DEV GENESIS

VIDEO 6: FUNCTIONS & METHODS

Reuse and modularize code

WHAT WE'LL COVER

- What are methods and why use them?
- Method syntax in Java
- Parameters and arguments
- Return types
- Method overloading
- Method calling process
- Best practices

WHAT ARE METHODS?

Methods are **named blocks of code** that perform specific tasks.
Key benefits of methods:

- **Code reusability** - Write once, use many times
- **Modularity** - Break complex problems into smaller, manageable parts
- **Readability** - Descriptive method names make code easier to understand
- **Maintainability** - Change implementation in one place

> In Java, the terms "method" and "function" are often used interchangeably, but technically everything in Java is a method since it's part of a class.

METHOD SYNTAX

Basic components of a method declaration:

```
accessModifier returnType methodName(parameterType
parameterName) {
    // Method body - code to be executed

    return value;  // Optional, depending on return
type
}
```

public static int sum(int a, int b) {
// Method body
int result = a + b;
return result;
}

- **Access modifier**: public
- **Modifier**: static
- **Return type**: int
- **Method name**: sum
- **Parameters**: int a, int b
- **Return statement**: return result;

METHOD PARAMETERS

Parameters are variables that receive values when the method is called:

```
public static void greet(String name, int age) {
    System.out.println("Hello, " + name + "! You
are " + age + " years old.");
}
```

Calling the method with arguments:

```
// Calling the greet method
greet("Alice", 25);    // Output: Hello, Alice! You
are 25 years old.
greet("Bob", 30);      // Output: Hello, Bob! You
are 30 years old.
```

> **Parameters** are in the method declaration. **Arguments** are the actual values passed to the method.

RETURN TYPES

Methods can return a value to the caller using the `return` statement:

Return Type	Description	Example
`void`	No return value	`public void printMessage() { ... }`
Primitive type	Returns a value of the specified type	`public int sum(int a, int b) { ... }`
Reference type	Returns an object reference	`public String getFullName() { ... }`
Array	Returns an array	`public int[] getNumbers() { ... }`

> The `return` statement immediately exits the method, returning control to the caller.

METHOD CALLING PROCESS

main() method

```
int result = isEven(42);
System.out.println(result);
```

isEven() method

```
boolean isEven(int num) {
  return num % 2 == 0;
}
```

1. Program execution starts in `main()`
2. When the method call is encountered, control transfers to `isEven()`
3. The `isEven()` method executes and returns a result
4. Control returns to `main()` with the returned value
5. Execution continues in `main()`

METHOD OVERLOADING

Java allows multiple methods with the same name but different parameters:

```
// Calculate area of a rectangle
public static double calculateArea(double length,
double width) {
    return length * width;
}

// Calculate area of a circle
public static double calculateArea(double radius) {
    return Math.PI * radius * radius;
}

// Usage:
double rectangleArea = calculateArea(5.0, 3.0);  //
Calls the first method
double circleArea = calculateArea(4.0);          //
Calls the second method
```

> Methods can be overloaded by changing the number of parameters, their types, or their order. Return type alone is not enough for overloading.

CODE EXAMPLE: EVEN OR ODD

```
public class EvenOddChecker {
    public static void main(String[] args) {
        // Test with different numbers
        checkAndPrint(7);
        checkAndPrint(42);

        // Alternative approach using a method that
returns a boolean
        int number = 15;
        if (isEven(number)) {
            System.out.println(number + " is
even");
        } else {
            System.out.println(number + " is odd");
        }
    }

    // Method that checks and prints the result
    public static void checkAndPrint(int number) {
        if (number % 2 == 0) {
            System.out.println(number + " is
even");
        } else {
            System.out.println(number + " is odd");
        }
    }

    // Method that returns a boolean result
    public static boolean isEven(int number) {
        return number % 2 == 0;
    }
```

```
}
```

Output:

```
7 is odd
42 is even
15 is odd
```

THE MAIN METHOD

The `main` method is the entry point for Java applications:

```
public static void main(String[] args) {
    // Program execution starts here
}
```

Key characteristics:

- **public** - Accessible from anywhere
- **static** - Can be called without creating an object
- **void** - Doesn't return a value
- **String[] args** - Command-line arguments

> The JVM looks specifically for this method signature to start your program.

