Topics Done

30 October 2023 13:17

1. What is Java, Why Java?
2. JDK vs JVM vs JRE
3. Variables
4. Strings
5. If-Statements
6. Loops
7. User Input
8. Arrays
9. Multidimensional arrays and nested for loops
10. Classes, Methods, and Objects
11. Return Types & Method Parameters
12. Packages
13. Constructors and the usage of this and this()
14. Static and Final
15. Access Modifiers
16. Singleton Class
17. Inheritance
18. Polymorphism
19. Encapsulation
20. Abstraction v/s Interface
21. Multiple Inheritance
22. Casting Numerical Values
23. Wrapper Class
24. Inner Class
25. Anonymous Classes
26. String Builder
27. toString()
28. Equals & Hash Code of Objects
29. Ternary Operator
30. Handling Exceptions
31. Passing by Value vs Pass by reference
32. Enums
33. Collections
34. ArrayList & Iterator
35. Stack
36. Queue & LinkedList
37. PriorityQueue
38. ArrayDeque
39. HashSet
40. LinkedHashSet
41. TreeSet
42. HashMap
43. TreeMap
44. ArrayClass
45. Collection Class
46. Thread
47. Runnable
48. Race Condition
49. Thread State
50. Class Loaders
51. Streams

52. Enums
53. Annotations
54. Lambda Expressions
55. Functional Interface
56. Date & Time Packages
57. Memory management - Heap vs stack
58. Best Practices

Variables

30 October 2023 13:05

Sure, I can explain the concepts of variables in Java for Oracle Certified Professional Java Programmer (OCPJP) certification. Variables are fundamental elements in Java that store data. In Java, variables have various types and rules associated with them. Let's cover these concepts with code examples.

1. **Variable Types**:
   In Java, there are several variable types:

   - **Primitive Data Types**: These are basic data types and include `int`, `double`, `char`, `boolean`, etc. They store simple values.
   - **Reference Data Types**: These include classes, interfaces, arrays, and enums.

2. **Declaring Variables**:
   To declare a variable, you specify its type and name.

   ```java
   int age;          // Declaration of an integer variable
   double salary;    // Declaration of a double variable
   String name;      // Declaration of a String reference
   ```

3. **Initializing Variables**:
   Variables can be initialized at the time of declaration.

   ```java
   int age = 30;         // Initialization of an integer variable
   double salary = 5000;  // Initialization of a double variable
   String name = "John";  // Initialization of a String reference
   ```

4. **Variable Scope**:
   Variables have a scope, which defines where they are accessible.

   ```java
   int x = 10;          // This variable is accessible in the entire method
   if (true) {
      int y = 20;       // This variable is only accessible within the if block
   }
   ```

5. **Instance and Local Variables**:
   - **Instance Variables**: These are declared within a class but outside any method. They are used to store object-specific data.
   - **Local Variables**: These are declared within methods or code blocks and have limited scope.

   ```java
   public class Person {
      String name;   // Instance variable
      int age;       // Instance variable

      public void doSomething() {
   ```

```
        int x = 5;   // Local variable
        // ...
    }
}
```

6. **Final Variables**:
   A `final` variable cannot be changed after it's initialized.

   ```java
   final int constantValue = 100;
   ```

7. **Static Variables**:
   Static variables are shared among all instances of a class and are declared with the `static` keyword.

   ```java
   public class Counter {
       static int count = 0;  // Static variable
   }
   ```

8. **Naming Conventions**:
   Variable names must start with a letter or underscore, followed by letters, digits, or underscores. They are case-sensitive.

   ```java
   int myVariable;
   int _variable;
   int variable123;
   ```

9. **Type Casting**:
   You can cast variables to different types when needed.

   ```java
   int a = 10;
   double b = (double) a; // Casting 'a' to a double
   ```

10. **Literals**:
    Literal values are hard-coded data values assigned to variables.

    ```java
    int num = 42;       // Integer literal
    double price = 19.99;  // Double literal
    char grade = 'A';   // Character literal
    ```

These are the fundamental concepts related to variables in Java. Understanding these concepts and applying them correctly is essential for Java programming and can be tested on the OCPJP certification exam.

Strings

30 October 2023 13:45

Certainly, I can explain the necessary concepts of Strings in Java for the Oracle Certified Professional Java Programmer (OCPJP) certification. Strings are a fundamental and important data type in Java for representing text. Here are the key concepts and some code examples:

1. **String Declaration**:
   To declare a String variable, you can use the `String` class.

   ```java
   String greeting; // Declaring a String variable
   ```

2. **String Initialization**:
   Strings can be initialized using literals or the `new` keyword.

   ```java
   String name = "Alice"; // Initialization using a literal
   String address = new String("123 Main St."); // Initialization using the 'new' keyword
   ```

3. **String Concatenation**:
   You can concatenate strings using the `+` operator.

   ```java
   String firstName = "John";
   String lastName = "Doe";
   String fullName = firstName + " " + lastName; // Concatenation
   ```

4. **String Methods**:
   The `String` class provides numerous methods to manipulate strings, such as `length()`, `charAt()`, `substring()`, `indexOf()`, and many more.

   ```java
   String text = "Hello, World!";
   int length = text.length(); // Length of the string
   char firstChar = text.charAt(0); // Access character at index 0
   String sub = text.substring(7); // Substring starting at index 7
   int index = text.indexOf("World"); // Find the index of a substring
   ```

5. **String Immutability**:
   Strings in Java are immutable, meaning their values cannot be changed. When you modify a string, it creates a new string.

   ```java
   String original = "Hello";
   String modified = original + ", World!"; // Creates a new string
   ```

6. **String Comparison**:
   Use `.equals()` or `.equalsIgnoreCase()` to compare string content. `==` compares references, not

content.
Study Run and Strings programs for more

```java
String str1 = "Hello";
String str2 = "Hello";
boolean isEqual = str1.equals(str2); // Compares content
```

7. **String Formatting**:
   You can format strings using `String.format()` or `printf()`.

```java
String formatted = String.format("My name is %s and I am %d years old.", "Alice", 25);
System.out.printf("Formatted: %s%n", formatted);
```

8. **String Literal Pool**:
   String literals are stored in a pool. When you create a string with a literal, Java may reuse an existing one from the pool. Study Run and Strings programs for more

```java
String a = "Hello";
String b = "Hello"; // Reuses the existing "Hello" from the pool
boolean sameObject = (a == b); // true
```

9. **StringBuilder and StringBuffer**:
   For mutable string manipulation, use `StringBuilder` or `StringBuffer` to avoid the overhead of creating new string objects.

```java
StringBuilder builder = new StringBuilder("Hello");
builder.append(", World!"); // Mutable string
String result = builder.toString(); // Convert back to String
```

10. **String Splitting**:
    You can split a string into an array of substrings using the `split()` method.

```java
String sentence = "This is a sample sentence";
String[] words = sentence.split(" "); // Splits the sentence into words
```

Understanding these string concepts and methods is crucial for Java programming and for success in the OCPJP certification exam.

If-Statements

02 November 2023 14:28

Sure, I can explain the necessary concepts of if-statements in Java for the Oracle Certified Professional Java Programmer (OCPJP) certification. If-statements are fundamental control structures in Java that allow you to make decisions in your code. Here are the key concepts and some code examples:

1. **Basic if-statement**:
   An if-statement allows you to execute a block of code if a condition is true.

   ```java
   int number = 5;

   if (number > 0) {
       System.out.println("Number is positive.");
   }
   ```

2. **if-else statement**:
   You can use an else block to specify code to be executed when the condition is false.

   ```java
   int number = -2;

   if (number > 0) {
       System.out.println("Number is positive.");
   } else {
       System.out.println("Number is non-positive.");
   }
   ```

3. **else-if statement**:
   You can use else-if blocks to handle multiple conditions sequentially.

   ```java
   int number = 0;

   if (number > 0) {
       System.out.println("Number is positive.");
   } else if (number < 0) {
       System.out.println("Number is negative.");
   } else {
       System.out.println("Number is zero.");
   }
   ```

4. **Nested if-statements**:
   You can nest if-statements inside each other for more complex condition handling.

   ```java
   int x = 10;
   int y = 20;
   ```

```
if (x > 0) {
    if (y > 0) {
        System.out.println("Both x and y are positive.");
    }
}
```

5. **Logical operators**:
   You can use logical operators (`&&` for AND, `||` for OR, `!` for NOT) to combine conditions.

```java
int age = 25;
boolean isStudent = false;

if (age > 18 && !isStudent) {
    System.out.println("You are eligible for a driver's license.");
}
```

6. **Ternary operator**:
   The ternary operator is a compact way to express a simple if-else statement.

```java
int number = 7;
String result = (number % 2 == 0) ? "Even" : "Odd";
System.out.println("The number is " + result);
```

7. **Switch statement**:
   A switch statement is used when you have multiple conditions based on the value of an expression.

```java
int day = 2;

switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    // ... other cases ...
    default:
        System.out.println("Invalid day");
}
```

8. **Comparing objects**:
   When comparing objects, use the `.equals()` method (for objects like strings) rather than `==`.
Study Run and Strings programs for more

```java
String str1 = "Hello";
String str2 = "Hello";

if (str1.equals(str2)) {
```

```
    System.out.println("Both strings are equal.");
}
```

Understanding these if-statement concepts is important for decision-making in your Java programs and for success in the OCPJP certification exam.

Loops

03 November 2023 10:38

Certainly, I can explain the necessary concepts of loops (while, for, and do-while) in Java for the Oracle Certified Professional Java Programmer (OCPJP) certification. Loops are fundamental control structures in Java used to execute a block of code repeatedly. Here are the key concepts and code examples for each type of loop:

1. **While Loop**:
   - A `while` loop repeatedly executes a block of code as long as a condition is true.
   - It checks the condition before executing the loop.

   ```java
   int count = 1;
   while (count <= 5) {
       System.out.println("Count: " + count);
       count++;
   }
   ```

2. **For Loop**:
   - A `for` loop provides a concise way to express loops with initialization, condition, and increment in a single line.
   - It's often used when you know the number of iterations in advance.

   ```java
   for (int i = 1; i <= 5; i++) {
       System.out.println("Count: " + i);
   }
   ```

3. **Do-While Loop**:
   - A `do-while` loop is similar to a `while` loop but guarantees that the loop body is executed at least once before checking the condition.
   - It checks the condition after executing the loop.

   ```java
   int count = 1;
   do {
       System.out.println("Count: " + count);
       count++;
   } while (count <= 5);
   ```

4. **Break Statement**:
   - The `break` statement is used to exit a loop prematurely based on a certain condition.

   ```java
   for (int i = 1; i <= 10; i++) {
       if (i == 5) {
           break; // Exit the loop when i is 5
       }
       System.out.println("Count: " + i);
   }
   ```

5. **Continue Statement**:
   - The `continue` statement is used to skip the current iteration and move to the next iteration of a loop.

   ```java
   for (int i = 1; i <= 5; i++) {
       if (i == 3) {
           continue; // Skip the iteration when i is 3
       }
       System.out.println("Count: " + i);
   }
   ```

6. **Nested Loops**:
   - You can nest loops within other loops to handle more complex scenarios.

   ```java
   for (int i = 1; i <= 3; i++) {
       for (int j = 1; j <= 2; j++) {
           System.out.println("i: " + i + ", j: " + j);
       }
   }
   ```

7. **Enhanced for-each Loop (for-each)**:
   - The enhanced for loop is used to iterate over arrays and collections.

   ```java
   int[] numbers = {1, 2, 3, 4, 5};
   for (int num : numbers) {
       System.out.println("Number: " + num);
   }
   ```

Understanding these loop concepts is crucial for creating efficient and well-structured Java programs and for success in the OCPJP certification exam.

User Input

03 November 2023 11:18

Certainly, I can explain the necessary concepts of user input from the console in Java for the Oracle Certified Professional Java Programmer (OCPJP) certification. User input is an important part of interactive Java programs, and you can achieve it using the `java.util.Scanner` class. Here are the key concepts and code examples:

1. **Importing the Scanner class**:
   To use the `Scanner` class for user input, you need to import it at the beginning of your Java file.

   ```java
   import java.util.Scanner;
   ```

2. **Creating a Scanner object**:
   To get user input, you create a `Scanner` object that is associated with the standard input stream (`System.in`).

   ```java
   Scanner scanner = new Scanner(System.in);
   ```

3. **Reading Integers**:
   You can use the `nextInt()` method to read integers entered by the user.

   ```java
   System.out.print("Enter an integer: ");
   int number = scanner.nextInt();
   System.out.println("You entered: " + number);
   ```

4. **Reading Doubles**:
   Use the `nextDouble()` method to read double-precision floating-point numbers.

   ```java
   System.out.print("Enter a double: ");
   double value = scanner.nextDouble();
   System.out.println("You entered: " + value);
   ```

5. **Reading Strings**:
   You can use the `next()` method to read a single word or the `nextLine()` method to read a whole line of text.

   ```java
   System.out.print("Enter your name: ");
   String name = scanner.next();
   System.out.println("Hello, " + name);
   ```

6. **Error Handling**:
   Always handle exceptions when reading input to avoid program crashes. For example, you can use `hasNext()` to check for input availability.

```java
if (scanner.hasNextInt()) {
    int number = scanner.nextInt();
    System.out.println("You entered: " + number);
} else {
    System.out.println("Invalid input. Please enter an integer.");
}
```

7. **Closing the Scanner**:
   It's good practice to close the `Scanner` when you're done using it.

```java
scanner.close();
```

Here's a full example that reads an integer and a string from the user:

```java
import java.util.Scanner;

public class UserInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int number = scanner.nextInt();
        System.out.println("You entered: " + number);

        scanner.nextLine(); // Consume the newline character

        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name);

        scanner.close();
    }
}
```

Understanding how to obtain user input from the console is essential for building interactive Java programs and is likely to be tested in the OCPJP certification exam.

Arrays

03 November 2023 11:58

Certainly, I can explain the necessary concepts of arrays in Java for the Oracle Certified Professional Java Programmer (OCPJP) certification. Arrays are data structures that allow you to store multiple values of the same data type. Here are the key concepts and code examples for working with arrays in Java:

1. **Array Declaration**:
   To declare an array, you specify the data type, followed by square brackets `[]`, and then the array name.

   ```java
   int[] numbers; // Declaration of an integer array
   ```

2. **Array Initialization**:
   Arrays can be initialized in a few ways:

   - **Static Initialization**:
     ```java
     int[] numbers = {1, 2, 3, 4, 5}; // Initialization with values
     ```

   - **Dynamic Initialization**:
     ```java
     int[] numbers = new int[5]; // Initialization with size
     ```

3. **Accessing Elements**:
   Array elements are accessed using an index, which starts at 0 for the first element.

   ```java
   int[] numbers = {1, 2, 3, 4, 5};
   int firstElement = numbers[0]; // Accessing the first element
   int thirdElement = numbers[2]; // Accessing the third element
   ```

4. **Array Length**:
   You can obtain the length of an array using the `length` property.

   ```java
   int[] numbers = {1, 2, 3, 4, 5};
   int arrayLength = numbers.length;
   ```

5. **Iterating Over Arrays**:
   You can use loops (e.g., `for` or `foreach`) to iterate over the elements of an array.

   ```java
   int[] numbers = {1, 2, 3, 4, 5};
   for (int i = 0; i < numbers.length; i++) {
       System.out.println(numbers[i]);
   }
   ```

```

6. **Multi-Dimensional Arrays**:
   Java supports multi-dimensional arrays, including 2D arrays, which can be thought of as an array of arrays.

   ```java
   int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
   int element = matrix[1][2]; // Accessing an element in a 2D array
   ```

7. **Array Copying**:
   You can copy the contents of one array to another using the `System.arraycopy` method or `Arrays.copyOf` method.

   ```java
   int[] sourceArray = {1, 2, 3};
   int[] targetArray = new int[sourceArray.length];
   System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
   ```

8. **Arrays as Objects**:
   In Java, arrays are objects, and they are stored on the heap. The array variable stores a reference to the actual array object.

   ```java
   int[] numbers = {1, 2, 3};
   int[] anotherArray = numbers; // both variables point to the same array
   ```

9. **Enhanced For-Each Loop**:
   This loop simplifies array traversal.

   ```java
   int[] numbers = {1, 2, 3, 4, 5};
   for (int number : numbers) {
      System.out.println(number);
   }
   ```

10. **Arrays of Objects**:
    Arrays can hold references to objects, including user-defined classes.

    ```java
    String[] names = {"Alice", "Bob", "Charlie"};
    ```

11. **Sorting Arrays**:
    You can use `Arrays.sort()` for sorting arrays of primitive data types and `Collections.sort()` for sorting arrays of objects.

    ```java
    int[] numbers = {5, 2, 8, 1, 9};
    Arrays.sort(numbers);
    ```

Understanding these array concepts and how to work with arrays is important for many Java
    ```

programming tasks and is likely to be tested on the OCPJP certification exam.

Multidimensional arrays and nested for loops

06 November 2023 11:38

Certainly, I can explain the necessary concepts of multidimensional arrays and nested for loops in Java for the Oracle Certified Professional Java Programmer (OCPJP) certification. Multidimensional arrays are arrays of arrays, and they are commonly used to represent tables, matrices, or other multi-dimensional data structures. Here are the key concepts and code examples:

Multidimensional Arrays:
In Java, you can have arrays with more than one dimension. A common example is a 2D array, which is essentially an array of arrays. You can extend this concept to create 3D arrays, 4D arrays, and so on. Here's how to work with 2D arrays:

1. **Declaration and Initialization**:
   To declare and initialize a 2D array, you specify the number of rows and columns.

   ```java
   int[][] matrix = new int[3][3]; // 3x3 integer matrix
   ```

   You can also initialize a 2D array with values:

   ```java
   int[][] matrix = {
       {1, 2, 3},
       {4, 5, 6},
       {7, 8, 9}
   };
   ```

2. **Accessing Elements**:
   Elements in a 2D array are accessed by specifying both the row and column indices.

   ```java
   int value = matrix[1][2]; // Accesses the element in the second row and third column (value is 6)
   ```

3. **Iterating Over a 2D Array**:
   Nested for loops are typically used to traverse and manipulate elements in a 2D array.

   ```java
   for (int i = 0; i < matrix.length; i++) {
       for (int j = 0; j < matrix[i].length; j++) {
           System.out.print(matrix[i][j] + " ");
       }
       System.out.println(); // Move to the next row
   }
   ```

Nested For Loops:
Nested for loops are used when you need to iterate over multiple dimensions or perform complex iterations. They are often used in conjunction with multidimensional arrays:

1. **Example 1 - Nested Loop for Rows and Columns**:

In the example above, we saw a nested loop for iterating over rows and columns in a 2D array. This pattern is very common.

