**🔍 What is this?**

java

CopyEdit

@FunctionalInterface

interface PreparedStatementSetter {

void setParameters(PreparedStatement ps) throws SQLException;

}

This is a **functional interface**, meaning it has just **one abstract method** — perfect for use with **lambda expressions** (which you saw in the executeSearchQuery method).

**🎯 Why did we create it?**

In your searchByCountry() and searchByYearOfMinting() methods, you're using:

java

CopyEdit

ps.setString(1, countryName.toLowerCase());

or

java

CopyEdit

ps.setInt(1, year);

These differ **only in how parameters are set**.

So instead of writing similar code again and again, we want to **pass the logic for setting parameters** as a function.

**✅ Functional Interface in Java**

A **functional interface** is one that contains **exactly one abstract method**. Java provides many built-in ones like Runnable, Consumer<T>, Predicate<T>, etc.

We define our own here:

java

CopyEdit

@FunctionalInterface

interface PreparedStatementSetter {

void setParameters(PreparedStatement ps) throws SQLException;

}

* It takes a PreparedStatement
* Sets the parameters
* Might throw an SQLException

**💡 How do we use it?**

In the reusable method executeSearchQuery():

java

CopyEdit

private List<Coin> executeSearchQuery(String sql, PreparedStatementSetter setter)

We pass the query and a lambda function like:

java

CopyEdit

return executeSearchQuery(query, ps -> ps.setString(1, countryName.toLowerCase()));

Or:

java

CopyEdit

return executeSearchQuery(query, ps -> ps.setInt(1, year));

Inside executeSearchQuery() it just does:

java

CopyEdit

setter.setParameters(ps);

Which calls the lambda — boom! Parameters are set.

**🎨 Visual Example**

Imagine this:

java

CopyEdit

// Instead of doing this in every method

PreparedStatement ps = con.prepareStatement("SELECT \* FROM coins WHERE yearofminting = ?");

ps.setInt(1, 2024);

// You now do this:

executeSearchQuery(

"SELECT \* FROM coins WHERE yearofminting = ?",

ps -> ps.setInt(1, 2024)

);

Less repetition. Cleaner. More elegant. ✅

**✅ Benefits**

| **Feature** | **Why it’s good** |
| --- | --- |
| 💡 DRY Principle | No repeating parameter-setting logic |
| 💪 Flexible | Supports any number/type of parameters |
| 🔐 Safe | Handles SQLException properly |
| 🧠 Readable | Each query looks super clean and understandable |

Let me know if you want to create a similar interface for **insert/update**, or to support **multiple parameters** like country + denomination — it scales really well!

4o

**You said:**

but why we can not directly pass (query, ps.setInt(1, year)) instead of lambda function

**ChatGPT said:**

Awesome question — and it shows you're thinking deeply about **how Java handles method calls vs. function references or lambdas**. Let's break it down:

**🚫 Why This Won’t Work:**

You suggested:

java

CopyEdit

executeSearchQuery(query, ps.setInt(1, year)); // ❌

This **won’t compile**, and here's why:

**🧠 Key Difference: Method Call vs Lambda**

**❌ ps.setInt(1, year)**

This is an **immediate method call**, not a function reference or lambda.

* It **executes immediately** at that line.
* It does **not return a function** — it returns void.
* So you're effectively doing:

java

CopyEdit

executeSearchQuery(query, void); // Not valid!

Java expects a **function** (i.e., something it can call later), **not the result** of a method call.

**✅ What Works: Lambda or Method Reference**

When you do this:

java

CopyEdit

executeSearchQuery(query, ps -> ps.setInt(1, year));

You're saying:

"Here is a function that **takes a PreparedStatement** and **sets a value** when you're ready."

This is a **lambda expression**, which matches this functional interface:

java

CopyEdit

interface PreparedStatementSetter {

void setParameters(PreparedStatement ps) throws SQLException;

}

It’s equivalent to saying:

java

CopyEdit

class SomeSetter implements PreparedStatementSetter {

public void setParameters(PreparedStatement ps) {

ps.setInt(1, year); // logic here

}

}

But done in one line!

**🧪 TL;DR:**

| **Syntax** | **Meaning** | **Works?** |
| --- | --- | --- |
| ps.setInt(1, year) | Calls the method **immediately**, returns void | ❌ |
| ps -> ps.setInt(1, year) | A lambda expression — deferred execution | ✅ |
| PreparedStatement::clearParameters | A method reference, if compatible | ✅ (in some cases) |

**🔧 Analogy:**

If I say:

* ❌ “Bring me the result of turning on the light” — you bring me void.
* ✅ “Bring me a **remote** that can turn on the light later” — that's the lambda!

Let me know if you want an example of using **method references** too!

public class CoinManagement {

private Connection con;

private List<Coin> coins = new ArrayList<>();

public CoinManagement() {

try {

this.con = Connectivity.getConnection();

} catch (Exception e) {

throw new RuntimeException("Database connection failed", e);

}

}

public boolean addCoin(Coin coin) {

String insertQuery = "INSERT INTO coins (country, denomination, currentvalue, yearofminting, acquiredate) VALUES (?, ?, ?, ?, ?)";

try (PreparedStatement ps = con.prepareStatement(insertQuery)) {

ps.setString(1, coin.getCountry());

ps.setInt(2, coin.getDenomination());

ps.setDouble(3, coin.getCurrentValue());

ps.setInt(4, coin.getYearOfMinting());

ps.setDate(5, coin.getAcquireDate());

return ps.executeUpdate() == 1;

} catch (SQLException e) {

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

return false;

}

}

public List<Coin> searchByCountry(String countryName) {

String query = "SELECT \* FROM coins WHERE LOWER(country) = ?";

return executeSearchQuery(query, ps -> ps.setString(1, countryName.toLowerCase()));

}

public List<Coin> searchByYearOfMinting(int year) {

String query = "SELECT \* FROM coins WHERE yearofminting = ?";

return executeSearchQuery(query, ps -> ps.setInt(1, year));

}

// 💡 Reusable method to execute SELECT queries and map results to List<Coin>

private List<Coin> executeSearchQuery(String sql, PreparedStatementSetter setter) {

List<Coin> result = new ArrayList<>();

try (PreparedStatement ps = con.prepareStatement(sql)) {

setter.setParameters(ps);

try (ResultSet rs = ps.executeQuery()) {

while (rs.next()) {

result.add(mapResultSetToCoin(rs));

}

}

} catch (SQLException e) {

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

}

return result;

}

// 💡 Interface to pass setting logic to the helper method

@FunctionalInterface

interface PreparedStatementSetter {

void setParameters(PreparedStatement ps) throws SQLException;

}

// 💡 Mapping logic in one place

private Coin mapResultSetToCoin(ResultSet rs) throws SQLException {

return new Coin(

rs.getInt("coinid"),

rs.getString("country"),

rs.getInt("denomination"),

rs.getDouble("currentvalue"),

rs.getInt("yearofminting"),

rs.getDate("acquiredate")

);

}

}