Course Level Coding Cheat Sheet

Welcome to this course-level coding sheet. This coding cheat sheet contains all of the code and explanations provide in the module-level cheat sheets. Like the module-level coding cheat sheets, you'll find code and explanations for all videos in the course that presented code. Within each video section, you'll find the code, organized by topics from within each video.

You can expand the hamburger menu in the upper left corner of this window to locate code by video name.

Here's list of the videos with code included in this reading.

- Structuring Java Code and Comments
- Exploring Data Types in Java
- Introduction to Operators in Java
- Using Advanced Operators in Java
- Working with Arrays
- Using Conditional Statements
- Introduction to Loops in Java
- Working with Strings in Java
- Using Packages and Imports
- Implementing Functions and Methods
- An Introduction to Exceptions
- Using Finally Block
- Using Multiple Try Catch
- Checked and Runtime Exceptions Compared

Structuring Java Code and Comments

Types of comments

Description	Code Example
A Java statement used to print text or other data to the standard output, typically the console.	`System.out.println("Hello, World!")`
Use two forward slashes to precede a single line comment in Java	`// This is a single-line comment.</pre`
All text after the two forward slash marks on this line is treated as a comment	`int number = 10; // This variable stores the number 10`
These comments start with `/*`	`/* This is a multi-line comment It can be used to explain a block of code or provide detailed information`

Description	Code Example
and end with `*/`. They can span multiple lines.	

This also is a multiline comment. Multiline comments start with `/*` and end with `*/`. They can span multiple lines.	
	```
	*/ int sum = 0;
	 */ This variable will hold the sum of numbers */.
	```

The documentation comments start with `/*` and ends with `*/` . These comments are used for generating documentation using tools like Javadoc.	
	```
	/*
	 This method calculates the square of a number.
	 @param number The number to be squared
	 @return The square of the input number
	 */
	public int square(int number) {
	 return number * number;
	 }
	```

Creating a package

Description	Code Example

To create a package, use the package keyword at the top of your Java source file.	
	```
	package com.example.myapp; // Declare a package
	public class MyClass {
	 // Class code goes here
	 }
	```

Folder structure for a package

Description	Folder Structure Example

The folder structure on your filesystem should match the package declaration. For instance, if your package is com.example.myapp, your source file should be located in the following path example.	
	```
	/src
	└── com
	└── example
	└──>└──>
	└──>└── └──>myapp
	└──>└── └──>└── MyClass.java
	```

Description	Folder Structure Example

Class and Methods Structure

Description	Code Example

In Java, a class is a blueprint for creating objects and can contain methods (functions) that define behaviors. For instance, the second line of this code displays the public class `library`, with methods like calculatePrice() or applyDiscount().	
	```
	 package com.example.library;
	 public class Library
	{
	private List <Book> books; // Private attribute to hold books
	 public Library()
	{
	books = new ArrayList<>(); // Initialize the list in the constructor
	 }
	}public void addBook(Book book) {
	books.add(book); // Method to add a book to the library
	} <br }
	```

Creating methods

Description	Code Example

Every Java application needs an entry point, which is typically a main method within a public class. In this code, the second line of code identifies the method named Main in the public class.	
	```
	package com.example.library;

	public class Main {

	public static void main(String[] args) {

	Library library = new Library(); // Create an instance of Library

	// Add books to the library
	 library.addBook(new Book("1984", "George Orwell"));
	library.addBook(new Book("To Kill a Mockingbird", "Harper Lee"));

	// Display all books
	 library.displayBooks();
	}
	}
	```

Organizing source files in directories

Description	Directory Structure

As your project grows, organizing your source files into directories can keep your code manageable. The following	
	```
	 MyProject/
	└── src/     # Source code goes here
	└── lib/     # External libraries/JARs
	```

Description	Directory Structure

example illustrates typical Java organizaton.	`└── resources/     # Configuration files, images, and others`
	`└── doc/     # Documentation`
	`└── test/    # Test files`

Using imports

Description	Code Example

When you need to use classes from other packages, you need to import them at the top of your source file. These examples illustrate two examples of importing classes.	`import java.util.List;`
	
	`import java.util.ArrayList; // Importing classes from Java's standard library </pre`

Exploring Data Types in Java

Primitive data types

Description	Code Example
Use the `byte` data type when you need to save memory in large arrays where the memory savings are critical, and the values are between -128 and 127.	`byte age = 25; // Age of a person`
Use the `short` small integers data type for numbers from −32,768 to 32,767.	`short temperature = -5; // Temperature in degrees`

Description	Code Example
Use the `int` integer data type to store whole numbers larger than what byte and short can hold. This is the most commonly used integer type.	`int population = 1000000; // Population of a city`
Use the `long` data type when you need to store very large integer values that exceed the range of `int`.	`long distanceToMoon = 384400000L; // Distance in meters`
Use the `float` when you need to store decimal numbers but do not require high precision (for example, up to 7 decimal places).	`float price = 19.99f; // Price of a product`
Use the `double` data type when you need to store decimal numbers and require high precision (up to 15 decimal places).	`double pi = 3.141592653589793; // Value of Pi`
Use the `char` data type when you need to store a single character such as a single letter or an initial.	`char initial = 'A'; // Initial of a person's name`
Use `boolean` when you need to represent a true/false value. Boolean is often used for conditions and decisions.	`boolean isLoggedIn = true; // User login status`