```java
for (int i = 0; i < numRows; i++) {
    for (int j = 0; j < numColumns; j++) {
        // Perform some action for each element in the matrix
    }
}
```

2. **Example 2 - Nested Loop for Patterns**:
Nested loops can be used to print patterns, such as stars or numbers. For instance, to print a right-angled triangle of stars:

```java
int n = 5;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print("* ");
    }
    System.out.println();
}
```

3. **Example 3 - Combining Nested Loops with Conditions**:
You can use nested loops in conjunction with conditions to perform more complex tasks.

```java
for (int i = 1; i <= numRows; i++) {
    for (int j = 1; j <= numColumns; j++) {
        if (i == j) {
            // Perform some action for diagonal elements
        }
    }
}
```

Understanding how to work with multidimensional arrays and nested for loops is important for tasks that involve complex data structures or require iterating over multiple dimensions. This knowledge is also valuable for the OCPJP certification exam.

Classes, Methods & Objects

27 November 2023 14:48

Certainly! Understanding the concepts of classes, methods, and objects is fundamental to Java programming and is important for the Oracle Certified Professional Java Programmer (OCPJP) certification. Let's go through each concept with code examples:

1. Classes:
A class is a blueprint or template for creating objects. It defines the attributes (fields) and behaviors (methods) that the objects of the class will have.

Example:
java
Copy code

```
// Defining a simple class
public class Car {
    // Fields (attributes)
    String make;
    String model;
    int year;

    // Constructor
    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    // Methods (behaviors)
    public void start() {
        System.out.println("The car is starting.");
    }

    public void drive() {
        System.out.println("The car is in motion.");
    }

    public void stop() {
        System.out.println("The car has stopped.");
    }
}
```

2. Objects:
An object is an instance of a class. It represents a real-world entity and encapsulates data and behaviors.

Example:
java
Copy code

```
public class CarDemo {
    public static void main(String[] args) {
        // Creating objects of the Car class
        Car myCar = new Car("Toyota", "Camry", 2022);
        Car anotherCar = new Car("Honda", "Civic", 2021);
```

```
        // Accessing fields
        System.out.println("My car: " + myCar.make + " " + myCar.model);
        System.out.println("Another car: " + anotherCar.make + " " + anotherCar.model);

        // Calling methods
        myCar.start();
        myCar.drive();
        myCar.stop();

        anotherCar.start();
        anotherCar.drive();
        anotherCar.stop();
    }
}
```

3. Methods:

Methods define the behavior of a class. They represent actions that objects of the class can perform.

Example:

java

Copy code

```
public class Calculator {
    // A simple method that adds two numbers
    public int add(int num1, int num2) {
        return num1 + num2;
    }

    // Another method that subtracts two numbers
    public int subtract(int num1, int num2) {
        return num1 - num2;
    }

    // Main method
    public static void main(String[] args) {
        // Creating an object of the Calculator class
        Calculator myCalculator = new Calculator();

        // Calling methods
        int result1 = myCalculator.add(5, 3);
        int result2 = myCalculator.subtract(8, 4);

        // Displaying results
        System.out.println("Addition result: " + result1);
        System.out.println("Subtraction result: " + result2);
    }
}
```

These examples illustrate the basic concepts of classes, objects, and methods in Java. Understanding how to design and use classes, create objects, and define methods is essential for Java development and will be tested in the OCPJP certification exam.

Return Types & Method Parameters

27 November 2023 15:56

Certainly! Understanding return types and method parameters is essential for Java programming and is a key aspect of the Oracle Certified Professional Java Programmer (OCPJP) certification. Let's explore these concepts with code examples:

1. **Return Types:**

A method's return type indicates the type of value that the method will return. In Java, every method has a return type. If a method doesn't return any value, its return type is specified as `void`.

Example:

```java
public class Calculator {
    // Method with return type int
    public int add(int num1, int num2) {
        return num1 + num2;
    }

    // Method with return type double
    public double divide(int numerator, int denominator) {
        // Ensure denominator is not zero to avoid division by zero
        if (denominator != 0) {
            return (double) numerator / denominator;
        } else {
            System.out.println("Cannot divide by zero.");
            return Double.NaN; // Not a Number
        }
    }

    // Method with return type void
    public void displayMessage(String message) {
        System.out.println("Message: " + message);
    }

    // Main method
    public static void main(String[] args) {
        Calculator myCalculator = new Calculator();

        // Calling methods with return types
        int sum = myCalculator.add(3, 5);
        double result = myCalculator.divide(10, 2);

        // Displaying results
        System.out.println("Sum: " + sum);
        System.out.println("Result of division: " + result);

        // Calling a method with void return type
        myCalculator.displayMessage("Hello, World!");
    }
}
```

2. **Method Parameters:**

Method parameters are variables that are specified in the method declaration. They represent the values that the method expects to receive when it is called.

Example:

```java
public class Greeting {
    // Method with parameters
    public void greetPerson(String name, int age) {
        System.out.println("Hello, " + name + "! You are " + age + " years old.");
    }

    // Main method
    public static void main(String[] args) {
        Greeting greeting = new Greeting();

        // Calling the method with arguments
        greeting.greetPerson("Alice", 25);
        greeting.greetPerson("Bob", 30);
    }
}
```

In this example, the `greetPerson` method takes two parameters: `name` (a `String`) and `age` (an `int`). When calling the method, actual values ("Alice" and 25, "Bob" and 30) are passed as arguments.

Understanding how to specify return types and use method parameters is crucial for writing effective and reusable Java code, and it's a topic that may be covered in the OCPJP certification exam.

Packages

28 November 2023 10:07

Certainly! Understanding packages is essential for Java programming and is a key concept tested in the Oracle Certified Professional Java Programmer (OCPJP) certification. Let's explore the concepts of packages with code examples:

1. **Packages Overview:**

A package in Java is a way to organize related classes, interfaces, and other types into a single unit. It helps in avoiding naming conflicts and provides a namespace for the classes. Packages also facilitate better project organization and maintenance.

2. **Package Declaration:**

To declare a class as a part of a package, you use the `package` statement at the beginning of your Java file. The package statement must be the first non-comment statement in the file.

Example:

```java
// Declaration of a package named "com.example"
package com.example;

// Class within the "com.example" package
public class MyClass {
    // Class code goes here
}
```

3. **Package Structure:**

Packages are organized hierarchically, and the package name reflects the directory structure. For example, a class in the package `com.example` is stored in a directory structure like `com/example/MyClass.java`.

4. **Import Statements:**

To use classes from other packages, you need to import them using the `import` statement. This statement informs the compiler about the location of the classes you want to use.

Example:

```java
// Importing a specific class from a package
import com.example.MyClass;

public class AnotherClass {
    public static void main(String[] args) {
        // Using MyClass
        MyClass myObject = new MyClass();
        // ...
    }
}
```

```

### 5. **Default Package:**

If a class does not specify a package using the `package` statement, it is considered part of the default package. However, it is recommended to use packages to avoid naming conflicts.

### 6. **Access Modifiers and Packages:**

Access modifiers (`public`, `private`, `protected`, and package-private/default) control the visibility of classes and members within packages.

#### Example:

```java
package com.example;

public class PublicClass {
   // This class can be accessed from other packages
}

class PackagePrivateClass {
   // This class can only be accessed within the "com.example" package
}
```

7. **Using Classes from the Java Standard Library:**

Many Java classes are part of the Java Standard Library and are organized into packages. For example, the `java.util` package contains utility classes like `ArrayList`, and the `java.io` package contains classes for input/output operations.

Example:

```java
import java.util.ArrayList;

public class UseArrayList {
   public static void main(String[] args) {
      // Using ArrayList from java.util
      ArrayList<String> myList = new ArrayList<>();
      myList.add("Java");
      myList.add("Programming");
      // ...
   }
}
```

Understanding how to create, organize, and use packages is crucial for developing maintainable and modular Java applications. It's a topic that is likely to be covered in the OCPJP certification exam.
```

Constructors and the usage of this and this()

28 November 2023 12:02

Certainly! Understanding constructors and the usage of `this` and `this()` in Java is crucial for Java programming and is a key concept tested in the Oracle Certified Professional Java Programmer (OCPJP) certification. Let's explore these concepts with code examples:

1. **Constructors:**

A constructor in Java is a special method used to initialize objects of a class. It has the same name as the class and does not have a return type. Constructors are called when an object is created using the `new` keyword.

Example:

```java
public class Car {
    // Default constructor
    public Car() {
        System.out.println("A new car is created.");
    }

    // Parameterized constructor
    public Car(String make, String model) {
        System.out.println("A " + make + " " + model + " is created.");
    }

    public static void main(String[] args) {
        // Creating objects using constructors
        Car defaultCar = new Car(); // Calls the default constructor
        Car customCar = new Car("Toyota", "Camry"); // Calls the parameterized constructor
    }
}
```

2. **`this` Keyword:**

The `this` keyword in Java is a reference to the current instance of the class. It is often used to differentiate between instance variables and parameters with the same name.

Example:

```java
public class Person {
    private String name;
    private int age;

    // Constructor using this to distinguish between instance variables and parameters
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method using this to refer to the current instance
```

```
    public void displayDetails() {
        System.out.println("Name: " + this.name + ", Age: " + this.age);
    }

    public static void main(String[] args) {
        // Creating an object and calling a method
        Person person = new Person("Alice", 25);
        person.displayDetails();
    }
}
```

3. **`this()` Constructor Call:**

The `this()` statement is used to invoke another constructor of the same class. It is often used to avoid code duplication when a class has multiple constructors.

Example:

```java
public class Book {
    private String title;
    private String author;

    // Parameterized constructor
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    // Constructor with a default author
    public Book(String title) {
        this(title, "Unknown"); // Calls the parameterized constructor
    }

    public static void main(String[] args) {
        // Creating objects using constructors
        Book book1 = new Book("Java Programming", "John Doe");
        Book book2 = new Book("Python Programming");

        // Displaying book details
        System.out.println("Book 1: " + book1.title + " by " + book2.author);
        System.out.println("Book 2: " + book2.title + " by " + book2.author);
    }
}
```

In this example, the second constructor uses `this(title, "Unknown")` to call the parameterized constructor, providing a default value for the author.

Understanding the use of constructors, `this`, and `this()` is important for creating well-structured and flexible Java classes. These concepts are likely to be covered in the OCPJP certification exam.

Static & Final

28 November 2023 14:19

Certainly! Understanding the concepts of `static` and `final` in Java is crucial for Java programming and is tested in the Oracle Certified Professional Java Programmer (OCPJP) certification. Let's explore these concepts with code examples:

1. **Static Keyword:**

a. **Static Variables (Class Variables):**

A static variable belongs to the class rather than instances of the class. It is shared among all instances of the class.

```java
public class MyClass {
    // Static variable
    static int staticVar = 10;

    public static void main(String[] args) {
        // Accessing a static variable
        System.out.println("Static variable: " + MyClass.staticVar);

        // Modifying a static variable
        MyClass.staticVar = 20;
        System.out.println("Modified static variable: " + MyClass.staticVar);
    }
}
```

b. **Static Methods:**

A static method belongs to the class rather than an instance of the class. It can be called using the class name.

```java
public class MathUtils {
    // Static method
    public static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        // Calling a static method
        int result = MathUtils.add(5, 3);
        System.out.println("Result of addition: " + result);
    }
}
```

2. **Final Keyword:**

a. **Final Variables:**

A final variable cannot be changed once it is assigned a value. It acts as a constant.

```java
public class Circle {
   // Final variable
   final double PI = 3.14159;
   int radius;

   // Constructor
   public Circle(int radius) {
      this.radius = radius;
   }

   // Method using a final variable
   public double calculateArea() {
      return PI * radius * radius;
   }

   public static void main(String[] args) {
      // Creating an object and using a final variable
      Circle myCircle = new Circle(5);
      System.out.println("Area of the circle: " + myCircle.calculateArea());
   }
}
```

b. **Final Methods:**

A final method cannot be overridden by subclasses.

```java
public class Animal {
   // Final method
   public final void makeSound() {
      System.out.println("Animal makes a sound.");
   }
}

class Dog extends Animal {
   // Attempting to override a final method will result in a compilation error
   // @Override
   // public void makeSound() {
   //     System.out.println("Dog barks.");
   // }
}
```

c. **Final Classes:**

A final class cannot be subclassed.

```java
final class FinalClass {
   // Class code
}

// Attempting to extend a final class will result in a compilation error
```

```
// class Subclass extends FinalClass {}
```

3. **Static and Final Together:**

A variable that is both `static` and `final` is essentially a constant shared among all instances of the class.

```java
public class Constants {
   // Static and final variable (constant)
   public static final double PI = 3.14159;

   public static void main(String[] args) {
      // Accessing a static and final variable
      System.out.println("Value of PI: " + Constants.PI);
   }
}
```

Understanding when and how to use `static` and `final` is crucial for creating efficient, maintainable, and secure Java code. These concepts are likely to be covered in the OCPJP certification exam.

Access Modifiers

28 November 2023 15:07

Access modifiers in Java are keywords that define the visibility or accessibility of classes, methods, and fields within a Java program. There are four access modifiers in Java: `public`, `protected`, `default` (package-private), and `private`. These modifiers control where a class, method, or field can be accessed from. Understanding access modifiers is essential for Java programming and is a key concept tested in the Oracle Certified Professional Java Programmer (OCPJP) certification. Let's explore these concepts with code examples:

1. **Public Access Modifier:**

A class, method, or field declared as `public` is accessible from any other class.

Example:

```java
// Public class
public class PublicClass {
    // Public method
    public void publicMethod() {
        System.out.println("This method is public.");
    }

    // Public field
    public int publicField = 10;
}
```

2. **Protected Access Modifier:**

A class, method, or field declared as `protected` is accessible within the same package and by subclasses.

Example:

```java
// Class with protected members
public class BaseClass {
    // Protected method
    protected void protectedMethod() {
        System.out.println("This method is protected.");
    }

    // Protected field
    protected int protectedField = 20;
}

// Subclass in the same package
class SubClass extends BaseClass {
    void accessProtectedMembers() {
        protectedMethod(); // Accessing protected method
        System.out.println("Value of protectedField: " + protectedField); // Accessing protected field
    }
```

```
}
```

3. **Default (Package-Private) Access Modifier:**

If no access modifier is specified (default), it is also known as package-private. A class, method, or field with default access is accessible only within the same package.

Example:

```java
// Class with default access
class DefaultAccessClass {
    // Default access method
    void defaultAccessMethod() {
        System.out.println("This method has default access.");
    }

    // Default access field
    int defaultAccessField = 30;
}
```

4. **Private Access Modifier:**

A class, method, or field declared as `private` is accessible only within the same class.

Example:

```java
// Class with private members
public class PrivateAccessClass {
    // Private method
    private void privateMethod() {
        System.out.println("This method is private.");
    }

    // Private field
    private int privateField = 40;

    // Public method to access private members
    public void accessPrivateMembers() {
        privateMethod(); // Accessing private method
        System.out.println("Value of privateField: " + privateField); // Accessing private field
    }
}
```

5. **Access Modifiers and Classes:**

When applied to classes, access modifiers control the visibility of the class itself.

Example:

```java
// Public class
public class PublicClass {
```

```
    // Class code
}

// Default access class (package-private)
class DefaultAccessClass {
    // Class code
}

// Class with a private constructor (preventing instantiation)
public class Singleton {
    private Singleton() {
        // Private constructor
    }

    public static Singleton getInstance() {
        return new Singleton();
    }
}
```

Understanding access modifiers is crucial for designing classes and maintaining encapsulation in Java programs. These concepts are likely to be covered in the OCPJP certification exam.

Singleton Class

02 December 2023 16:07

In Java, a singleton class is a class that is designed to have only one instance, and it provides a global point of access to that instance. Here's a simple explanation and example:

1. **Explanation:**
   - A singleton class typically has a private constructor to prevent external instantiation.
   - It contains a static method that returns the instance of the class.
   - The first time the static method is called, it creates an instance of the class. Subsequent calls return the same instance.

2. **Example:**
   ```java
   public class Singleton {
       // Private static instance variable
       private static Singleton instance;

       // Private constructor to prevent external instantiation
       private Singleton() {
           // Initialization code, if needed
       }

       // Public method to get the singleton instance
       public static Singleton getInstance() {
           // Create instance if not already present
           if (instance == null) {
               instance = new Singleton();
           }
           return instance;
       }

       // Other methods or properties of the singleton class can be added here

       // Example method
       public void showMessage() {
           System.out.println("Hello, I am a singleton instance!");
       }
   }
   ```

3. **Usage:**
   ```java
   public class Main {
       public static void main(String[] args) {
           // Get the singleton instance
           Singleton singletonInstance1 = Singleton.getInstance();
           Singleton singletonInstance2 = Singleton.getInstance();

           // Both instances will be the same
           System.out.println(singletonInstance1 == singletonInstance2);  // Output: true

           // Use the singleton instance
           singletonInstance1.showMessage();  // Output: Hello, I am a singleton instance!
   ```

```
    }
}
```

In this example, `Singleton` is a basic singleton class with a static method `getInstance()` that returns the singleton instance. The `main` method demonstrates that multiple calls to `getInstance()` return the same instance, confirming the singleton behavior.

Inheritance

02 December 2023 16:30

The Oracle Certified Professional Java Programmer (OCPJP) exam covers a variety of topics related to Java programming, including inheritance. Let's cover the essential concepts of inheritance in Java with code examples:

1. **Basic Inheritance:**

Inheritance allows a class to inherit properties and behaviors from another class.

```java
// Parent class
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

// Child class inheriting from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Inherited method
        myDog.bark(); // Method from the child class
    }
}
```

2. **Method Overriding:**

Child classes can provide a specific implementation for a method defined in the parent class.

```java
class Animal {
    void makeSound() {
        System.out.println("Generic Animal Sound");
    }
}

class Dog extends Animal {
    // Method overriding
    void makeSound() {
        System.out.println("Bark");
    }
}
```

```
// Usage
public class Main {
   public static void main(String[] args) {
      Dog myDog = new Dog();
      myDog.makeSound();  // Calls overridden method in Dog
   }
}
```

3. **Super Keyword:**

The `super` keyword is used to refer to the immediate parent class instance.

```java
class Animal {
   void eat() {
      System.out.println("Animal is eating");
   }
}

class Dog extends Animal {
   void eat() {
      super.eat();  // Calls the eat() method of the parent class
      System.out.println("Dog is eating");
   }
}

// Usage
public class Main {
   public static void main(String[] args) {
      Dog myDog = new Dog();
      myDog.eat();  // Calls overridden eat() method in Dog
   }
}
```

4. **Polymorphism:**

Polymorphism allows objects of different classes to be treated as objects of a common parent class.

