

Class And Objects In Java

Methods and Method Signatures:

A method in Java is a block of code that performs a specific task. The method signature is a combination of the method's name, parameter list, and return type. It's like a unique identifier for the method that defines how it can be used.

Method Signature Components:

Method Name: The name of the method.

Parameters: The input values the method accepts (if any).

Return Type: The data type of the value the method returns (if any).

• }

```
SampleProgram.java > MathUtils
1  public class MathUtils {
2      // Method signature
3      public int addNumbers(int num1, int num2) {
4          // Method implementation
5          return num1 + num2;
6      }
7  }
```

In this Java example, the method addNumbers has a signature:

Method Name: addNumbers

Parameters: int num1, int num2

Return Type: int

Method Prototype

In Java, the term method prototype is often used interchangeably with method signature. It's the part of the method declaration that includes the method name, parameter list, and return type, but excludes the method body.

Purpose of Method Prototype:

- Clarifies how to interact with the method.
- Helps catch errors in method calls.
- Serves as a documentation aid for developers.

In this Java example, the method prototype for calculateArea includes:

- **Method Name:** calculateArea
- **Parameters:** None
- **Return Type:** double

```
SampleProgram.java > ...  
1  public class Circle {  
2      // Method prototype  
3      public double calculateArea() {  
4          // Method body not shown  
5      }  
6  }  
7
```

Mutator and Accessor Methods in Java

Mutator Methods (Setters):

Mutator methods, also known as setters, are used to modify the internal state of an object by changing the values of its attributes. They allow controlled changes to the object's properties, often including validation and business logic to maintain consistency. In this example, setName is a mutator method that sets the student's name while ensuring it's not empty or null.

```
SampleProgram.java > ...
1  public class Student {
2      private String name;
3
4      public void setName(String newName) {
5          if (newName != null && !newName.isEmpty()) {
6              name = newName;
7          }
8      }
9  }
0
```

Accessor Methods (Getters):

Accessor methods, also known as getters, provide controlled access to an object's attributes. They retrieve the values of attributes without allowing direct modification, thereby maintaining the encapsulation of the object's internal state .

Here, getName is an accessor method that returns the value of the student's name attribute.

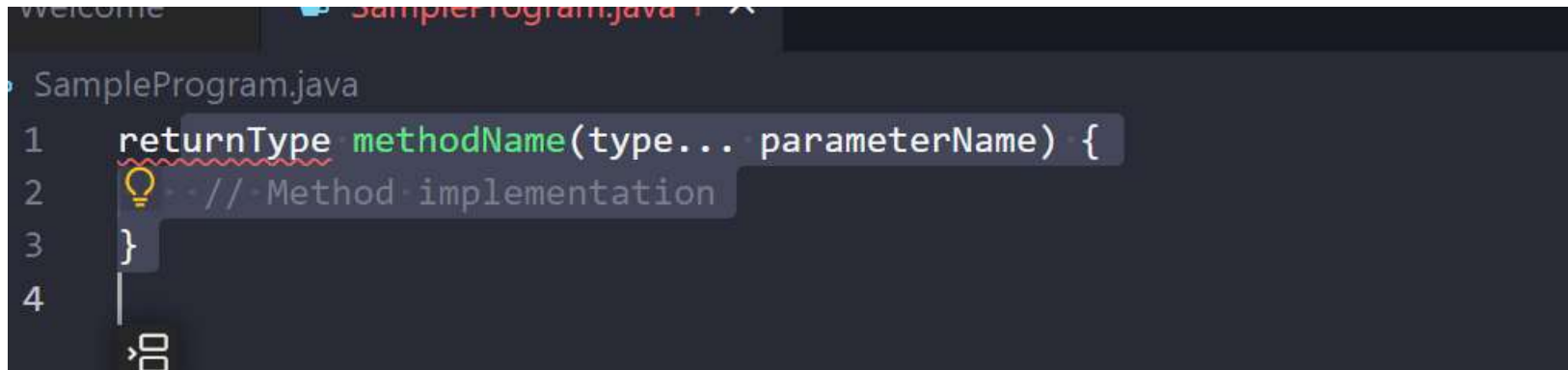
```
SampleProgram.java > ...
1  public class Student {
2      ⚡ private String name;
3
4      public String getName() {
5          return name;
6      }
7  }
```

Benefits of Mutator and Accessor Methods:

- Encapsulation: Mutators and accessors help maintain encapsulation by controlling how data is modified and accessed, preventing unauthorized or unintended changes.
- Validation: Mutators can include validation logic to ensure that data changes adhere to predefined rules.
- Flexibility: Accessors provide a controlled way to retrieve data, allowing you to make changes to the internal representation without affecting external code.
- Data Integrity: Using mutators and accessors reduces the risk of invalid or inconsistent data.
- Security: By exposing only necessary attributes and controlling their access, you enhance security.