Reference data types

Description	**Code Example**
A `string` data type is a sequence of characters. The `string` data type is very useful for handling text in your programs.	`String greeting = "Hello, World!";`
An `array` is a collection of multiple values stored under a single variable name. All the values in an array must be of the same type. Arrays are great for storing lists of items, like student scores or names. The following code defines an integer array type of scores that include 85, 90, 78, and 92.	`int[] scores = {85, 90, 78, 92};`

The reference data type `class` is like a blueprint for creating objects. You can see the class identified in the first line of code.	`class Car {`
	` String color;`
	` int year;`
	` void displayInfo() {`
	` System.out.println("Color: " + color + ", Year: " + year);`
	` }`
	`}`

Objects are classes that contain both data and functions. In this code, `Car myCar = new Car();` is the object.	`public class Main {`
	` public static void main(String[] args) {`
	`    Car myCar = new Car();`
	`     myCar.color = "Red";`
	`     myCar.year = 2026;`
	`     myCar.displayInfo(); // Output: Color: Red, Year: 2026`
	`   }`
	`}`

When you create an interface, you only declare the methods without providing their actual code. All methods in an interface are empty by default. Here's an example of an interface called MyInterfaceClass with three methods.	`// The interface class`
	`interface MyInterfaceClass {`
	`     void methodExampleOne();`
	`     void methodExampleTwo();`
	`     void methodExampleThree();`
	`}`

Description	Code Example

An `enum` is a special data type that defines a list of named values. An enum is useful for representing fixed sets of options, such as days of the week or colors.	`enum DaysOfWeek {`
	`MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;`
	`}`

Introduction to Operators in Java

Arithemtic operators

The following operators perform basic mathematical functions.
In these code examples `int a = 10;` and `int b = 5;`

Description	Example
The plus symbol `+` performs addition operations.	`System.out.println("Addition: " + (a + b));        // 15`
The minus symbol `-` performs subtraction operations.	`System.out.println("Subtraction: " + (a - b));     // 5`
The asterisk symbol `*` performs multiplication operations.	`System.out.println("Multiplication: " + (a * b));  // 50`
The division symbol `/` performs division operations.	`System.out.println("Division: " + (a / b));         // 2`
The percentage symbol `%` performs modulus (percentage) operations.	`ystem.out.println("Modulus: " + (a % b));          // 0`

Description	Example

Relational operators

You use relational operators to compare two values. These operators return a boolean result (true or false).

In the following examples `int a = 10;` and `int b = 5;`.

Description	Example
The double equal symbols `==` check for equality.	`System.out.println("Is a equal to b? " + (a == b));      // false`
The exclamation point and equal symbol together `!=` check for a not equal result.	`System.out.println("Is a not equal to b? " + (a != b));  // true`
The greater than symbol `>` checks for the greater than result.	`System.out.println("Is a greater than b? " + (a > b));   // true 0`
The less than symbol `<` checks if the result for a less than state.	`System.out.println("Is a less than  b? " + (a < b)); // false`
The greater than equal symbols `>=` compares the values to check for a greater than or equal to result.	`System.out.println("Is a greater than or equal to b? " + (a >= b)); // true`

Description	Example
The less than equal symbols `<=` compares the values to check for a less than or equal to result.	`System.out.println("Is a less than or equal to b? " + (a <= b)); // false`

Logical operators

You can use logical operators to combine boolean expressions. The following code examples use `boolean x = true;` and `boolean y = false;`.

Description	Example
The double ampersands `&&` combines two boolean expressions (both must be true).	`if (a > b && b < c) { ... }`
The double vertical bar symbols `||` used with boolean values always evaluates both expressions, even if the first one is true.	`if (a > b || b < c) { ... }`
The single exclamation point symbol `!` operator in Java is the logical NOT operator. This operator is used to negate a boolean expression, meaning it reverses the truth value.	`if (!(a > b)) { ... }`

Using Advanced Operators in Java

Assignment operators

You can use assignment operators to assign values to a variable.

Description	Example
The single equal symbol `=` assigns the right operand to left	`a = 10`

Description	Example
The plus sign and equal symbols combined `+=` adds and assigns values to variables	`a += 5`
The minus sign and equal symbols combined `-=` subtracts values to variables	`a -= 2`
The asterisk sign and equal symbols combined `*=` multipies and assigns values to variables	`a *= 3`
The forward slash sign and equal symbols combined `/=` divides and assigns values to variables	`a /= 2`
The percentage and equal symbols combined `%=` take the modulus (percentage) and assigns values to the variables	`a %= 4`

Unary operators

A unary operation is a mathematical or programming operation that uses only one operand or input. Developers use unary operations to manipulate data and perform calculations. You would use the following assignment operators to assign values to variables.