```java
class Animal {
   void makeSound() {
      System.out.println("Generic Animal Sound");
   }
}

class Dog extends Animal {
   void makeSound() {
      System.out.println("Bark");
   }
}

// Usage
public class Main {
   public static void main(String[] args) {
      Animal myAnimal = new Dog();  // Polymorphic reference
```

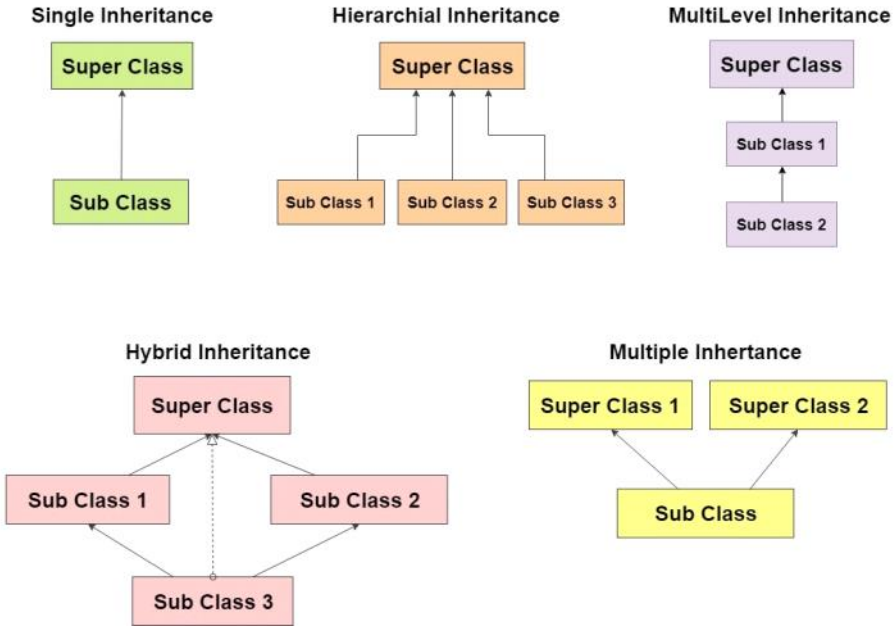
```
        myAnimal.makeSound();  // Calls overridden method in Dog
    }
}
```

5. **Abstract Classes and Methods:**

An abstract class cannot be instantiated, and it may contain abstract methods that must be implemented by its subclasses.

```java
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a Circle");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Circle myCircle = new Circle();
        myCircle.draw();  // Calls overridden draw() method in Circle
    }
}
```

These concepts are fundamental to understanding inheritance in Java and are likely to be covered in the OCPJP exam. Make sure to practice and understand how these concepts work in different scenarios.



Abstraction v/s Interface

02 December 2023 16:55

Certainly! Let's explore the concepts of Abstraction and Interface in Java with code examples.

Abstract classes are used to provide a base class for concrete subclasses to inherit from, while interfaces are used to define a set of methods that a class must implement. Abstract classes can have implemented and abstract methods, while interfaces can only have abstract methods.

In both cases, you cannot directly instantiate an abstract class or an interface in Java. Instead, you instantiate concrete subclasses or classes that implement the interface.

No, you cannot instantiate an interface. Generally, it contains abstract methods (except default and static methods introduced in Java8), which are incomplete.

Abstraction:

Abstraction is a fundamental concept in Java that involves representing essential features without providing the implementation details. In Java, abstraction is achieved through abstract classes and abstract methods. Abstract classes cannot be instantiated, and they may contain a mix of abstract (without a method body) and concrete methods.

```java
// Abstract class with abstract and concrete methods
abstract class Shape {
   abstract void draw();  // Abstract method without implementation

   void resize() {
      System.out.println("Resizing the shape");
   }
}

// Concrete class extending the abstract class
class Circle extends Shape {
   // Implementation of the abstract method
   void draw() {
      System.out.println("Drawing a Circle");
   }
}

// Usage
public class Main {
   public static void main(String[] args) {
      Circle myCircle = new Circle();
      myCircle.draw();    // Calls the overridden draw() method in Circle
      myCircle.resize();  // Calls the inherited resize() method from Shape
   }
}
```

Interface:

An interface in Java defines a contract for classes that implement it. It contains abstract methods (methods without a body) and constant declarations. A class can implement multiple interfaces,

allowing for multiple inheritance of behavior.

```java
// Interface with abstract methods
interface Shape {
   void draw();  // Abstract method without implementation

   void resize();  // Another abstract method
}

// Concrete class implementing the interface
class Circle implements Shape {
   // Implementation of the draw() method
   public void draw() {
      System.out.println("Drawing a Circle");
   }

   // Implementation of the resize() method
   public void resize() {
      System.out.println("Resizing the Circle");
   }
}

// Usage
public class Main {
   public static void main(String[] args) {
      Circle myCircle = new Circle();
      myCircle.draw();    // Calls the implemented draw() method in Circle
      myCircle.resize();  // Calls the implemented resize() method in Circle
   }
}
```

In this example, the `Shape` interface declares abstract methods `draw` and `resize`, and the `Circle` class implements these methods. The interface provides a contract for any class that wants to be considered a `Shape`. Implementing interfaces is a crucial concept for achieving abstraction and is frequently tested in the Oracle Certified Professional Java Programmer (OCPJP) exam.

Polymorphism

02 December 2023 17:03

Certainly! Polymorphism is a key concept in Java that allows objects to be treated as instances of their parent class, even if they are actually instances of a subclass. There are two types of polymorphism in Java: compile-time polymorphism (method overloading) and runtime polymorphism (method overriding). Here are examples for both:

Compile-Time Polymorphism (Method Overloading):

Compile-time polymorphism is achieved through method overloading, where multiple methods with the same name exist in the same class, but with different parameters.

```java
public class Calculator {
    // Method Overloading
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }

    String add(String a, String b) {
        return a + b;
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        System.out.println(calculator.add(1, 2));          // Output: 3
        System.out.println(calculator.add(1.5, 2.5));      // Output: 4.0
        System.out.println(calculator.add("Hello", "World"));  // Output: HelloWorld
    }
}
```

Runtime Polymorphism (Method Overriding):

Runtime polymorphism is achieved through method overriding, where a subclass provides a specific implementation for a method that is already defined in its superclass.

```java
// Parent class
class Animal {
    void makeSound() {
        System.out.println("Generic Animal Sound");
    }
}

// Child class
```

```
class Dog extends Animal {
    // Method overriding
    void makeSound() {
        System.out.println("Bark");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();  // Polymorphic reference
        myDog.makeSound();       // Calls overridden method in Dog
    }
}
```

In this example, `Animal` is the superclass with a generic `makeSound` method, and `Dog` is the subclass that overrides this method to provide a specific sound. The polymorphic reference (`Animal myDog`) allows us to call the overridden method in the `Dog` class.

Understanding polymorphism is crucial for the Oracle Certified Professional Java Programmer (OCPJP) exam, as it demonstrates your ability to design and implement Java classes following object-oriented principles.

Encapsulation

02 December 2023 17:16

Encapsulation is a fundamental concept in Java that involves bundling data (attributes) and methods that operate on the data into a single unit known as a class. The encapsulation principle helps in hiding the internal details of an object and protecting its state from outside interference. Here's an explanation with code examples:

Example 1: Basic Encapsulation

```java
public class Student {
    // Private attributes (encapsulated)
    private String name;
    private int age;

    // Public methods to access and modify the attributes (getters and setters)
    public String getName() {
        return name;
    }

    public void setName(String name) {
        // Additional validation or logic can be added here
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        // Additional validation or logic can be added here
        this.age = age;
    }
}
```

In this example, the `Student` class encapsulates the `name` and `age` attributes by making them private. Public getter and setter methods are provided to access and modify these attributes, allowing controlled access to the internal state of the object.

Example 2: Encapsulation with Constructor

```java
public class Person {
    // Private attributes (encapsulated)
    private String name;
    private int age;

    // Constructor for initializing attributes during object creation
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    // Public methods to access and modify the attributes (getters and setters)
    public String getName() {
        return name;
    }

    public void setName(String name) {
        // Additional validation or logic can be added here
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        // Additional validation or logic can be added here
        this.age = age;
    }
}
```

In this example, the `Person` class encapsulates the `name` and `age` attributes, and a constructor is used to initialize these attributes during object creation. This ensures that the object is in a valid state from the beginning.

Encapsulation is one of the four fundamental OOP principles and is emphasized in the Oracle Certified Professional Java Programmer (OCPJP) exam. It helps create well-organized and maintainable code by controlling access to the internal details of objects.

Multiple Inheritance

28 January 2024 15:16

In Java, multiple inheritance refers to the ability of a class to inherit behavior and characteristics from more than one parent class. However, Java does not support multiple inheritance of classes, meaning a class can only extend one superclass. This is because of the "diamond problem," where ambiguity arises if two superclasses have a method with the same name and signature.

However, Java does support multiple inheritance of interfaces. A class can implement multiple interfaces, allowing it to inherit abstract methods and constants from each interface. Here's an explanation of this concept along with code examples:

Interface Multiple Inheritance:

In Java, a class can implement multiple interfaces, allowing it to inherit behaviors and contracts from each interface. This effectively provides a form of multiple inheritance.

```java
// Interface defining behavior related to flying
interface Flyable {
    void fly();
}

// Interface defining behavior related to swimming
interface Swimmable {
    void swim();
}

// Concrete class implementing both Flyable and Swimmable interfaces
class Bird implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("Bird is flying.");
    }

    @Override
    public void swim() {
        System.out.println("Bird is swimming.");
    }
}

// Main class to demonstrate multiple interface inheritance
public class MultipleInheritanceExample {
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.fly();
        bird.swim();
    }
}
```

In the example above, the `Bird` class implements both the `Flyable` and `Swimmable` interfaces, which means it must provide concrete implementations for the `fly()` and `swim()` methods. This allows the `Bird` class to exhibit behaviors of both flying and swimming.

Why Interfaces for Multiple Inheritance?

Using interfaces for multiple inheritance in Java addresses the diamond problem because interfaces cannot contain concrete implementations of methods. They only provide method declarations, allowing classes implementing these interfaces to provide their own implementations.

Diamond Problem and Java:

Java avoids the diamond problem by not allowing multiple inheritance of classes. However, it allows multiple inheritance of interfaces, as interfaces only declare methods and do not provide concrete implementations.

Summary:

- Java supports multiple inheritance of interfaces, allowing a class to inherit behavior from multiple sources.
- A class can implement multiple interfaces, providing concrete implementations for methods declared in those interfaces.
- Interfaces provide a way to achieve the benefits of multiple inheritance without the ambiguities introduced by multiple inheritance of classes.

This approach ensures that Java maintains its simplicity and clarity in handling inheritance and method resolution.

Casting Numerical Values

28 January 2024 15:37

In Java, casting numerical values involves converting a value of one numeric data type to another. This is done when you want to assign a value of one type to a variable of another type or when you want to perform arithmetic operations involving different data types. Understanding casting is crucial for the Oracle Certified Professional Java Programmer exam. Here's an explanation along with code examples:

Widening (Implicit) Conversion:

Widening conversion occurs when you convert a value from a smaller data type to a larger data type. Java performs this conversion automatically as it doesn't result in loss of information.

```java
// Widening conversion (implicit)
int intValue = 10;
double doubleValue = intValue; // int to double (implicit conversion)
System.out.println(doubleValue); // Output: 10.0
```

In the example above, the `int` value `10` is automatically converted to a `double` value `10.0` when assigned to the `doubleValue`. This is because `double` has a larger range and can accommodate the `int` value without loss of precision.

Narrowing (Explicit) Conversion:

Narrowing conversion occurs when you convert a value from a larger data type to a smaller data type. This conversion may result in loss of information, so it requires explicit casting.

```java
// Narrowing conversion (explicit)
double doubleValue = 10.5;
int intValue = (int) doubleValue; // double to int (explicit conversion)
System.out.println(intValue); // Output: 10
```

In the example above, the `double` value `10.5` is explicitly cast to an `int` value. The fractional part is truncated, resulting in `intValue` being `10`.

Be cautious with narrowing conversions:

```java
int intValue = 1000000000;
byte byteValue = (byte) intValue; // int to byte (explicit conversion)
System.out.println(byteValue); // Output: -46
```

In the example above, the `int` value `1000000000` is narrowed to a `byte` value. Since `byte` can only hold values from -128 to 127, the result is unexpected (`-46`) due to overflow.

Beware of loss of precision:

```java
```

```
int intValue = 123456789;
float floatValue = intValue; // int to float (implicit conversion)
System.out.println(floatValue); // Output: 1.23456792E8
```

In the example above, the `int` value `123456789` is implicitly converted to a `float` value. Although there's no loss of precision, the floating-point representation might not be exact due to limited precision of `float`.

Summary:

- **Widening conversion:** Implicit conversion from a smaller data type to a larger data type. No explicit casting needed.
- **Narrowing conversion:** Explicit conversion from a larger data type to a smaller data type. May result in loss of information and requires explicit casting.
- Be cautious with narrowing conversions to avoid overflow or loss of precision.

Understanding these concepts of casting numerical values is essential for handling data types and performing arithmetic operations correctly in Java, which is important for the Oracle Certified Professional Java Programmer exam.

Wrapper Classes

28 January 2024 15:52

Wrapper classes in Java are used to convert primitive data types into objects and vice versa. This process is known as boxing (converting a primitive type to its corresponding wrapper class) and unboxing (converting a wrapper object back to its primitive type). Here's an explanation of the necessary concepts with code examples:

1. **Wrapper Classes**: In Java, there are eight primitive data types: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and `char`. Each primitive type has a corresponding wrapper class in Java.

2. **Autoboxing and Unboxing**: Autoboxing is the automatic conversion of primitive types into their corresponding wrapper objects, while unboxing is the reverse process. Java automatically performs these conversions when necessary.

Here are some examples demonstrating autoboxing and unboxing:

```java
public class WrapperExample {
    public static void main(String[] args) {
        // Autoboxing: Converting primitive types to wrapper objects
        Integer intValue = 10; // Autoboxing int to Integer
        Double doubleValue = 3.14; // Autoboxing double to Double
        Character charValue = 'a'; // Autoboxing char to Character

        // Unboxing: Converting wrapper objects to primitive types
        int a = intValue; // Unboxing Integer to int
        double b = doubleValue; // Unboxing Double to double
        char c = charValue; // Unboxing Character to char

        System.out.println("Autoboxing and Unboxing Examples:");
        System.out.println("Integer value: " + intValue);
        System.out.println("Double value: " + doubleValue);
        System.out.println("Character value: " + charValue);

        System.out.println("Unboxing values:");
        System.out.println("Unboxed int value: " + a);
        System.out.println("Unboxed double value: " + b);
        System.out.println("Unboxed char value: " + c);
    }
}
```

Output:
```
Autoboxing and Unboxing Examples:
Integer value: 10
Double value: 3.14
Character value: a
Unboxing values:
Unboxed int value: 10
Unboxed double value: 3.14
Unboxed char value: a
```

In the example above:
- Autoboxing occurs when assigning primitive values directly to their corresponding wrapper classes (`Integer`, `Double`, `Character`).
- Unboxing occurs when retrieving primitive values from wrapper objects.

Understanding wrapper classes and boxing concepts is essential for Java developers, especially for handling collections, generics, and other scenarios where primitive types need to be treated as objects.

Inner Class

29 January 2024 10:48

Java inner class or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Anonymous Classes

29 January 2024 10:49

In Java, a class can contain another class known as nested class. It's possible to create a nested class without giving any name.

A nested class that doesn't have any name is known as an anonymous class.

Needed for one time specific use.

Link - Java Anonymous Class (programiz.com)

String Builder

29 January 2024 13:55

Certainly! The `StringBuilder` class in Java is a mutable sequence of characters, providing an efficient way to concatenate strings. It's part of the `java.lang` package and is widely used for building and manipulating strings in situations where frequent changes are expected.

Here are the key concepts related to `StringBuilder` along with code examples:

1. **Instantiation and Initialization**:

- You can create an instance of `StringBuilder` using its constructor or by using the `StringBuilder` literal.

```java
// Using the constructor
StringBuilder stringBuilder1 = new StringBuilder();

// Using StringBuilder literal
StringBuilder stringBuilder2 = new StringBuilder("Hello");
```

2. **Append and Insert Operations**:

- The `append()` method is used to add characters or other data to the end of the `StringBuilder`.
- The `insert()` method is used to insert characters or other data at a specific position.

```java
StringBuilder stringBuilder = new StringBuilder("Hello");

// Append
stringBuilder.append(" World"); // Result: Hello World

// Insert
stringBuilder.insert(5, " Beautiful"); // Result: Hello Beautiful World
```

3. **Chaining Operations**:

- `StringBuilder` methods can be chained together for concise and readable code.

```java
StringBuilder result = new StringBuilder()
        .append("Hello")
        .append(" World")
        .insert(5, " Beautiful");

// Result: Hello Beautiful World
```

4. **String Conversion**:

- You can convert a `StringBuilder` object to a `String` using the `toString()` method.

```java
StringBuilder stringBuilder = new StringBuilder("Hello");
String resultString = stringBuilder.toString();
```

5. **Length and Capacity**:

- `length()`: Returns the current number of characters in the `StringBuilder`.
- `capacity()`: Returns the current capacity of the `StringBuilder`. The capacity is the maximum number of characters the `StringBuilder` can hold without resizing.

```java
StringBuilder stringBuilder = new StringBuilder("Hello");
int length = stringBuilder.length();     // 5
int capacity = stringBuilder.capacity(); // Default capacity or minimum capacity required to hold the string
```

6. **Set Length and Trim To Size**:

- `setLength(int newLength)`: Sets the length of the `StringBuilder`. If the new length is greater than the current length, null characters are added; if less, the `StringBuilder` is truncated.
- `trimToSize()`: Reduces the capacity of the `StringBuilder` to its current length.

```java
StringBuilder stringBuilder = new StringBuilder("Hello");
stringBuilder.setLength(3); // Result: Hel
stringBuilder.trimToSize(); // Reduces the capacity to fit the current length
```

7. **Reverse**:

- `reverse()`: Reverses the characters of the `StringBuilder` in-place.

```java
StringBuilder stringBuilder = new StringBuilder("Hello");
stringBuilder.reverse(); // Result: olleH
```

8. **Delete and Replace**:

- `delete(int start, int end)`: Deletes the characters from the `start` index to the `end` index.
- `replace(int start, int end, String str)`: Replaces the characters from the `start` index to the `end` index with the specified string.

```java
StringBuilder stringBuilder = new StringBuilder("Hello World");
stringBuilder.delete(5, 11);          // Result: Hello
stringBuilder.replace(0, 5, "Hi");    // Result: Hi
```

9. **Char Sequence Append**:

- `append(CharSequence cs)`: Appends a `CharSequence` (which includes `String`, `StringBuilder`, etc.) to the `StringBuilder`.

```java
```

```
StringBuilder stringBuilder = new StringBuilder("Hello");
stringBuilder.append(" World"); // Result: Hello World
```

10. **Performance Benefits**:

- `StringBuilder` is more efficient for frequent string manipulations compared to `String` concatenation using the `+` operator because it doesn't create a new object for each operation.