PARAMETER PASSING

Java uses "pass-by-value" for all parameters:

PRIMITIVE TYPES

A copy of the value is passed. Changes to the parameter do not affect the original variable.

```
void modifyValue(int x) {
    x = x + 10;  // Changes only the local copy
}

int num = 5;
modifyValue(num);
System.out.println(num);  // Still prints 5
```

REFERENCE TYPES

A copy of the reference is passed. The method can modify the object's state.

```
void modifyArray(int[] arr) {
    arr[0] = 100;  // Modifies the actual array
}

int[] numbers = {1, 2, 3};
modifyArray(numbers);
```

```
System.out.println(numbers[0]);  // Prints 100
```

METHOD BEST PRACTICES

- Use **descriptive names** that indicate what the method does
- Follow the **Single Responsibility Principle** - each method should do one thing well
- Keep methods **short and focused** (typically under 20-30 lines)
- Use **comments** to explain complex logic or non-obvious functionality
- Be consistent with **parameter ordering**
- Validate **input parameters** to prevent errors
- Use **meaningful return values** or throw appropriate exceptions

METHOD EXAMPLES

```
// Calculate the average of an array of numbers
public static double calculateAverage(int[]
numbers) {
    if (numbers == null || numbers.length == 0) {
        return 0.0;  // Handle edge case
    }

    int sum = 0;
    for (int number : numbers) {
        sum += number;
    }

    return (double) sum / numbers.length;
}

// Check if a year is a leap year
public static boolean isLeapYear(int year) {
    // Leap year is divisible by 4
    // But if it's divisible by 100, it must also
be divisible by 400
    return year % 4 == 0 && (year % 100 != 0 ||
year % 400 == 0);
}
```

> Notice how the method names clearly indicate what they do, and they handle edge cases.

UNDERSTANDING THE CALL STACK

When methods call other methods, Java uses a call stack:

```
public static void main(String[] args) {
    System.out.println("Starting main");
    methodA();
    System.out.println("Back in main");
}

public static void methodA() {
    System.out.println("In methodA");
    methodB();
    System.out.println("Back in methodA");
}

public static void methodB() {
    System.out.println("In methodB");
}
```

Output:

```
Starting main
In methodA
In methodB
Back in methodA
Back in main
```

PALINDROME CHECKER EXAMPLE

```
public class PalindromeChecker {
    public static void main(String[] args) {
        // Test with various strings
        String[] testStrings = {
            "racecar",
            "hello",
            "Madam",
            "A man, a plan, a canal: Panama",
            "12321"
        };

        for (String str : testStrings) {
            if (isPalindrome(str)) {
                System.out.println("\"" + str + "\"
is a palindrome");
            } else {
                System.out.println("\"" + str + "\"
is NOT a palindrome");
            }
        }
    }

    // Method to check if a string is a palindrome
    public static boolean isPalindrome(String str)
{
        // Remove non-alphanumeric characters and
convert to lowercase
        String cleaned = str.replaceAll("[^a-zA-Z0-
9]", "").toLowerCase();

        // Check if the cleaned string is the same
forward and backward
        int left = 0;
```

```
        int right = cleaned.length() - 1;

        while (left < right) {
            if (cleaned.charAt(left) !=
cleaned.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }

        return true;
    }
}
```

INTERVIEW INSIGHT

COMMON INTERVIEW QUESTIONS:

1. "What is method overloading and overriding?"
2. "How do you handle method parameters in Java?"
3. "Explain the difference between pass-by-value and pass-by-reference"
4. "What are best practices for writing methods?"

Key points to remember:

- Methods should have a single responsibility
- Parameter validation is crucial for robust code
- Java always uses pass-by-value
- Method names should be descriptive and follow conventions

YOUR CODING EXERCISE

👨‍💻 Write a method to check if a string is a palindrome:

1. Create a new Java class named `PalindromeExercise`
2. Implement a method `isPalindrome` that takes a String and returns a boolean
3. The method should check if the string reads the same forward and backward
4. Ignore case (treat 'A' and 'a' as the same)
5. Ignore non-alphanumeric characters (spaces, punctuation, etc.)
6. Test your method with several examples like "racecar", "A man, a plan, a canal: Panama", and "hello"

SOLUTION OUTLINE

```
public class PalindromeExercise {
    public static void main(String[] args) {
        // Test cases

System.out.println(isPalindrome("racecar"));
// true
        System.out.println(isPalindrome("hello"));
// false
        System.out.println(isPalindrome("A man, a
plan, a canal: Panama")); // true
        System.out.println(isPalindrome("Was it a
car or a cat I saw?"));   // true
    }

    public static boolean isPalindrome(String str)
{
        // Remove non-alphanumeric characters and
convert to lowercase
        String cleaned = str.replaceAll("[^a-zA-Z0-
9]", "").toLowerCase();

        // Option 1: Compare with reversed string
        String reversed = new
StringBuilder(cleaned).reverse().toString();
        return cleaned.equals(reversed);

        /* Option 2: Two-pointer approach
        int left = 0;
        int right = cleaned.length() - 1;

        while (left < right) {
            if (cleaned.charAt(left) !=
cleaned.charAt(right)) {
```

```
                return false;
            }
            left++;
            right--;
        }

        return true;
        */
    }
}
```

Both solution approaches work well. The first uses Java's StringBuilder for clarity, the second is more efficient.