methods:**
  - No instance methods.
- **Methods:**
  - **Parameterized Constructor** with a `String` parameter.

For `CarPuncture`:

- Extends `Exception`.
- This class is used to raise an exception if the car is punctured.
- **Instance Methods:**
  - No instance methods.
- **Methods:**
  - **Parameterized Constructor** with a `String` parameter.

For `CarHeat`:

- Extends `Exception`.
- This class is used to raise an exception if the car engine temperature exceeds `50°C`.
- **Instance Methods:**
  - No instance methods.
- **Methods:**
  - **Parameterized Constructor** with a `String` parameter.

For `CarTest`:

- **Instance Variables:**
  - No instance variables.
- **Methods:**
  1. **`Stop(String): void: static`**
     - This method throws a `CarStopped` exception. If the string is "stop", throw a new exception and get the message. Otherwise, the message should be "Car not stalled".
  2. **`puncture(String): void: static`**
     - This method throws a `CarPuncture` exception. If the string is "puncture", throw a new exception and get the message: "Car is Punctured". Otherwise, the message should be "Car not punctured".
  3. **`carHeat(int): void: static`**
     - This method throws a `CarHeat` exception. If the car temperature is more than 50°C, throw a new exception with the message: "Car is heated more than 50 degrees". Otherwise, the message should be "Car not stalled".

For the `CarTest` class:

- This class contains the **main method** and is used to test your solution's classes and methods.

Expected Input and Output for Program 02:

Test Case 1: Input = "stop"

- **Input:** "stop"
- **Output:**

```
CarStopped: Car is stopped
```

Test Case 2: Input = "puncture"

- **Input:** "puncture"
- **Output:**

```
CarPuncture: Car is Punctured
```

Test Case 3: Input = 60 (for car temperature)

- **Input:** 60
- **Output:**

  ```
  CarHeat: Car is heated more than 50 degrees
  ```

Test Case 4: Input = "go" (for `Stop`)

- **Input:** "go"
- **Output:**

  ```
  Car not stalled
  ```

Test Case 5: Input = "non-puncture" (for `puncture`)

- **Input:** "non-puncture"
- **Output:**

  ```
  Car not punctured
  ```

Test Case 6: Input = 40 (for car temperature)

- **Input:** 40
- **Output:**

  ```
  Car not stalled
  ```

Program 03:

You are tasked with implementing a simple Java program that simulates bank account transactions. The program should include two custom exceptions: a checked exception (`InvalidTransactionException`) and an unchecked exception (`InsufficientFundsException`). These exceptions will be used to handle different scenarios during transactions and withdrawals.

Exceptions:

1. **InvalidTransactionException** (Checked Exception):
   - This exception should be thrown when a transaction is attempted with an invalid amount (non-positive) or when there are insufficient funds for the transaction.
2. **InsufficientFundsException** (Unchecked Exception):
   - This exception should be thrown when a withdrawal is attempted with an amount exceeding the account balance.

Class 1: BankAccount

Instance Variable:

- `balance` (double): Represents the account balance.

Constructor:

- **BankAccount(double initialBalance):** Initializes the account with the provided balance.

Methods:

1. **performTransaction(double amount):**
    - **Purpose:** Handles a transaction (deposit or withdrawal).
    - **Throws:**
        - `InvalidTransactionException`: If the transaction amount is non-positive or exceeds the account balance.
2. **withdraw(double amount):**
    - **Purpose:** Handles a withdrawal from the account.
    - **Throws:**
        - `InsufficientFundsException`: If the withdrawal amount exceeds the account balance.
        - `IllegalArgumentException`: If the withdrawal amount is non-positive.

Example of a Bank Account Class Behavior:

- If a valid transaction is performed, the balance is updated.
- If the transaction amount is invalid or insufficient funds are present, the respective exception is thrown.

Class 2: BankAccountScenario (Main Class)

Main Method:

- Create an instance of `BankAccount` with an initial balance of 1000.
- Demonstrate the use of `performTransaction` for both valid and invalid transactions, and handle the `InvalidTransactionException`.
- Demonstrate the use of `withdraw` for both valid and invalid withdrawals, and handle both `InsufficientFundsException` and `IllegalArgumentException`.

Expected Input and Output:

Test Case 1: Valid Transaction

Input:

- Call `performTransaction(500)` on the `BankAccount` instance.
- **Initial balance:** 1000

Output:

```
Transaction successful. New balance: 500
```

Test Case 2: Invalid Transaction (Negative Amount)

Input:

- Call `performTransaction(-50)` on the `BankAccount` instance.
- **Initial balance:** 1000

Output:

```
Transaction Error: Transaction amount must be positive
```

Test Case 3: Invalid Transaction (Insufficient Funds)

Input:

- Call `performTransaction(1200)` on the `BankAccount` instance.
- **Initial balance:** 1000

Output:

```
Transaction Error: Insufficient funds for the transaction
```

Test Case 4: Valid Withdrawal

Input:

- Call `withdraw(200)` on the `BankAccount` instance.
- **Initial balance:** 1000

Output:

```
Withdrawal successful. New balance: 800
```

Test Case 5: Invalid Withdrawal (Negative Amount)

Input:

- Call `withdraw(-50)` on the `BankAccount` instance.
- **Initial balance:** 1000

Output:

```
Withdrawal Error: Withdrawal amount must be positive
```

Test Case 6: Invalid Withdrawal (Insufficient Funds)

Input:

- Call `withdraw(1500)` on the `BankAccount` instance.
- **Initial balance:** 1000

Output:

```
Withdrawal Error: Insufficient funds for withdrawal
```

Test Case 7: Invalid Withdrawal (Zero Amount)

Input:

- Call `withdraw(0)` on the `BankAccount` instance.
- **Initial balance:** 1000

Output:

```
Withdrawal Error: Withdrawal amount must be positive
```