```java
// Less efficient
String result = "";
for (int i = 0; i < 10000; i++) {
   result += i;
}

// More efficient
StringBuilder stringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++) {
   stringBuilder.append(i);
}
String result = stringBuilder.toString();
```

Using `StringBuilder` is especially important in scenarios where a large number of string manipulations are involved to avoid unnecessary object creations and improve performance.

toString()

02 February 2024 11:48

The `toString()` method is a fundamental method in Java that is inherited from the `Object` class. It is used to obtain a string representation of an object. When you print an object directly using `System.out.println()` or concatenate it with a string, Java implicitly calls the `toString()` method to convert the object to a string representation.

Here are the key concepts related to the `toString()` method:

1. **Signature**: The `toString()` method is declared in the `Object` class as follows:
   ```java
   public String toString()
   ```

2. **Override**: It's common practice to override the `toString()` method in your custom classes to provide a meaningful string representation of the object's state. By default, the `toString()` method in the `Object` class returns a string containing the class name followed by `@` and the hexadecimal representation of the object's hash code.

3. **Customization**: When you override the `toString()` method, you can customize the string representation to include relevant information about the object's state, such as its fields or properties.

4. **Usage**: The `toString()` method is frequently used for logging, debugging, and printing object information in a human-readable format.

Here's an example demonstrating the usage of the `toString()` method:

```java
public class MyClass {
    private String name;
    private int age;

    public MyClass(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Override the toString() method to provide a custom string representation
    @Override
    public String toString() {
        return "MyClass{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    public static void main(String[] args) {
        MyClass obj = new MyClass("John", 30);
        System.out.println(obj); // Automatically calls obj.toString()
    }
}
```

In this example:
- We have a class `MyClass` with fields `name` and `age`.
- We override the `toString()` method to return a string containing the values of the `name` and `age` fields.
- When we print an instance of `MyClass`, Java implicitly calls the `toString()` method to obtain its string representation, which includes the values of the fields.

Output:
```
MyClass{name='John', age=30}
```

By overriding the `toString()` method, you can provide a more meaningful and informative string representation of your objects, which can be helpful for debugging and understanding the state of your program.

Equals & HashCode of Objects

02 February 2024 12:09

Certainly! Let's dive into the concepts of `equals()` and `hashCode()` methods in Java, along with the difference between `equals()` and `==`, with code examples.

1. `equals()` Method:

- **Purpose**: The `equals()` method is used to compare the content or value equality of two objects in Java.
- **Signature**: `public boolean equals(Object obj)`.
- **Default Implementation**: In the `Object` class, `equals()` checks whether two object references point to the same memory location (reference equality).
- **Contract**: The `equals()` method should satisfy the following properties:
  - Reflexive: `x.equals(x)` should return `true`.
  - Symmetric: `x.equals(y)` should return the same result as `y.equals(x)`.
  - Transitive: If `x.equals(y)` and `y.equals(z)` both return `true`, then `x.equals(z)` should also return `true`.
  - Consistent: Multiple invocations of `x.equals(y)` should consistently return the same result unless the objects are modified.
  - Null Handling: `x.equals(null)` should return `false`.
- **Override**: It's common practice to override `equals()` in user-defined classes to provide custom value-based equality comparisons.

2. `hashCode()` Method:

- **Purpose**: The `hashCode()` method returns a hash code value for an object, used primarily in hash-based collections (`HashMap`, `HashSet`, etc.).
- **Signature**: `public int hashCode()`.
- **Contract**: If two objects are equal according to the `equals()` method, they must have the same hash code. However, the reverse is not necessarily true (equal hash codes don't guarantee equal objects).
- **Default Implementation**: In the `Object` class, `hashCode()` returns the memory address of the object in hexadecimal form.
- **Override**: When `equals()` is overridden, it's necessary to override `hashCode()` as well to maintain the contract between these methods.

3. Difference Between `equals()` and `==`:

- **`equals()`**:
  - Compares the content or value equality of two objects.
  - It's a method defined in the `Object` class and can be overridden in user-defined classes.
  - Used to compare the actual contents of objects.
  - Example: `if (obj1.equals(obj2)) { /* Code block */ }`.

- **`==`**:
  - Compares object references for reference equality.
  - It's an operator used for reference comparison.
  - Used to check if two references point to the same memory location.
  - Example: `if (obj1 == obj2) { /* Code block */ }`.

Example Code:

```java
```

```
class MyClass {
    private int id;
    private String name;

    public MyClass(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        MyClass myClass = (MyClass) obj;
        return id == myClass.id && name.equals(myClass.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, name);
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass(1, "John");
        MyClass obj2 = new MyClass(1, "John");

        // Using equals() method
        System.out.println("Using equals() method: " + obj1.equals(obj2)); // true

        // Using == operator
        System.out.println("Using == operator: " + (obj1 == obj2)); // false
    }
}
```

In this example:
- We override `equals()` and `hashCode()` methods in the `MyClass` to provide custom value-based equality comparisons.
- We demonstrate the difference between `equals()` and `==` by comparing two `MyClass` objects with the same content. `equals()` returns `true`, indicating value equality, while `==` returns `false`, indicating reference inequality.

Super

16 February 2024 17:06

In object-oriented programming, `super` keyword is used to refer to the parent class of a subclass. It is commonly used in Java to access methods or fields of the superclass from within a subclass.

Here's how `super` is typically used in Java:

1. **Accessing Superclass Constructor**: When creating an instance of a subclass, you can call a constructor of the superclass using `super()`.

```java
public class SubClass extends SuperClass {
    public SubClass() {
        super(); // Calling superclass constructor
    }
}
```

2. **Accessing Superclass Methods**: You can use `super` to call methods of the superclass from within the subclass, especially if the subclass overrides the superclass method and you want to invoke the superclass implementation.

```java
public class SuperClass {
    public void display() {
        System.out.println("SuperClass display method");
    }
}

public class SubClass extends SuperClass {
    @Override
    public void display() {
        super.display(); // Calling superclass method
        System.out.println("SubClass display method");
    }
}
```

3. **Accessing Superclass Fields**: Similarly, `super` can be used to access fields of the superclass from within the subclass.

```java
public class SuperClass {
    protected int value = 10;
}

public class SubClass extends SuperClass {
    public void display() {
        System.out.println(super.value); // Accessing superclass field
    }
}
```

4. **Constructor Chaining**: `super()` can also be used to call a superclass constructor from within the constructor of a subclass. This is often used to perform initialization tasks defined in the superclass before initializing the subclass.

```java
public class SuperClass {
    public SuperClass(int value) {
        // superclass initialization
    }
}

public class SubClass extends SuperClass {
    public SubClass(int value) {
        super(value); // Calling superclass constructor
        // subclass initialization
    }
}
```

5. **Accessing Overridden Methods**: If a method in the subclass overrides a method in the superclass, `super` can be used to invoke the superclass implementation of the method.

```java
public class SuperClass {
    public void display() {
        System.out.println("SuperClass display method");
    }
}

public class SubClass extends SuperClass {
    @Override
    public void display() {
        super.display(); // Calling superclass method
        // Additional subclass logic
    }
}
```

In summary, `super` keyword is used in Java to access members of the superclass within the subclass, including constructors, methods, and fields. It facilitates code reuse and allows subclasses to extend and customize the behavior of their parent classes.

Ternary Operator

19 February 2024 09:54

Certainly! The ternary operator in Java is a shorthand way of writing a simple if-else statement. It's often used for assigning values to variables based on a condition. Here's how it works:

```java
// Syntax: condition ? value_if_true : value_if_false

int x = 10;
int y;

// Example 1: Assigning a value based on a condition
y = (x > 5) ? 1 : 0; // If x is greater than 5, y will be 1, otherwise 0

System.out.println(y); // Output: 1

// Example 2: Using ternary operator in a return statement
public int getMax(int a, int b) {
   return (a > b) ? a : b; // Returns the maximum of a and b
}

System.out.println(getMax(3, 7)); // Output: 7

// Example 3: Nesting ternary operators
int score = 85;
String result = (score >= 70) ? "Pass" : (score >= 50) ? "Retake" : "Fail";
System.out.println(result); // Output: Pass
```

In Example 1, the ternary operator `(x > 5) ? 1 : 0` evaluates to `1` because `x` is indeed greater than `5`. Therefore, `y` is assigned the value `1`.

In Example 2, the `getMax` method returns the maximum of two integers `a` and `b`. It uses the ternary operator to check if `a` is greater than `b`. If it is, `a` is returned, otherwise `b` is returned.

In Example 3, the ternary operator is nested to check different conditions. It first checks if the `score` is greater than or equal to `70`, if true, it returns `"Pass"`, otherwise, it checks if the `score` is greater than or equal to `50`, if true, it returns `"Retake"`, otherwise `"Fail"` is returned.

Handling Exceptions

19 February 2024 10:34

Handling exceptions in Java is crucial for writing robust and error-tolerant code. Here are the key concepts related to exception handling along with code examples:

1. **Try-Catch Block**:
   - Use `try` block to enclose the code that might throw an exception.
   - Use `catch` block to handle the exception.

```java
try {
    // Code that might throw an exception
    int result = 10 / 0; // This will throw ArithmeticException
} catch (ArithmeticException e) {
    // Handling the exception
    System.out.println("Cannot divide by zero");
}
```

2. **Multiple Catch Blocks**:
   - You can have multiple catch blocks to handle different types of exceptions.

```java
try {
    // Code that might throw exceptions
    int[] arr = new int[5];
    arr[5] = 10; // This will throw ArrayIndexOutOfBoundsException
} catch (ArithmeticException e) {
    // Handling ArithmeticException
    System.out.println("Arithmetic Exception occurred");
} catch (ArrayIndexOutOfBoundsException e) {
    // Handling ArrayIndexOutOfBoundsException
    System.out.println("Array Index Out Of Bounds Exception occurred");
}
```

3. **Finally Block**:
   - `finally` block is used to execute code regardless of whether an exception is thrown or not.

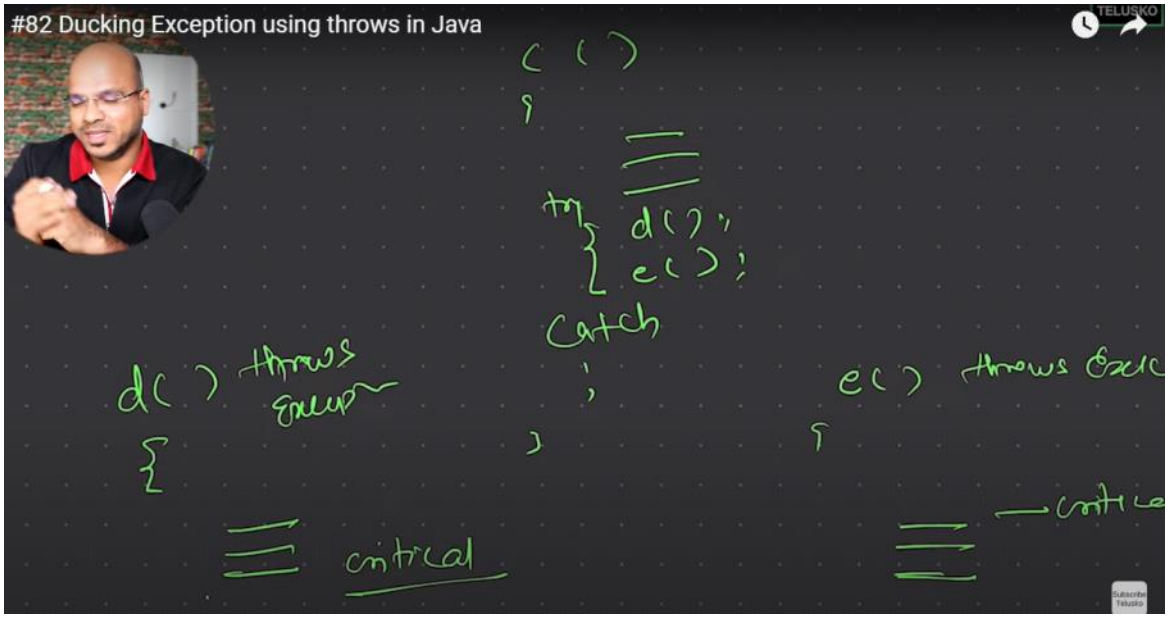
```java
try {
    // Code that might throw an exception
    int[] arr = new int[5];
    arr[5] = 10; // This will throw ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException e) {
    // Handling ArrayIndexOutOfBoundsException
    System.out.println("Array Index Out Of Bounds Exception occurred");
} finally {
    // This block will always execute
    System.out.println("Finally block executed");
}
```

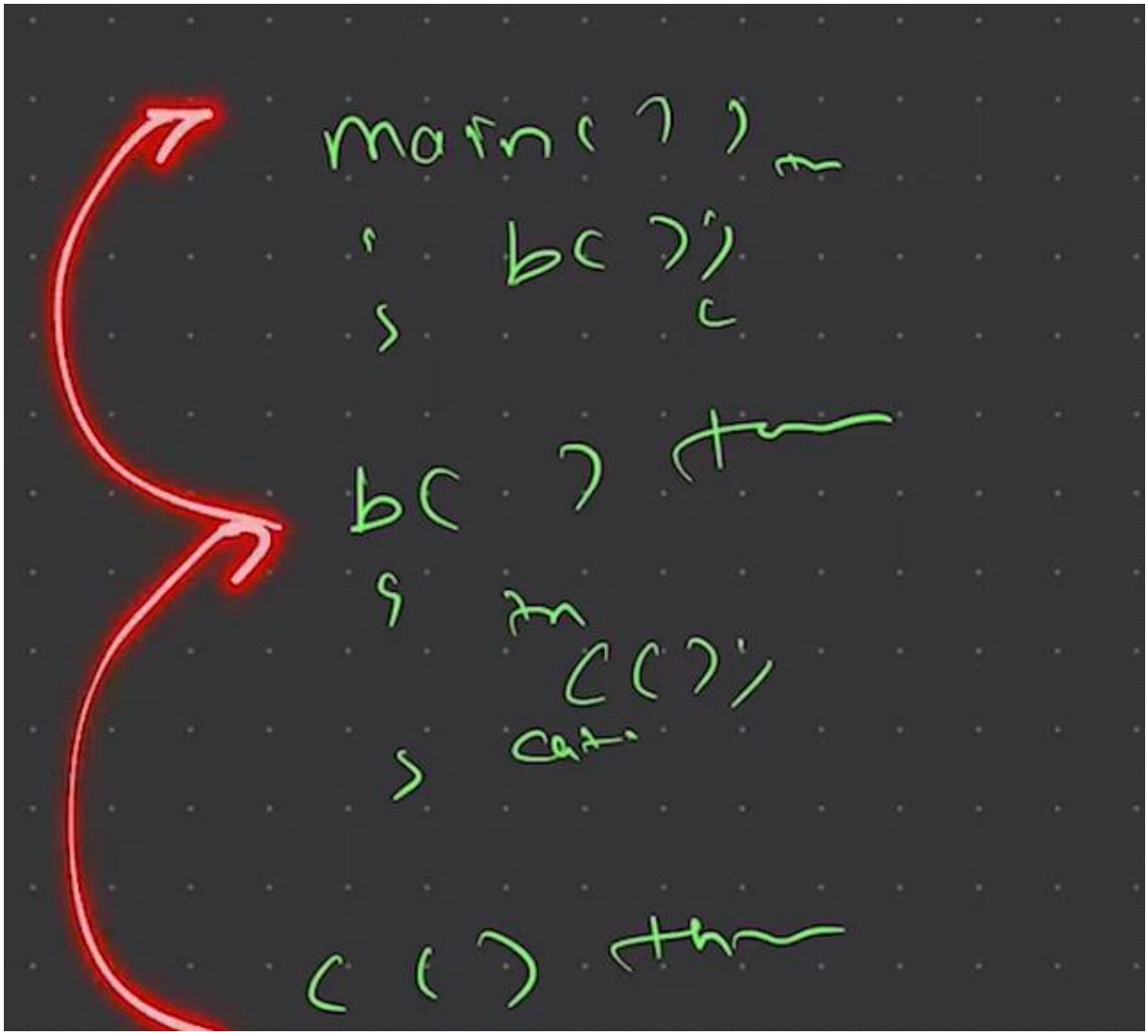
4. **Throwing Exceptions**:
   - You can explicitly throw exceptions using the `throw` keyword.

```java
public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            throw new CustomException("Custom Exception Occurred");
        } catch (CustomException e) {
            System.out.println(e.getMessage());
        }
    }
}

class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}
```

5. **Exception Propagation**:
   - If an exception is not caught at a particular level, it propagates up the call stack until it's caught or the program terminates.

These concepts are essential for handling exceptions effectively in Java and writing robust applications.





Passing by Value vs Pass by reference

19 February 2024 14:40

In Java, it's important to understand the difference between passing by value and passing by reference when working with methods and objects. Let's break down both concepts with code examples:

1. **Passing by Value**:
   - When you pass a primitive data type (like int, float, char, etc.) to a method, a copy of the value is passed.
   - Changes made to the parameter inside the method do not affect the original value outside the method.

```java
public class PassByValueExample {
    public static void main(String[] args) {
        int x = 10;
        System.out.println("Before method call: " + x);
        modifyValue(x);
        System.out.println("After method call: " + x); // Output: After method call: 10
    }

    public static void modifyValue(int num) {
        num = 20; // Changes made to num inside the method do not affect the original value of x
    }
}
```

2. **Passing by Reference (or Reference Copy)**:
   - When you pass an object (non-primitive data type) to a method, a copy of the reference to the object is passed, not the actual object itself.
   - This means changes made to the object's state inside the method will affect the original object outside the method.

```java
public class PassByReferenceExample {
    public static void main(String[] args) {
        StringBuilder str = new StringBuilder("Hello");
        System.out.println("Before method call: " + str);
        modifyString(str);
        System.out.println("After method call: " + str); // Output: After method call: Hello World
    }

    public static void modifyString(StringBuilder s) {
        s.append(" World"); // Changes made to s (StringBuilder object) inside the method affect the original StringBuilder object
    }
}
```

In the second example, even though we pass a copy of the reference `str` to the `modifyString` method, changes made to the `StringBuilder` object inside the method are reflected in the original object outside the method.

Understanding these concepts is important for correctly managing data and objects in Java programs, especially when working with methods and object-oriented programming.

Enums

19 February 2024 14:59

Enums in Java provide a way to define a set of named constants. They are used to represent fixed sets of constants, like days of the week, months, status codes, etc. Here's a breakdown of concepts related to enums along with code examples:

1. **Declaring Enums**:
   - Enums are declared using the `enum` keyword.