Varargs (Variable-Length Argument):

In Java, varargs, short for "variable-length arguments," allow methods to accept a variable number of arguments of the same type. This flexibility simplifies method calls and enables more concise and versatile code.



```
SampleProgram.java
1  returnType methodName(type... parameterName) {
2  // Method implementation
3  }
4  |
  >
```

The screenshot shows a code editor window titled 'SampleProgram.java'. The code defines a method with the signature `returnType methodName(type... parameterName) {`. The `type...` part is highlighted with a light blue background, indicating it represents a variable-length argument (vararg). Below the opening curly brace, there is a comment `// Method implementation`. The code is numbered 1 through 4 on the left margin. A lightbulb icon is visible next to the comment line, and a cursor is at the end of the line below the closing brace.

```
int result = findMax(5, 8, 2, 10);
```

```
int anotherResult = findMax(15, 3, 7, 22, 12, 9);
```

With varargs, you can call this method with any number of arguments:



```
SampleProgram.java
1  public int findMax(int... numbers) {
2      int max = Integer.MIN_VALUE;
3      for (int num : numbers) {
4          if (num > max) {
5              max = num;
6          }
7      }
8      return max;
9  }
10
```

Understanding the hashCode() Method:

The hashCode() method in Java is used to generate a unique integer representation (hash code) for an object. This hash code is primarily employed by data structures like hash maps and hash sets to optimize storage and retrieval operations.

//Sample code snippet for this output is given next page

Hash code for person1: -704684809

Hash code for person2: 44249406

```

1  import java.util.Objects;
2
3  public class Person {
4      private String name;
5      private int age;
6
7      public Person(String name, int age) {
8          this.name = name;
9          this.age = age;
10     }
11
12     @Override
13     public int hashCode() {
14         return Objects.hash(name, age);
15     }
16
17     Run | Debug
18     public static void main(String[] args) {
19         Person person1 = new Person(name:"Alice", age:25);
20         Person person2 = new Person(name:"Bob", age:30);
21
22         int hashCode1 = person1.hashCode();
23         int hashCode2 = person2.hashCode();
24
25         System.out.println("Hash code for person1: " + hashCode1);
26         System.out.println("Hash code for person2: " + hashCode2);
27     }
28 }

```

Understanding the toString() Method:

The toString() method returns a string representation of an object, providing a human-readable way to describe the object's state. It's commonly used for debugging and logging purposes.

```
SampleProgram.java
1  @Override
2  public String toString() {
3      return "Object details: attribute1=" + attribute1 + ", attribute2=" + attribute2;
4  }
5
```

Mutable Objects:

Mutable objects are objects whose state can be modified after creation. They offer flexibility but need careful handling to avoid unintended side effects.

Characteristics:

- 1.Their state can change throughout their lifecycle.
- 2.Can be more complex to manage than immutable objects.
- 3.Prone to concurrency issues if not synchronized properly in multithreaded environments

In this example, the name attribute of the Person class can be changed using the setName method.

```
public class Person {  
    ⚡ private String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String newName) {  
        this.name = newName;  
    }  
}
```

User-Defined Mutable Objects:

User-defined mutable objects are custom classes you create that have mutable properties. Managing their state requires careful consideration to ensure consistency and avoid unexpected behavior.

In this example, the balance attribute of the BankAccount class can be modified using the deposit method.

For code refer next page:

SimpleProgram.java > ...

```
public class BankAccount {  
    private String accountNumber;  
    private double balance;  
  
    public BankAccount(String accountNumber, double balance) {  
        this.accountNumber = accountNumber;  
        this.balance = balance;  
    }  
  
    public String getAccountNumber() {  
        return accountNumber;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
  
    public void deposit(double amount) {  
        if (amount > 0) {  
            balance += amount;  
        }  
    }  
}
```

Immutable Objects:

Immutable objects are objects whose state cannot be changed after they are created. They offer advantages in terms of simplicity, thread safety, and predictability.

Characteristics:

- Once created, their state remains constant.
- Any "modification" results in a new object.
- Suitable for sharing data across threads without synchronization.