the **Console** class provides methods for reading input from the console and writing output to the console.

import java.io.Console;

public class ConsoleExample {

public static void main(String[] args) {

Console console = System.console();

if (console != null) {

// Reading input from the console

String name = console.readLine("Enter your name: ");

char[] password = console.readPassword("Enter your password: ");

// Writing output to the console

console.printf("Hello, %s! Your password is: %s%n", name, new String(password));

// Clearing the password from memory

java.util.Arrays.fill(password, ' ');

} else {

System.out.println("No console available");

}

}

}

strong builder class in Java is a design pattern that provides a flexible and readable way to construct objects with complex initialization or configuration requirements. It is often used to create immutable objects with a large number of parameters.

public class Person {

private final String firstName;

private final String lastName;

private final int age;

private final String address;

private final String phoneNumber;

private Person(Builder builder) {

this.firstName = builder.firstName;

this.lastName = builder.lastName;

this.age = builder.age;

this.address = builder.address;

this.phoneNumber = builder.phoneNumber;

}

// Getters for the fields

public static class Builder {

private final String firstName;

private final String lastName;

private int age;

private String address;

private String phoneNumber;

public Builder(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

public Builder age(int age) {

this.age = age;

return this;

}

public Builder address(String address) {

this.address = address;

return this;

}

public Builder phoneNumber(String phoneNumber) {

this.phoneNumber = phoneNumber;

return this;

}

public Person build() {

return new Person(this);

}

}

// Usage Example

public static void main(String[] args) {

Person person = new Person.Builder("John", "Doe")

.age(30)

.address("123 Main St")

.phoneNumber("555-1234")

.build();

// Accessing the person's properties

System.out.println(person.getFirstName());

System.out.println(person.getLastName());

System.out.println(person.getAge());

System.out.println(person.getAddress());

System.out.println(person.getPhoneNumber());

}

}

In Java, there are several formatting techniques available to improve the readability and consistency of your code. Here are some commonly used formatting techniques:

1. Indentation: Use indentation to visually separate blocks of code and indicate their hierarchy. Generally, an indentation of four spaces is recommended for each level of indentation.
2. Line Length: Limit the length of lines to improve readability. The commonly recommended line length is around 80-120 characters. If a line exceeds the limit, break it into multiple lines using line continuation characters or by utilizing proper line wrapping.
3. Braces Placement: Place opening braces **{** on the same line as the associated statement or method declaration, and place closing braces **}** on a new line by themselves. For example:

if (condition) {

// code block

} else {

// code block

}

White Spaces: Use white spaces to enhance code readability. Add spaces around binary operators, after commas in method arguments, and between keywords and parentheses. For example:

int sum = num1 + num2;

System.out.println("Hello, " + name + "!");

1. Blank Lines: Insert blank lines to separate logical sections of code, improve readability, and provide visual clarity. Use blank lines to separate methods, classes, loops, and other code blocks.
2. Naming Conventions: Follow consistent naming conventions for classes, variables, methods, and constants. Use meaningful names that reflect the purpose or functionality of the element. For example, use **calculateSum()** instead of **calc()**.
3. Javadoc Comments: Include Javadoc comments to provide documentation for classes, methods, and important code sections. Use Javadoc tags to describe parameters, return values, and exceptions. Properly formatted and descriptive Javadoc comments enhance code readability and facilitate understanding.
4. Alignment: Align related code elements vertically to improve code organization and readability. For example, align variable declarations or method parameters for better visual grouping.

These formatting techniques can help make your code more readable, maintainable, and consistent across your projects. Consistent formatting practices improve code collaboration and make it easier for other developers to understand and work with your code.

Regular expressions in Java provide a powerful way to match, search, and manipulate strings based on specific patterns. Java provides the **java.util.regex** package, which includes classes for working with regular expressions. Here's a brief overview of regular expressions in Java:

1. Pattern Class: The **Pattern** class represents a compiled regular expression pattern. You can create a **Pattern** object using the **compile()** method, which takes the regular expression as a string. For example:

Pattern pattern = Pattern.compile("ab+c");

Matcher Class: The **Matcher** class is used to perform matching operations on a string using a compiled **Pattern**. You can create a **Matcher** object by invoking the **matcher()** method on a **Pattern** object and passing the input string. For example:

Matcher matcher = pattern.matcher("abc");

Basic Matching Operations: The **Matcher** class provides methods for basic matching operations such as **matches()**, **find()**, and **lookingAt()**. These methods return a boolean value indicating whether the pattern matches the input string. For example:

boolean isMatch = matcher.matches();

Searching and Extracting: The **Matcher** class provides methods for searching and extracting substrings based on the regular expression pattern. Methods like **find()** and **group()** are used for this purpose. For example:

while (matcher.find()) {

String matchedText = matcher.group();

// Process the matched text

}

Replacement: The **Matcher** class also supports string replacement using the **replaceAll()** and **replaceFirst()** methods. These methods allow you to replace parts of the input string that match the regular expression with specified replacement strings. For example:

String replacedText = matcher.replaceAll("xyz");