```java
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

2. **Using Enums**:
   - Enums can be used to define variables and method parameters.

```java
public class EnumExample {
    public static void main(String[] args) {
        Day today = Day.MONDAY;
        System.out.println("Today is " + today);
    }
}
```

3. **Enum Methods**:
   - Enums can have fields, constructors, and methods like a regular class.

```java
public enum Day {
    MONDAY("First day of the week"),
    TUESDAY("Second day of the week"),
    // Other days...

    private final String description;

    private Day(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }
}
```

4. **Switch Statements with Enums**:
   - Enums are often used with switch statements for better code readability.

```java
public class EnumSwitchExample {
```

```
    public static void main(String[] args) {
        Day today = Day.MONDAY;
        switch (today) {
            case MONDAY:
                System.out.println("Monday is the start of the week");
                break;
            case TUESDAY:
                System.out.println("Tuesday is the second day of the week");
                break;
            // Other cases...
        }
    }
}
```

5. **Enum Constants with Fields**:
   - Enums can have fields and methods associated with each constant.

```java
public enum Day {
    MONDAY(1),
    TUESDAY(2),
    // Other days...

    private final int value;

    private Day(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

These two lines define an enum `DayOfWeek` with constants representing days of the week, each associated with an integer value:

1. `MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);`

   - Defines enum constants for each day of the week, with Monday having a value of 1, Tuesday with 2, and so on, up to Sunday with a value of 7.

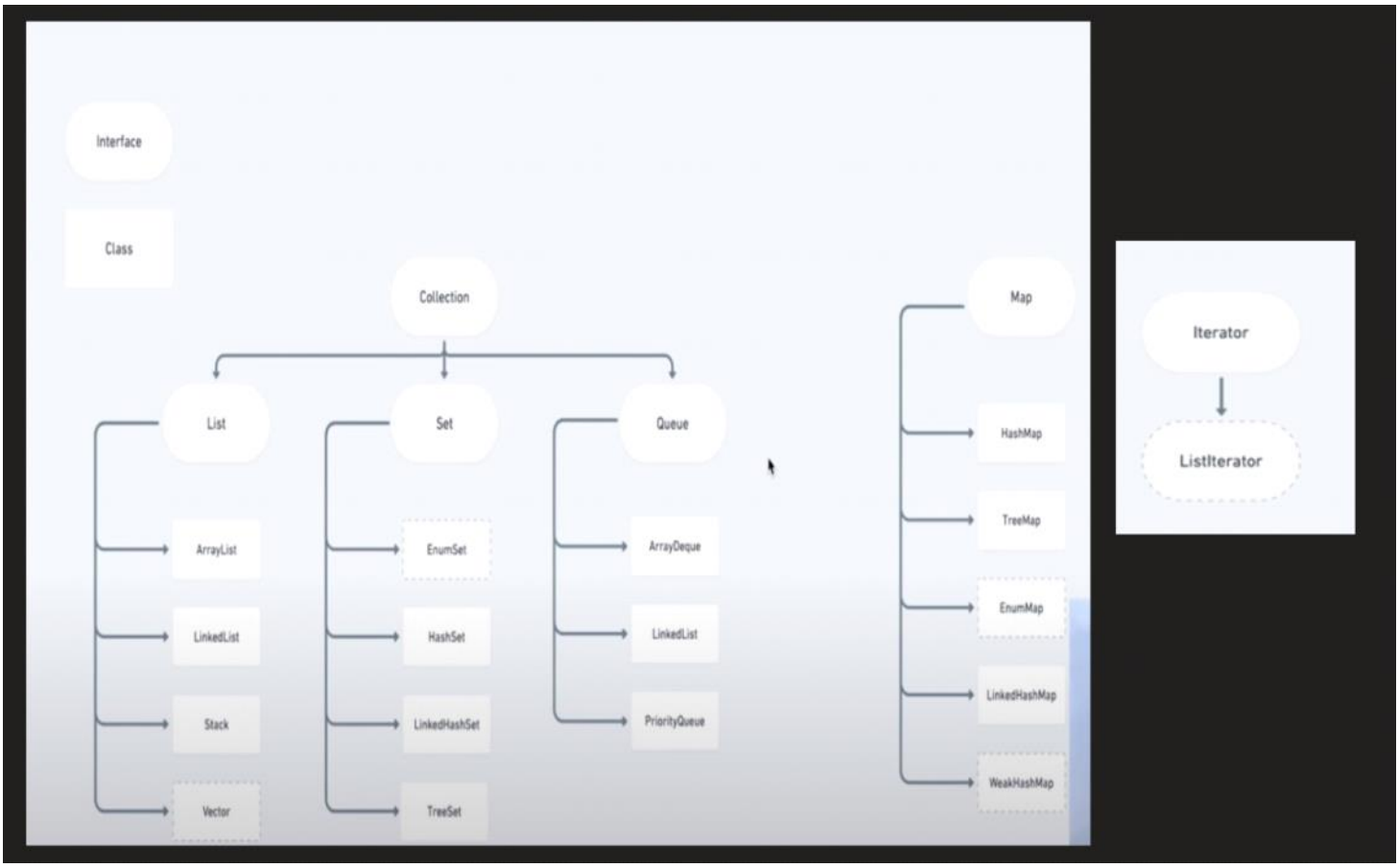
2. `private DayOfWeek(int dayValue) { this.dayValue = dayValue; }`

   - Defines a constructor for the enum `DayOfWeek` that takes an integer parameter `dayValue` and assigns it to the `dayValue` field for each enum constant. This allows setting the integer value associated with each day of the week when defining the enum constants.

Enums are a powerful feature in Java, providing type safety and readability to your code. They are often used to represent fixed sets of constants and improve code maintainability. Understanding how to declare, use, and work with enums is essential for Java developers, especially for OCPJP certification.

Collections

29 February 2024 11:54



Java Collections Cheat Sheet

Java Concept Of The Day

Basics

What is Java Collection Framework?

Java Collection Framework is a framework which provides some predefined classes and interfaces to store and manipulate the group of objects. Using Java collection framework, you can store the objects as a List or as a Set or as a Queue or as a Map and perform basic operations like adding, removing, updating, sorting, searching etc... with ease.

Why Java Collection Framework?

Earlier, arrays are used to store the group of objects. But, arrays are of fixed size. You can't change the size of an array once it is defined. It causes lots of difficulties while handling the group of objects. To overcome this drawback of arrays, Java Collection Framework is introduced from JDK 1.2.

Java Collections Hierarchy :

All the classes and interfaces related to Java collections are kept in java.util package. List, Set, Queue and Map are four top level interfaces of Java collection framework. All these interfaces (except Map) inherit from java.util.Collection interface which is the root interface in the Java collection framework.

List	Queue	Set	Map
Intro : • List is a sequential collection of objects. • Elements are positioned using zero-based index. • Elements can be inserted or removed or retrieved from any arbitrary position using an integer index. **Popular Implementations :** • ArrayList, Vector And LinkedList **Internal Structure :**	**Intro :** • Queue is a data structure where elements are added from one end called tail of the queue and elements are removed from another end called head of the queue. • Queue is typically FIFO (First-In-First-Out) type of data structure. **Popular Implementations :** • PriorityQueue, ArrayDeque and LinkedList (implements List also) **Internal Structure :**	**Intro :** • Set is a linear collection of objects with no duplicates. • Set interface does not have its own methods. All its methods are inherited from Collection interface. It just applies restriction on methods so that duplicate elements are always avoided. **Popular Implementations :** • HashSet, LinkedHashSet and TreeSet **Internal Structure :**	**Intro :** • Map stores the data in the form of key-value pairs where each key is associated with a value. • Map interface is part of Java collection framework but it doesn't inherit Collection interface. **Popular Implementations :** • HashMap, LinkedHashMap And TreeMap **Internal Structure :** • **HashMap :** It internally uses an

- ArrayList, Vector And LinkedList

Internal Structure :

- **ArrayList :** Internally uses re-sizable array which grows or shrinks as we add or delete elements.
- **Vector :** Same as ArrayList but it is synchronized.
- **LinkedList :** Elements are stored as Nodes where each node consists of three parts – Reference To Previous Element, Value Of The Element and Reference To Next Element.

Null Elements :

- **ArrayList :** Yes
- **Vector :** Yes
- **LinkedList :** Yes

Duplicate Elements :

- **ArrayList :** Yes
- **Vector :** Yes
- **LinkedList :** Yes

Order Of Elements :

- **ArrayList :** Insertion Order
- **Vector :** Insertion Order
- **LinkedList :** Insertion Order

Synchronization :

- **ArrayList :** Not synchronized
- **Vector :** Synchronized
- **LinkedList :** Not synchronized

Performance :

- **ArrayList :** Insertion -> O(1) (if insertion causes restructuring of internal array, it will be O(n)), Removal -> O(1) (if removal causes restructuring of internal array, it will be O(n)), Retrieval -> O(1)
- **Vector :** Similar to ArrayList but little slower because of synchronization.
- **LinkedList :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(n)

When to use?

- **ArrayList :** Use it when more search operations are needed then insertion and removal.
- **Vector :** Use it when you need synchronized list.
- **LinkedList :** Use it when insertion and removal are needed frequently.

- PriorityQueue, ArrayDeque and LinkedList (implements List also)

Internal Structure :

- **PriorityQueue :** It internally uses re-sizable array to store the elements and a Comparator to place the elements in some specific order.
- **ArrayDeque :** It internally uses re-sizable array to store the elements.

Null Elements :

- **PriorityQueue :** Not allowed
- **ArrayDeque :** Not allowed

Duplicate Elements :

- **PriorityQueue :** Yes
- **ArrayDeque :** Yes

Order Of Elements :

- **PriorityQueue :** Elements are placed according to supplied Comparator or in natural order if no Comparator is supplied.
- **ArrayDeque :** Supports both LIFO and FIFO

Synchronization :

- **PriorityQueue :** Not synchronized
- **ArrayDeque :** Not synchronized

Performance :

- **PriorityQueue :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(1)
- **ArrayDeque :** Insertion -> O(1) , Removal -> O(n), Retrieval -> O(1)

When to use?

- **PriorityQueue :** Use it when you want a queue of elements placed in some specific order.
- **ArrayDeque :** You can use it as a queue OR as a stack.

- HashSet, LinkedHashSet and TreeSet

Internal Structure :

- **HashSet :** Internally uses HashMap to store the elements.
- **LinkedHashSet :** Internally uses LinkedHashMap to store the elements.
- **TreeSet :** Internally uses TreeMap to store the elements.

Null Elements :

- **HashSet :** Maximum one null element
- **LinkedHashSet :** Maximum one null element.
- **TreeSet :** Doesn't allow even a single null element

Duplicate Elements :

- **HashSet :** Not allowed
- **LinkedHashSet :** Not allowed
- **TreeSet :** Not allowed

Order Of Elements :

- **HashSet :** No order
- **LinkedHashSet :** Insertion order
- **TreeSet :** Elements are placed according to supplied Comparator or in natural order if no Comparator is supplied.

Synchronization :

- **HashSet :** Not synchronized
- **LinkedHashSet :** Not synchronized
- **TreeSet :** Not synchronized

Performance :

- **HashSet :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **LinkedHashSet :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **TreeSet :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(log(n))

When to use?

- **HashSet :** Use it when you want only unique elements without any order.
- **LinkedHashSet :** Use it when you want only unique elements in insertion order.
- **TreeSet :** Use it when you want only unique elements in some specific order.

~~TreeMap~~

Internal Structure :

- **HashMap :** It internally uses an array of buckets where each bucket internally uses linked list to hold the elements.
- **LinkedHashMap :** Same as HashMap but it additionally uses a doubly linked list to maintain insertion order of elements.
- **TreeMap :** It internally uses Red-Black tree.

Null Elements :

- HashMap : Only one null key and can have multiple null values
- LinkedHashMap : Only one null key and can have multiple null values.
- TreeMap : Doesn't allow even a single null key but can have multiple null values.

Duplicate Elements :

- **HashMap :** Doesn't allow duplicate keys but can have duplicate values.
- **LinkedHashMap :** Doesn't allow duplicate keys but can have duplicate values.
- **TreeMap :** Doesn't allow duplicate keys but can have duplicate values.

Order Of Elements :

- **HashMap :** No Order
- **LinkedHashMap :** Insertion Order
- **TreeMap :** Elements are placed according to supplied Comparator or in natural order of keys if no Comparator is supplied.

Synchronization :

- **HashMap :** Not synchronized
- **LinkedHashMap :** Not Synchronized
- **TreeMap :** Not Synchronized

Performance :

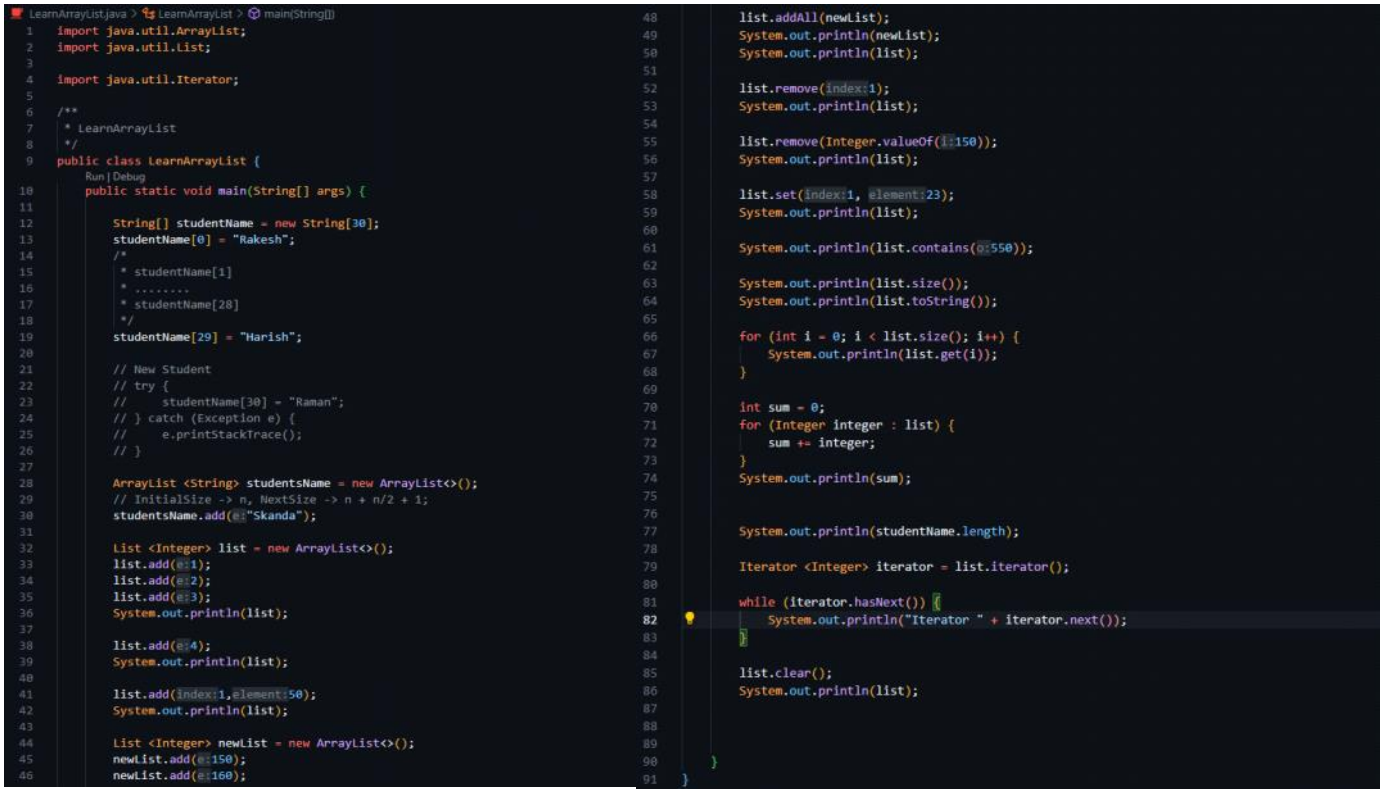
- **HashMap :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **LinkedHashMap :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **TreeMap :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(log(n))

When to use?

- **HashMap :** Use it if you want only key-value pairs without any order.
- **LinkedHashMap :** Use it if you want key-value pairs in insertion order.
- **TreeMap :** Use it when you want key-value pairs sorted in some specific order.

ArrayList & Iterator

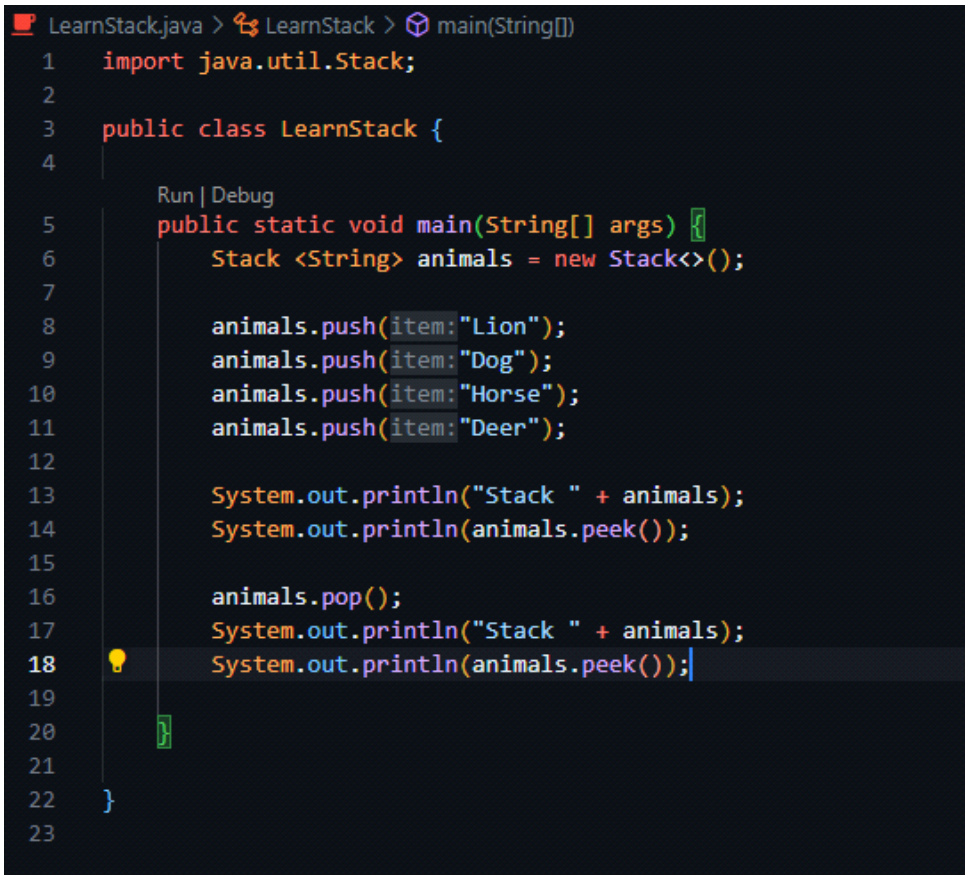
04 March 2024 11:30



```java
import java.util.ArrayList;
import java.util.List;

import java.util.Iterator;

/**
 * LearnArrayList
 */
public class LearnArrayList {
    Run | Debug
    public static void main(String[] args) {

        String[] studentName = new String[30];
        studentName[0] = "Rakesh";
        /*
         * studentName[1]
         * ........
         * studentName[28]
         */
        studentName[29] = "Harish";

        // New Student
        // try {
        //     studentName[30] = "Raman";
        // } catch (Exception e) {
        //     e.printStackTrace();
        // }

        ArrayList <String> studentsName = new ArrayList<>();
        // InitialSize -> n, NextSize -> n + n/2 + 1;
        studentsName.add(e:"Skanda");

        List <Integer> list = new ArrayList<>();
        list.add(e:1);
        list.add(e:2);
        list.add(e:3);
        System.out.println(list);

        list.add(e:4);
        System.out.println(list);

        list.add(index:1,element:50);
        System.out.println(list);

        List <Integer> newList = new ArrayList<>();
        newList.add(e:150);
        newList.add(e:160);
        list.addAll(newList);
        System.out.println(newList);
        System.out.println(list);

        list.remove(index:1);
        System.out.println(list);

        list.remove(Integer.valueOf(i:150));
        System.out.println(list);

        list.set(index:1, element:23);
        System.out.println(list);

        System.out.println(list.contains(o:550));

        System.out.println(list.size());
        System.out.println(list.toString());

        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }

        int sum = 0;
        for (Integer integer : list) {
            sum += integer;
        }
        System.out.println(sum);


        System.out.println(studentName.length);

        Iterator <Integer> iterator = list.iterator();

        while (iterator.hasNext()) {
            System.out.println("Iterator " + iterator.next());
        }

        list.clear();
        System.out.println(list);


    }
}
```

Stack

04 March 2024 11:32



Queue & LinkedList

04 March 2024 11:36

```java
import java.util.LinkedList;
import java.util.Queue;

public class LearnQueueLinkedList {
    Run | Debug
    public static void main(String[] args) {

        Queue <Integer> queue = new LinkedList<>();

        queue.offer(e:1);
        queue.offer(e:2);
        queue.offer(e:3);

        System.out.println(queue);

        System.out.println(queue.poll());

        System.out.println(queue);

        System.out.println(queue.peek());

    }
}
```

Note : All functions used in ArrayList can be used in LinkedList

PriorityQueue

04 March 2024 11:59

LearnPriorityQueue.java > LearnPriorityQueue > main(String[])

```java
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

public class LearnPriorityQueue {
    Run | Debug
    public static void main(String[] args) {

        // Observe first Element
        // Use PriorityQueue to implement minheap
        Queue<Integer> pQueue = new PriorityQueue<>();

        pQueue.offer(e:4);
        pQueue.offer(e:1);
        pQueue.offer(e:2);
        pQueue.offer(e:3); // Output : [1, 3, 2, 4] , reason : minheap

        System.out.println(pQueue);
        pQueue.poll();
        System.out.println(pQueue); // // Output : [2, 3, 4] , reason : heapify & minheap

        System.out.println(pQueue.peek());

        // Use PriorityQueue to implement maxheap
        Queue<Integer> pQueue1 = new PriorityQueue<>(Comparator.reverseOrder());

        pQueue1.offer(e:4);
        pQueue1.offer(e:1);
        pQueue1.offer(e:2);
        pQueue1.offer(e:3); // Output :  , reason : maxheap

        System.out.println(pQueue1);
        pQueue1.poll();
        System.out.println(pQueue1); // // Output :  , reason : heapify & maxheap

        System.out.println(pQueue1.peek());

    }
}
```

ArrayDeque

04 March 2024 12:36

```java
// ArrayDeck

import java.util.ArrayDeque;

public class LearnArrayDeque {
    Run | Debug
    public static void main(String[] args) {

        ArrayDeque<Integer> adQueue = new ArrayDeque<>();

        adQueue.offer(e:23);
        adQueue.offerFirst(e:12);
        System.out.println(adQueue);

        adQueue.offerLast(e:45);
        System.out.println(adQueue);

        adQueue.offerLast(e:26);
        System.out.println(adQueue);

        System.out.println("adQueue.peek() : " + adQueue.peek());
        System.out.println("adQueue.peekFirst() : " + adQueue.peekFirst());
        System.out.println("adQueue.peekLast() : " + adQueue.peekLast());

        System.out.println(adQueue);
        System.out.println("adQueue.poll() " + adQueue.poll());
        System.out.println(adQueue);
        System.out.println("adQueue.pollFirst() " + adQueue.pollFirst());
        System.out.println(adQueue);
        System.out.println("adQueue.pollLast() " + adQueue.pollLast());
        System.out.println("" + adQueue);

        // Output :
        // [12, 23]
        // [12, 23, 45]
        // [12, 23, 45, 26]
        // adQueue.peek() : 12
        // adQueue.peekFirst() : 12
        // adQueue.peekLast() : 26
        // [12, 23, 45, 26]
        // adQueue.poll() 12
        // [23, 45, 26]
        // adQueue.pollFirst() 23
        // [45, 26]
        // adQueue.pollLast() 26
        // [45]
```

HashSet

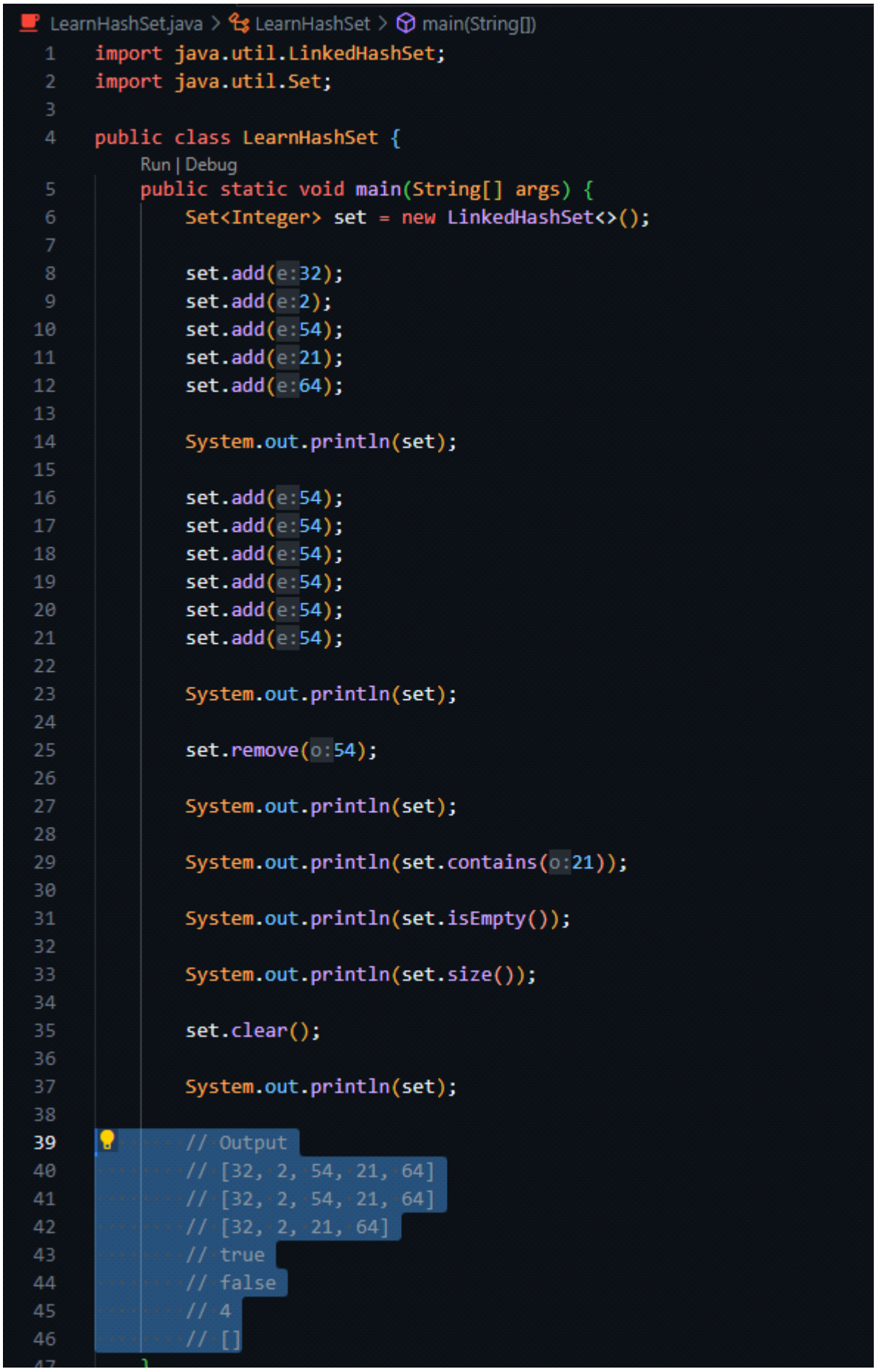
06 March 2024 14:58

LearnHashSet.java > LearnHashSet > main(String[])

```java
import java.util.HashSet;
import java.util.Set;

public class LearnHashSet {
    Run | Debug
    public static void main(String[] args) {
        Set<Integer> set = new HashSet<>();

        set.add(e:32);
        set.add(e:2);
        set.add(e:54);
        set.add(e:21);
        set.add(e:64);

        System.out.println(set);

        set.add(e:54);
        set.add(e:54);
        set.add(e:54);
        set.add(e:54);
        set.add(e:54);
        set.add(e:54);

        System.out.println(set);

        set.remove(o:54);

        System.out.println(set);

        System.out.println(set.contains(o:21));

        System.out.println(set.isEmpty());

        System.out.println(set.size());

        set.clear();

        System.out.println(set);

        // Output
        // [32, 64, 2, 21, 54]
        // [32, 64, 2, 21, 54]
        // [32, 64, 2, 21]
        // true
        // false
        // 4
        // []
    }
```

LinkedHashSet

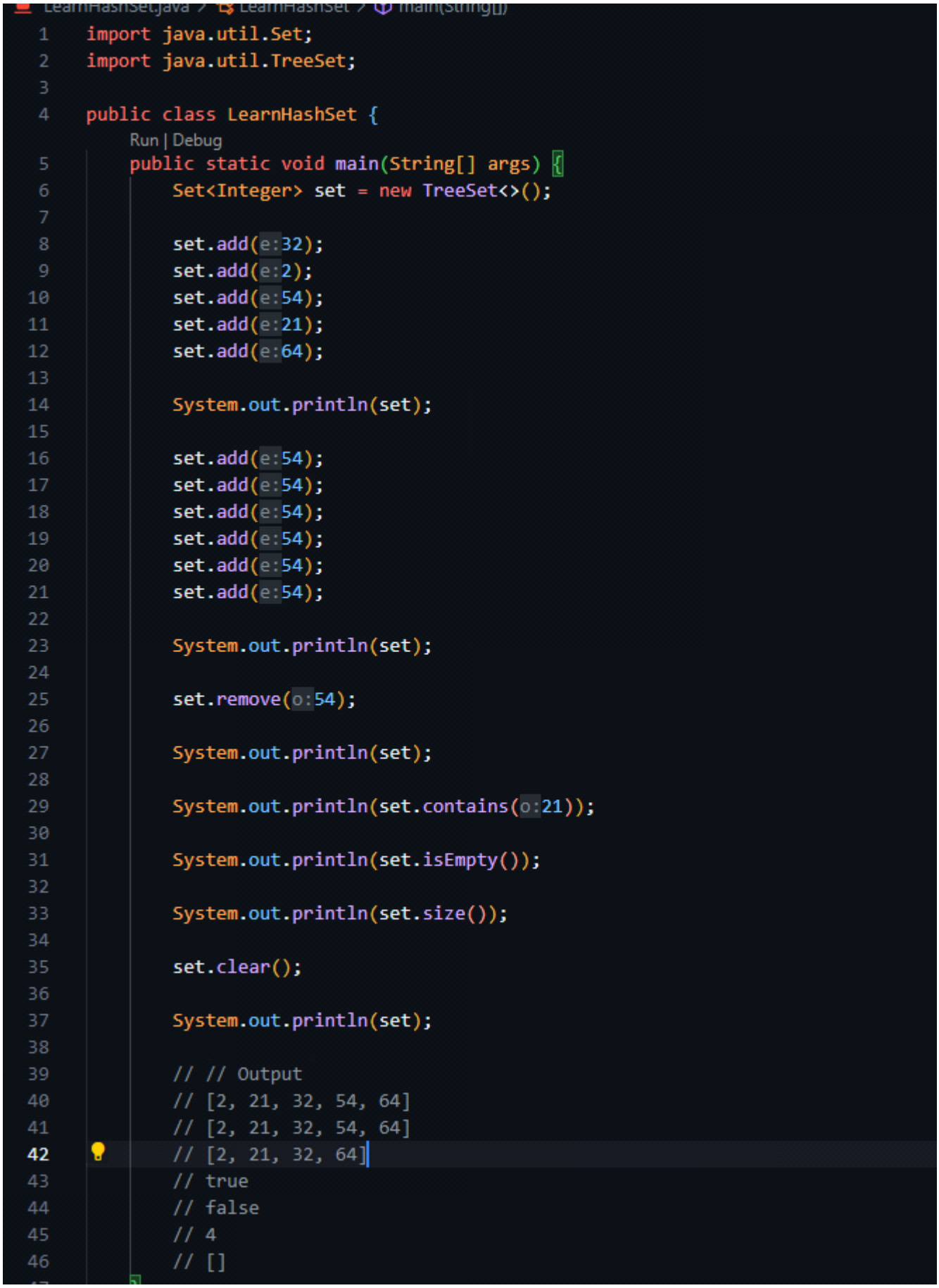
06 March 2024 16:47



```java
import java.util.LinkedHashSet;
import java.util.Set;

public class LearnHashSet {
    Run | Debug
    public static void main(String[] args) {
        Set<Integer> set = new LinkedHashSet<>();

        set.add(32);
        set.add(2);
        set.add(54);
        set.add(21);
        set.add(64);

        System.out.println(set);

        set.add(54);
        set.add(54);
        set.add(54);
        set.add(54);
        set.add(54);
        set.add(54);

        System.out.println(set);

        set.remove(54);

        System.out.println(set);

        System.out.println(set.contains(21));

        System.out.println(set.isEmpty());

        System.out.println(set.size());

        set.clear();

        System.out.println(set);

        // Output
        // [32, 2, 54, 21, 64]
        // [32, 2, 54, 21, 64]
        // [32, 2, 21, 64]
        // true
        // false
        // 4
        // []
```

TreeSet

06 March 2024 16:51



```java
import java.util.Set;
import java.util.TreeSet;

public class LearnHashSet {
    Run | Debug
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<>();

        set.add(e:32);
        set.add(e:2);
        set.add(e:54);
        set.add(e:21);
        set.add(e:64);

        System.out.println(set);

        set.add(e:54);
        set.add(e:54);
        set.add(e:54);
        set.add(e:54);
        set.add(e:54);
        set.add(e:54);

        System.out.println(set);

        set.remove(o:54);

        System.out.println(set);

        System.out.println(set.contains(o:21));

        System.out.println(set.isEmpty());

        System.out.println(set.size());

        set.clear();

        System.out.println(set);

        // // Output
        // [2, 21, 32, 54, 64]
        // [2, 21, 32, 54, 64]
        // [2, 21, 32, 64]
        // true
        // false
        // 4
        // []
```

HashMap

11 March 2024 11:57

```
Map <String,Integer> numberMap = new HashMap<>();

numberMap.put(key:"One", value:1);
numberMap.put(key:"Two", value:2);
numberMap.put(key:"Three", value:3);

System.out.println(numberMap);

// if (!numberMap.containsKey("Two")) {
//     numberMap.put("Two", 23);
// }


numberMap.putIfAbsent(key:"Two", value:23);
System.out.println(numberMap);

for (Map.Entry<String,Integer> e : numberMap.entrySet()){
    System.out.println(e);
    System.out.println("Key : " + e.getKey() + ",Value : " + e.getValue()
}

System.out.println(numberMap.keySet());
System.out.println(numberMap.values());

System.out.println(numberMap.containsValue(value:3));
System.out.println(numberMap.isEmpty());
```

TreeMap

12 March 2024 12:47

```java
Map <String,Integer> numberMap = new TreeMap<>();
numberMap.put(key:"One", value:1);
numberMap.put(key:"Two", value:2);
numberMap.put(key:"Three", value:3);

System.out.println(numberMap);

// if (!numberMap.containsKey("Two")) {
//     numberMap.put("Two", 23);
// }

numberMap.putIfAbsent(key:"Two", value:23);
System.out.println(numberMap);

for (Map.Entry<String,Integer> e : numberMap.entrySet()){
    System.out.println(e);
    System.out.println("Key : " + e.getKey() + ",Value : " + e.getValue());
}

System.out.println(numberMap.keySet());
System.out.println(numberMap.values());

System.out.println(numberMap.containsValue(value:3));
System.out.println(numberMap.isEmpty());

numberMap.remove(key:"Three");
System.out.println(numberMap);

numberMap.clear();
System.out.println(numberMap);

// Output
// {One=1, Two=2, Three=3}
// {One=1, Two=2, Three=3}
// One=1
// Key : One,Value : 1
// Two=2
// Key : Two,Value : 2
// Three=3
// Key : Three,Value : 3
// [One, Two, Three]
// [1, 2, 3]
// true
// false
// {}
```

ArrayClass

12 March 2024 12:53

```java
public class LearnArrayClass {
    Run | Debug
    public static void main(String[] args) {

        int [] numbers = {1,2,3,4,5,6,7,8,9,10};
        int index = Arrays.binarySearch(numbers, key:4);

        System.out.println(index);

        Integer [] valuIntegers = {5, 2, 9, 1, 7, 4, 8, 3, 6, 10};
        Arrays.sort(valuIntegers);

        for (Integer integer : valuIntegers) {
            System.out.print(integer + " ");
        }

        System.out.println();
        Arrays.fill(valuIntegers,val:13);

        for (Integer integer : valuIntegers) {
            System.out.print(integer + " ");
        }
    }
}
```

Collection Class

22 March 2024 12:11

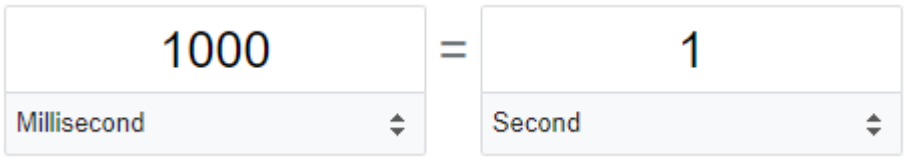
```java
/**
 * LearnCollectionsClass
 */
public class LearnCollectionsClass {

    Run | Debug
    public static void main(String[] args) {

        List <Integer> list = new ArrayList<>();
        list.add(e:34);
        list.add(e:12);
        list.add(e:9);
        list.add(e:76);
        list.add(e:29);
        list.add(e:75);

        System.out.println("min : " + Collections.min(list));
        System.out.println("max : " + Collections.max(list));
        System.out.println("frequency (9) : " + Collections.frequency(list,o:9));

        System.out.println(list);
        Collections.sort(list);
        System.out.println(list);
        Collections.sort(list,Comparator.reverseOrder());
        System.out.println(list);

        // min : 9
        // max : 76
        // frequency (9) : 1
        // [34, 12, 9, 76, 29, 75]
        // [9, 12, 29, 34, 75, 76]
        // [76, 75, 34, 29, 12, 9]
    }
}
```

Thread

26 March 2024 11:34

Advanced Java > Threads.java > Threads > main(String[])

```java
class A extends Thread{
    public void run(){
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread 1 : " + i);
            try {
                Thread.sleep(millis:10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class B extends Thread{
    public void run(){
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread 2 : " + i);
            try {
                Thread.sleep(millis:10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Threads {
    Run | Debug
    public static void main(String[] args) {
        A obj1 = new A();
        B obj2 = new B();

        System.out.println(obj1.getPriority()); // Default Priority = 5
        System.out.println(obj2.getPriority());

        obj2.setPriority(Thread.MAX_PRIORITY);

        obj1.start();
        try {
            Thread.sleep(millis:10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        obj2.start();
    }
}
```


1000
=
1
Millisecond
Second

Runnable

26 March 2024 12:49

```java
class A implements Runnable{
    public void run(){
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread 1 : " + i);
            try {
                Thread.sleep(millis:1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class B implements Runnable{
    public void run(){
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread 2 : " + i);
            try {
                Thread.sleep(millis:1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Threads {
    Run | Debug
    public static void main(String[] args) {
        Runnable obj1 = new A();
        Runnable obj2 = new B();

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);

        t1.start();
        try {
            Thread.sleep(millis:1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        t2.start();
    }
}
```

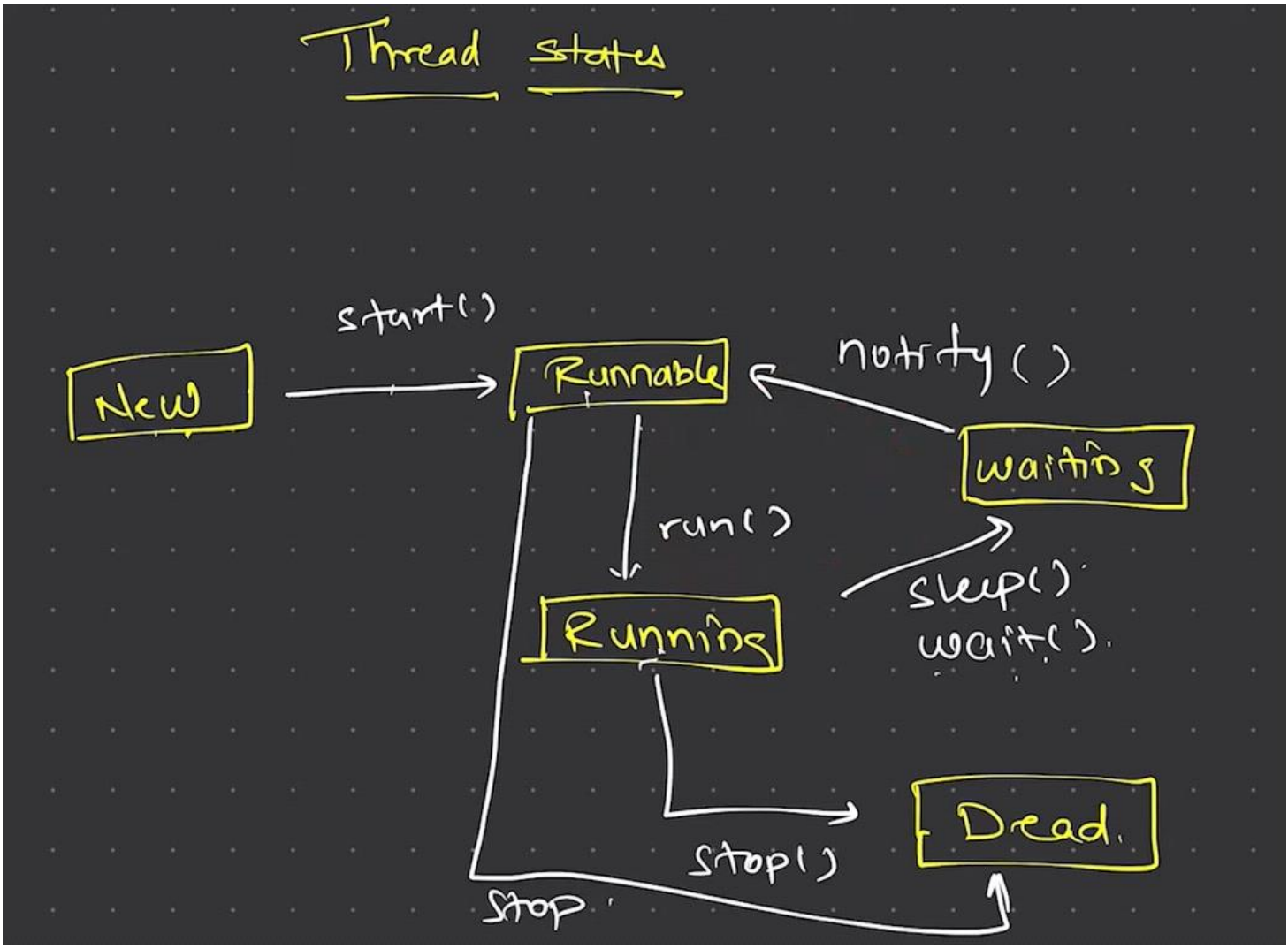
Race Condition

26 March 2024 17:30

```java
class Counter{
    private int count;
    public int getCount() {
        return count;
    }
    public synchronized void increment(){
        count++;
    }
}

public class Threads {
    Run | Debug
    public static void main(String[] args) {

        Counter c = new Counter();

        Runnable obj1 = () -> {
            for (int i = 1; i <= 10000; i++) {
                c.increment();
            }
        };

        Runnable obj2 = () -> {
            for (int i = 1; i <= 10000; i++) {
                c.increment();
            }
        };

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);

        t1.start();
        t2.start();

        try {
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(c.getCount());
    }
```

Thread State

26 March 2024 18:06



Class Loaders

27 March 2024 10:05

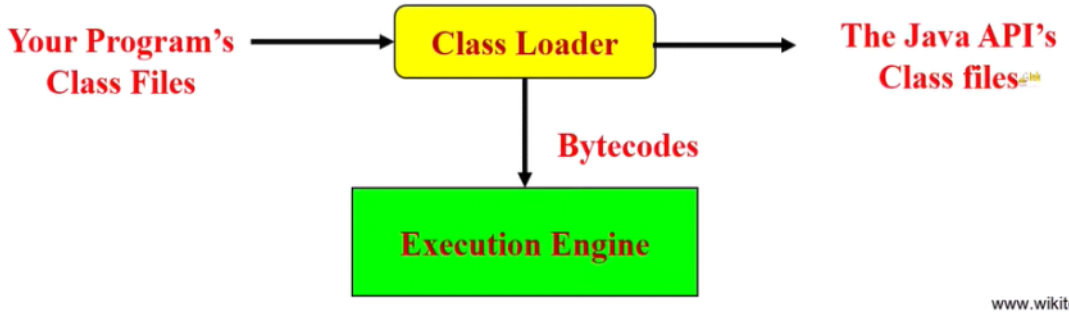
TCS

What is class loader?

- ✓ The class loader is a subsystem of JVM that is used to load classes and interfaces. There are many types of class loaders

Example:

Bootstrap class loader, Extension class loader, System class loader, Plugin class loader etc.



GTEK

www.wikitechy.com

Streams

27 March 2024 15:08

```java
public class Streams {
    public static void main(String[] args) {
        List <Integer> numbers = Arrays.asList(...a:4,5,7,3,2,6);
        System.out.println(numbers);

        int sum = 0;
        for (Integer integer : numbers) {
            if (integer % 2 == 0) {
                integer = integer * 2;
                sum = sum + integer;
            }
        }

        for (int i = 0; i < numbers.size(); i++) {
            System.out.print(numbers.get(i) + " ");
        }
        System.out.println();

        for (Integer integer : numbers) {
            System.out.print(integer + " ");
        }
        System.out.println();

        numbers.forEach(n -> System.out.print(n + " "));
        System.out.println();

        System.out.println(sum);

        Consumer <Integer> con = new Consumer <Integer>(){
            public void accept(Integer n){
                System.out.print(n + " ");
            }
        };

        numbers.forEach(con);
        System.out.println();

        Consumer <Integer> consumer = (Integer n) -> System.out.print(n + " ");

        numbers.forEach(consumer);
        System.out.println();

        Consumer <Integer> consumer1 = (n) -> System.out.print(n + " ");

        numbers.forEach(consumer1);
        System.out.println();

        // Final
```

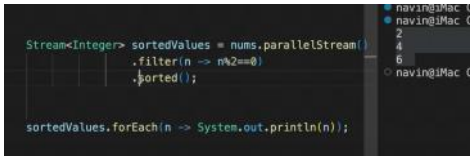
```java
Stream <Integer> stream = numbers.stream();
// stream.forEach(n -> System.out.print(n + " "));
// Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
// at java.base/java.util.stream.AbstractPipeline.sourceStageSpliterator(AbstractPipeline.java:279)
// at java.base/java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:762)
// at Streams.main(Streams.java:57)

Stream <Integer> stream2 = stream.filter(n -> n % 2 == 0);
Stream <Integer> stream3 = stream2.map(n -> n * 2);

int res = stream3.reduce(identity:0, (c,e) -> c + e);
System.out.println(res);
// stream3.forEach(n -> System.out.print(n + " "));
```

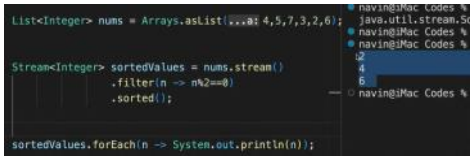
```
+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Z8STD\AppData\Roaming\Code\User\workspa
9b3a2858b4ce5ea1c53861dd12e4d6\redhat.java\jdt_ws\Java_7bd61535\bin' 'Streams'
java.util.stream.ReferencePipeline$2@3f99bd52
PS C:\Code Files\Java>  c:; cd 'c:\Code Files\Java'; & 'C:\Program Files\Java\jdk-17\bin\jav
+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Z8STD\AppData\Roaming\Code\User\workspa
9b3a2858b4ce5ea1c53861dd12e4d6\redhat.java\jdt_ws\Java_7bd61535\bin' 'Streams'
4 2 6
PS C:\Code Files\Java>  c:; cd 'c:\Code Files\Java'; & 'C:\Program Files\Java\jdk-17\bin\jav
+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Z8STD\AppData\Roaming\Code\User\workspa
9b3a2858b4ce5ea1c53861dd12e4d6\redhat.java\jdt_ws\Java_7bd61535\bin' 'Streams'
8 4 12
PS C:\Code Files\Java>  c:; cd 'c:\Code Files\Java'; & 'C:\Program Files\Java\jdk-17\bin\jav
+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Z8STD\AppData\Roaming\Code\User\workspa
9b3a2858b4ce5ea1c53861dd12e4d6\redhat.java\jdt_ws\Java_7bd61535\bin' 'Streams'
24
```

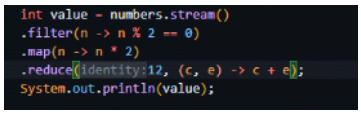
```java
int value = numbers.stream()
.filter(n -> n % 2 == 0)
.map(n -> n * 2)
.reduce(identity:0, (c, e) -> c + e);
System.out.println(value);
```

```java
Function<Integer, Integer> fun = new Function<Integer,Inte

    public Integer apply(Integer n) {

        return n*2;
    }

};


int result = nums.stream()
                .filter(n -> n%2==0)
                .map(fun)
                .reduce(identity: 0, (c,e) -> c+e);
```

```java
Predicate<Integer> p = new Predicate<Integer>() {
    public boolean test(Integer n) {
        if(n%2==0)
            return true;
        else
            return false;
    }

};

int result = nums.stream()
                .filter(p)
                .map(n -> n*2)
                .reduce(identity: 0, (c,e) -> c+e);

System.out.println(result);
```

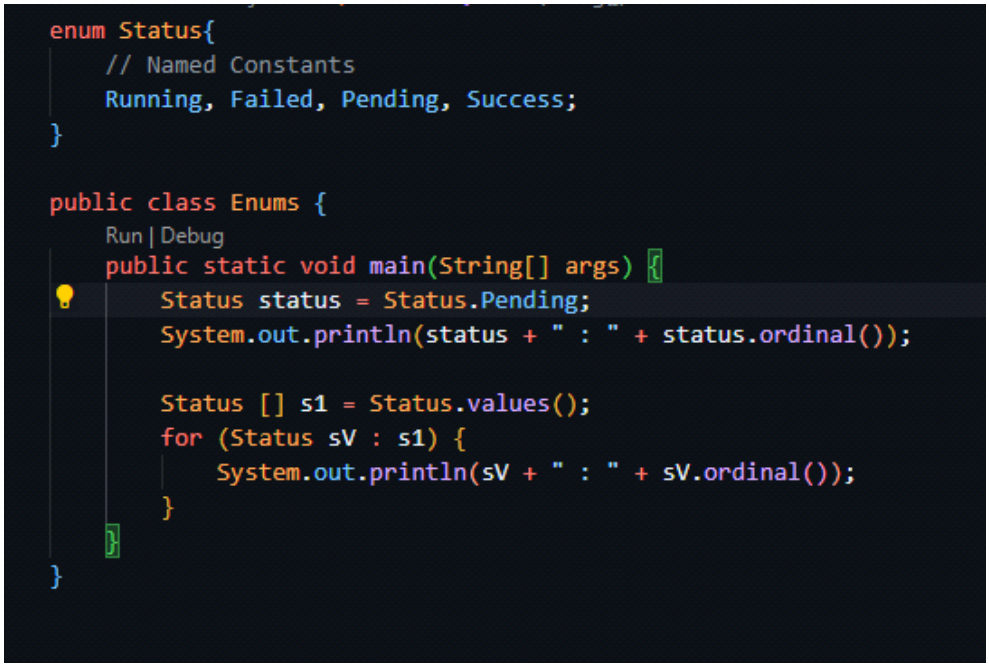

Stream<Integer> sortedValues = nums.parallelStream()
.filter(n -> n%2==0)
.sorted();
sortedValues.forEach(n -> System.out.println(n));
navin@iMac
navin@iMac
2
4
6
navin@iMac


List<Integer> nums = Arrays.asList(...a: 4,5,7,3,2,6);
Stream<Integer> sortedValues = nums.stream()
.filter(n -> n%2==0)
.sorted();
sortedValues.forEach(n -> System.out.println(n));
navin@iMac Codes %
java.util.stream.Sc
navin@iMac Codes %
navin@iMac Codes %
2
4
6
navin@iMac Codes %

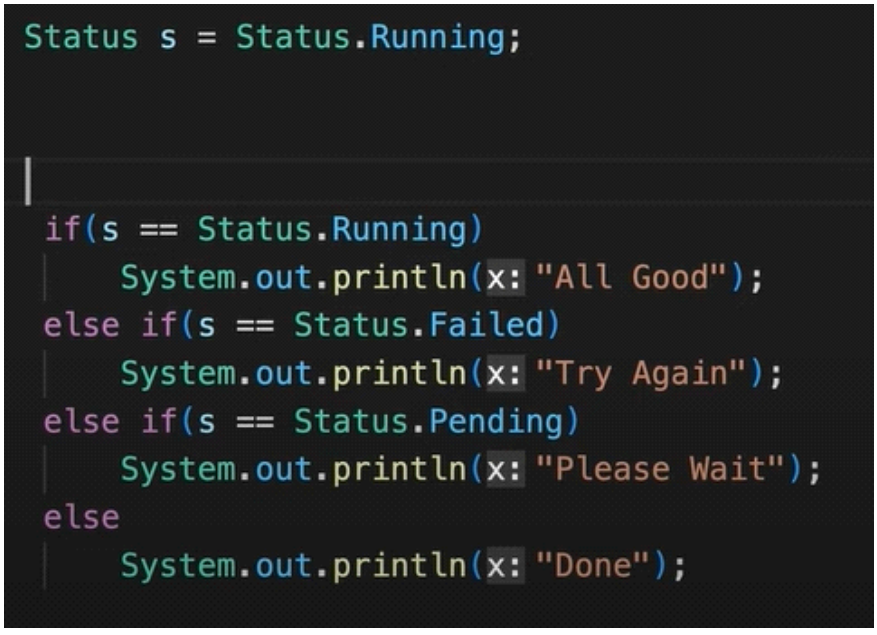

int value = numbers.stream()
.filter(n -> n % 2 == 0)
.map(n -> n * 2)
.reduce(identity:12, (c, e) -> c + e);
System.out.println(value);

Enums

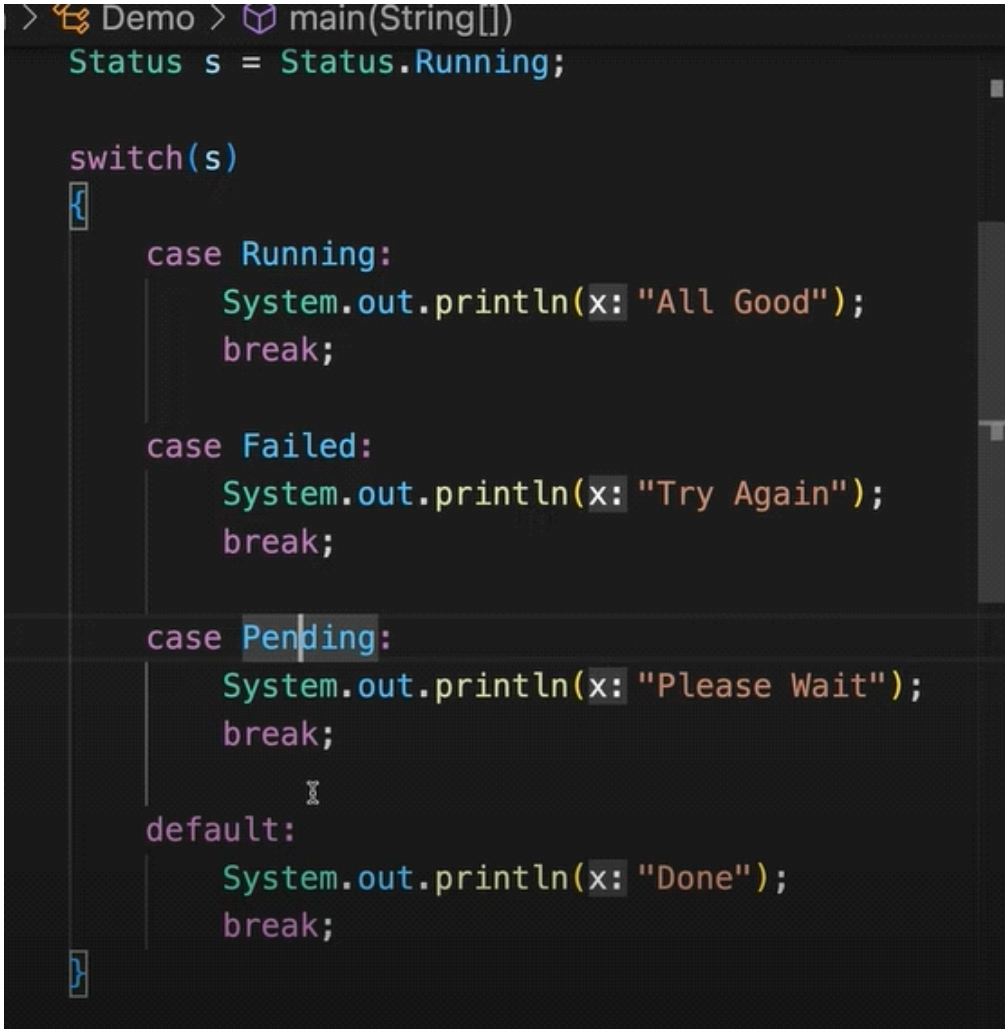
29 March 2024 15:13



```java
enum Status{
    // Named Constants
    Running, Failed, Pending, Success;
}

public class Enums {
    Run | Debug
    public static void main(String[] args) {
        Status status = Status.Pending;
        System.out.println(status + " : " + status.ordinal());

        Status [] s1 = Status.values();
        for (Status sV : s1) {
            System.out.println(sV + " : " + sV.ordinal());
        }
    }
}
```



```java
Status s = Status.Running;


if(s == Status.Running)
    System.out.println(x: "All Good");
else if(s == Status.Failed)
    System.out.println(x: "Try Again");
else if(s == Status.Pending)
    System.out.println(x: "Please Wait");
else
    System.out.println(x: "Done");
```


Demo > main(String[])
Status s = Status.Running;
switch(s)
{
case Running:
System.out.println(x: "All Good");
break;
case Failed:
System.out.println(x: "Try Again");
break;
case Pending:
System.out.println(x: "Please Wait");
break;
default:
System.out.println(x: "Done");
break;
}


System.out.println(s.getClass().getSuperclass());

dvanced Java > Enums.java > Laptop > Laptop()

```java
enum Laptop{
    // Create Laptop constructor with price input;
    Macbook(price:2000), XPS(price:2200), Surface, ThinkPad(price:1800);

    private int price;

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    private Laptop(int price) {
        this.price = price;
        System.out.println("Const Count : " + this.name());
    }

    private Laptop() {
        this.price = 500;
        System.out.println("Const Count : " + this.name());
    }
}

public class Enums {
    Run | Debug
    public static void main(String[] args) {
        // Laptop laptop = Laptop.Macbook;
        // System.out.println(laptop + " : $" + laptop.getPrice());

        for (Laptop laptop : Laptop.values()) {
            System.out.println(laptop + " : $" + laptop.getPrice());
        }
    }
}
```

Annotations

29 March 2024 15:42

```java
@Deprecated
class A{
    public void showMethod(){
        System.out.println(x:"In A");
    }
}

class B extends A{
    @Override
    public void showMethod(){
        System.out.println(x:"In B");
    }
}

public class Annotations {
    Run | Debug
    public static void main(String[] args) {
        B obj = new B();
        obj.showMethod();

        // FutureRefactor in SAC (Variables also).
    }
}
```

Functional Interface

29 March 2024 15:52

```java
@FunctionalInterface
interface AB{
    void show();
    // Only single un-implemented method;
}

// class BC implements AB{
//     @Override
//     public void show() {
//         System.out.println("AB implmented in BC");
//     }
// }

public class FuncInter {
    Run | Debug
    public static void main(String[] args) {
        // AB obj = new BC();
        AB obj = new AB() {
            @Override
            public void show(){
                System.out.println(x:"AB implmented in BC");
            }
        };
        obj.show();
    }
}

```

Lambda Expressions

29 March 2024 14:58

```java
@FunctionalInterface
interface AB {
    void show(int integer);
    // Only single un-implemented method;
}

// class BC implements AB{
// @Override
// public void show() {
// System.out.println("AB implmented in BC");
// }
// }

public class FuncInter {
    Run | Debug
    public static void main(String[] args) {
        // AB obj = new BC();
        // Lambda Expression see FunctionalInterface Link
        AB obj = integer -> System.out.println("AB implmented and int value is : " + integer);
        obj.show(integer:5);
    }
}
```

```java
@FunctionalInterface
interface AB {
    int add(int a, int b);
    // Only single un-implemented method;
}

// class BC implements AB{
// @Override
// public void show() {
// System.out.println("AB implmented in BC");
// }
// }

public class FuncInter {
    Run | Debug
    public static void main(String[] args) {
        // AB obj = new BC();
        // Lambda Expression see FunctionalInterface Link
        // AB obj = integer -> System.out.println("AB implmented and int value is : " + integer);
        // obj.show(5);

        AB obj = (a,b) -> a + b;
        System.out.println(obj.add(a:1, b:2));
    }
}
```

Date & Time Packages

01 April 2024 10:49

```java
import java.time.*;
import java.time.format.*;
import java.time.temporal.*;

public class DateTimeExample {
    Run | Debug
    public static void main(String[] args) {
        // LocalDate
        LocalDate today = LocalDate.now();
        System.out.println("LocalDate: " + today);

        // LocalTime
        LocalTime now = LocalTime.now();
        System.out.println("LocalTime: " + now);

        // LocalDateTime
        LocalDateTime currentDateTime = LocalDateTime.now();
        System.out.println("LocalDateTime: " + currentDateTime);

        // ZonedDateTime
        ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId.of(zoneId:"America/New_York"));
        System.out.println("ZonedDateTime: " + zonedDateTime);

        // Instant
        Instant instant = Instant.now();
        System.out.println("Instant: " + instant);

        // Duration
        Duration duration = Duration.ofHours(hours:2);
        System.out.println("Duration: " + duration);

        // Period
        Period period = Period.ofMonths(months:3);
        System.out.println("Period: " + period);

        // TemporalAdjuster
        TemporalAdjuster nextTuesday = TemporalAdjusters.next(DayOfWeek.TUESDAY);
        LocalDate nextTuesdayDate = today.with(nextTuesday);
        System.out.println("Next Tuesday: " + nextTuesdayDate);

        // TemporalAmount
        TemporalAmount amount = Duration.ofDays(days:5);
        LocalDateTime futureDateTime = currentDateTime.plus(amount);
        System.out.println("Future DateTime: " + futureDateTime);
```

```java
public static void main(String[] args) {
    // Period
    Period period = Period.ofMonths(months:3);
    System.out.println("Period: " + period);

    // TemporalAdjuster
    TemporalAdjuster nextTuesday = TemporalAdjusters.next(DayOfWeek.TUESDAY);
    LocalDate nextTuesdayDate = today.with(nextTuesday);
    System.out.println("Next Tuesday: " + nextTuesdayDate);

    // TemporalAmount
    TemporalAmount amount = Duration.ofDays(days:5);
    LocalDateTime futureDateTime = currentDateTime.plus(amount);
    System.out.println("Future DateTime: " + futureDateTime);

    // DateTimeFormatter
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern:"yyyy-MM-dd HH:mm:ss");
    String formattedDateTime = currentDateTime.format(formatter);
    System.out.println("Formatted DateTime: " + formattedDateTime);

    // ChronoUnit
    long daysBetween = ChronoUnit.DAYS.between(LocalDate.of(year:2023, month:1, dayOfMonth:1), LocalDate.of(year:2024, month:1, dayOfMonth…1));
    System.out.println("Days between 2023-01-01 and 2024-01-01: " + daysBetween);

    // Clock
    Clock clock = Clock.systemUTC();
    Instant clockInstant = Instant.now(clock);
    System.out.println("Clock Instant: " + clockInstant);

    // Parsing Example
    String dateStr = "2024-04-01";
    LocalDate parsedDate = LocalDate.parse(dateStr);
    System.out.println("Parsed date: " + parsedDate);

    // Period Example
    LocalDate startDate = LocalDate.of(year:2020, month:1, dayOfMonth:1);
    LocalDate endDate = LocalDate.of(year:2022, month:12, dayOfMonth:31);
    Period period1 = Period.between(startDate, endDate);
    System.out.println("Period between " + startDate + " and " + endDate + ": " + period1);

    // Duration Example
    LocalTime startTime = LocalTime.of(hour:10, minute:0);
    LocalTime endTime = LocalTime.of(hour:12, minute:30);
    Duration duration1 = Duration.between(startTime, endTime);
    System.out.println("Duration between " + startTime + " and " + endTime + ": " + duration1);
```

Memory management - Heap vs stack

29 March 2024 17:28

Heap Memory:

- **What is it?**
  - The heap is like a big storage space in your computer's memory.
  - It's where Java stores objects created with the `new` keyword.
- **How does it work?**
  - When you create an object, Java finds space for it in the heap.
  - The object stays there until you don't need it anymore or until Java decides to clean it up (garbage collection).
- **Diagram:**

diff Copy code

```
+----------------------+
|       Object 1       |
+----------------------+
|       Object 2       |
+----------------------+
|       Object 3       |
+----------------------+
           ...
```

Stack Memory:

- **What is it?**
  - The stack is like a stack of plates.
  - It's used for keeping track of what your program is doing.
- **How does it work?**
  - Every time you call a method, Java puts a "plate" on the stack with information about the method.
  - When the method finishes, Java takes the "plate" off the stack.
- **Diagram:**

sql Copy code

```
|       Method A       |
|----------------------|
|    Method B (current method)   |
|----------------------|
|       Method C       |
|----------------------|
```

- **Additional Points:**
  - The stack is small and limited in size.
  - It's very fast because adding or removing a "plate" is quick.
  - Primitive data types (like `int`, `float`, etc.) and references to objects are stored on the stack.
- **Comparison:**
  - **Heap**: Stores objects, grows as needed, managed by garbage collector.
  - **Stack**: Stores method calls, fixed size, ↓ aged automatically.

Best Practices

29 March 2024 18:28

Certainly! Here's a comprehensive list of best practices in Java:

1. **Use Meaningful Variable Names:**
   - Choose descriptive names that convey the purpose of the variable.

2. **Follow Java Naming Conventions:**
   - Use camelCase for variable names, PascalCase for class names, and UPPER_CASE for constants.

3. **Use Generics:**
   - Utilize generics to ensure type safety and code reusability.

4. **Avoid Magic Numbers:**
   - Declare constants instead of using magic numbers to enhance code readability and maintainability.

5. **Use Immutable Classes:**
   - Make classes immutable when possible to prevent unintended changes and improve thread safety.

6. **Minimize Mutability:**
   - Reduce the mutability of objects to prevent unexpected side effects and make code easier to reason about.

7. **Use Interfaces and Abstract Classes:**
   - Favor interfaces and abstract classes for defining contracts and promoting code extensibility.

8. **Handle Exceptions Properly:**
   - Catch exceptions at an appropriate level, and handle or propagate them accordingly.

9. **Follow SOLID Principles:**
   - Aim for code that is:
     - Single Responsibility: Each class should have only one reason to change.
     - Open/Closed: Open for extension, closed for modification.
     - Liskov Substitution: Subtypes should be substitutable for their base types.
     - Interface Segregation: Clients should not be forced to depend on interfaces they don't use.
     - Dependency Inversion: High-level modules should not depend on low-level modules; both should depend on abstractions.

10. **Use Design Patterns Wisely:**
    - Apply common design patterns like Singleton, Factory, Strategy, etc., where appropriate to solve recurring design problems.

11. **Optimize Loops:**
    - Minimize unnecessary looping and use optimized looping constructs like enhanced for loop or streams.

12. **Avoid Nested Loops:**
    - Refactor nested loops to improve code readability and maintainability.

13. **Close Resources Properly:**
    - Use try-with-resources or ensure resources are closed in a finally block to prevent resource leaks.

14. **Use StringBuilder for String Manipulation:**
    - Use StringBuilder instead of concatenation for efficient string manipulation.

15. **Use Enums for Constants:**
    - Use enums to represent fixed sets of constants, improving type safety and readability.

16. **Document Your Code:**
    - Write clear and concise comments to explain the purpose and behavior of your code.

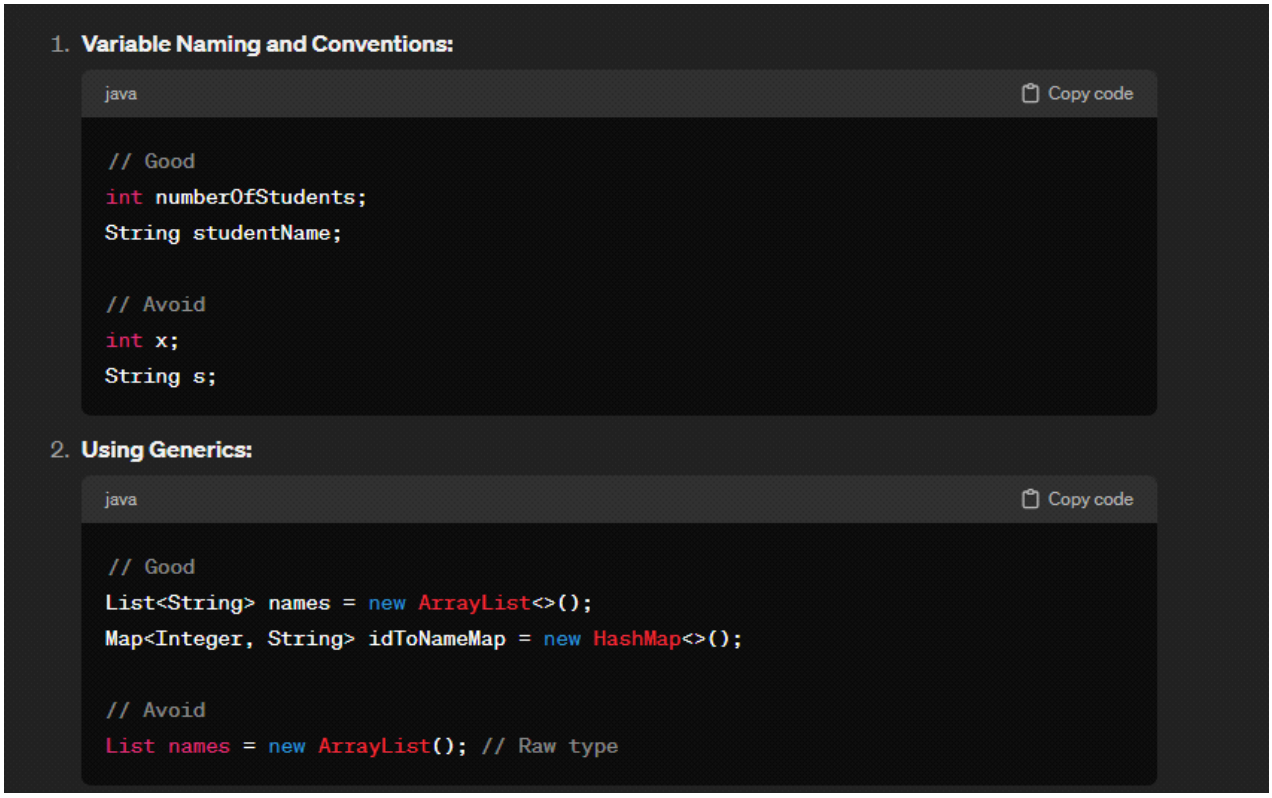
17. **Test Your Code:**
    - Write unit tests to ensure the correctness of your code and facilitate future refactoring.

18. **Follow DRY Principle:**
    - Don't Repeat Yourself. Refactor duplicate code into reusable methods or classes.

19. **Version Control:**
    - Use version control systems like Git to track changes and collaborate with others effectively.

20. **Keep Up with Java Updates:**
    - Stay updated with the latest Java features, enhancements, and best practices.

By adhering to these best practices, you can write cleaner, more maintainable, and efficient Java code.



3. **Avoiding Magic Numbers:**

java Copy code

```java
// Good
final int MAX_ATTEMPTS = 3;
if (attempts <= MAX_ATTEMPTS) {
    // Code
}

// Avoid
if (attempts <= 3) {
    // Code
}
```

4. **Using Immutable Classes:**

java Copy code

```java
// Good
final class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}

// Avoid
class Person {
    private String name;
    public String getName() {
        return name;
    }
}
```

5. **Minimizing Mutability:**

java Copy code

```java
// Good
private final List<String> colors = Collections.unmodifiableList(Arrays.asList("red"

// Avoid
private List<String> colors = new ArrayList<>();
```

6. **Handling Exceptions Properly:**

java Copy code

```java
// Good
try {
    // Code that may throw an exception
} catch (IOException e) {
    // Handle or log the exception
}

// Avoid
try {
    // Code
} catch (Exception e) {
    // Handle all exceptions
}
```

7. **Using Interfaces and Abstract Classes:**

java Copy code

```java
// Good
interface Vehicle {
    void start();
    void stop();
}

// Avoid
class Vehicle {
    void start() {}
    void stop() {}
}
```

8. **Using Design Patterns:**

java Copy code

```java
// Factory Pattern
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class ShapeFactory {
    public static Shape getShape(String type) {
        if (type.equalsIgnoreCase("circle")) {
            return new Circle();
        }
        // More shape implementations...
    }
}
```

9. **Optimizing Loops:**

java Copy code

```java
// Good
List<String> names = new ArrayList<>();
for (String name : names) {
    System.out.println(name);
}

// Avoid
for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}
```

10. **Closing Resources Properly:**

java Copy code

```java
// Good
try (FileInputStream fis = new FileInputStream("file.txt");
     InputStreamReader isr = new InputStreamReader(fis);
     BufferedReader br = new BufferedReader(isr)) {
    // Code to read from file
} catch (IOException e) {
    // Handle or log the exception
}

// Avoid
FileInputStream fis = null;
try {
    fis = new FileInputStream("file.txt");
    // Code
} finally {
    if (fis != null) {
        try {
            fis.close();
        } catch (IOException e) {
            // Handle or log the exception
        }
    }
}
```