Description	Example
Use the plus symbol to indicate a positive value.	`int positive = +a;`

Description	Example

	`int a = 10;`
	`System.out.println("Unary plus: " + (+a));   // 10`

Ternary operators

Ternary operators are a shorthand form of the conditional statement. They can use three operands.

Description	Example

Ternary operators uses the Syntax: condition ? expression1 : expression2;. In the following code, `int max = (a > b) ? a : b;`	`int a = 10;`
	`int b = 20;`
	`int max = (a > b) ? a : b; // If a is greater than b, assign a; otherwise assign b`
	`System.out.println("Maximum value is: " + max); // 20`

Working with Arrays

Declaring an array

Description	Example
To declare an array in Java, you use the following syntax: `dataType[] arrayName;`. The following doce displays the data type of int and creates an array named `numbers`.	`int[] numbers;`

Initializing an array

After you declare an array, you need to initialize the array to allocate memory for it.

Description	Example

These code samples display three methods used to create an array of 5 integers. You can initialize an array using the `new` keyword. You can also declare and initialize an array in a single line. Alternatively, you can create an array and directly assign values using curly braces.	`numbers = new int[5];`
	
	`int[] numbers = new int[5];`
	
	`int[] numbers = {1, 2, 3, 4, 5};`

Accessing an array

You can access individual elements in an array by specifying the index inside square brackets.

Description	Example

Remember that indices start at zero. Here are two examples:	`System.out.println(numbers[0]);System.out.println(numbers[0]); // Outputs: 1`
	
	`System.out.println(numbers[4]); // Outputs: 5`

Modifying array elements

Description	Example
Modify the array element value by accessing it within its index.	`numbers[2] = 10; // Changing the third element to 10` `System.out.println(numbers[2]);>System.out.println(numbers[2]); // Outputs: 10`

Verify the array length

Description	Example
Verify the array length by using the length property.	`System.out.println("The length of the array is: " + numbers.length);`

Using a for loop to iterate through an array

Description	Example

You can use a `for` loop for to iterarate through and array. Here's an example that prints all elements in the numbers array. The `for` loop includes this code `for (int i = 0` in the following code snippet.	`for (int i = 0; i < numbers.length; i++) {`
	`    System.out.println(numbers[i]);`
	`}`

Using a for each loop to iterate through an array

Description	Example

You can also use the enhanced `for` loop, known as the "for-each" loop. The enhanced `for each` loop code include this portion, `for (int` of the following code snippet.	`for (int number : numbers) {`
	`    System.out.println(number);`
	`}`

Description	Example

Declare and intialize a multidimensional array

Java supports multi-dimensional arrays, which are essentially arrays of arrays. The most common type is the two-dimensional array.

Description	Example

Here's how to declare and initialize a 2D array. You will declare the integer data type and create the matrix array.	`int[][] matrix = {`
	`    {1,{1, 2, 3},`
	`    {4, 5, 6},`
	`   {7, 8, 9}`
	`};`

You can access elements in a two-dimensional array by specifying both indices, which are shown in this code inside of the square brackets.	`System.out.println(matrix[0][1]); // Outputs: 2`

Iterating Through a 2D Array

Description	Example

You can use nested loops to iterate through all elements of a 2D array.	`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.print(matrix[i][j] + " ");`
	`}`
	`     System.out.println(); // Move to the next line after each row`

Using Conditional Statements

if statement

The `if` statement checks a condition. If the condition is `true`, it executes the code inside the block. If the condition is `false`, the program skips the `if` block.

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
A variable `number` of type `int` is declared and initialized with the value `10`.	`int number = 10;`

The `if` statement checks if `number` is greater than `5`. If `true`, it prints "The number is greater than 5."	`if (number > 5) {`
	`    System.out.println("The number is greater than 5.");`
	`}`

Closing curly braces to end the `main` method and class definition.	`    }`
	`}`

Explanation: Since `number` is `10`, which is greater than `5`, the condition evaluates to `true`, and the program prints `"The number is greater than 5."`.

---

if-else statement

The `if-else` statement gives an alternative if the condition is `false`.

Description	**Example**
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
A variable `number` of type `int` is declared and initialized with the value `3`.	`int number = 3;`

The `if` statement checks if `number` is greater than `5`. If `true`, it prints "The number is greater than 5.".	`if (number > 5) {`
	    `System.out.println("The number is greater than 5.");`
	`}`

The `else` block executes when the `if` condition is `false`, printing "The number is not greater than 5.".	`else {`
	    `System.out.println("The number is not greater than 5.");`
	`}`

Closing curly braces to end the `main` method and class definition.	`}` `}`

else if statement

You can check multiple conditions using `else if`.

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
A variable `number` of type `int` is declared and initialized with the value `5`.	`int number = 5;`

The `if` statement checks if `number` is greater than `5`. If `true`, it prints "The number is greater than 5.".	`if (number > 5) {`
	`    System.out.println("The number is greater than 5.");`
	`}`

The `else if` statement checks if `number` is exactly `5`. If `true`, it prints "The number is equal to 5.".	`else if (number == 5) {`
	`    System.out.println("The number is equal to 5.");`
	`}`

The `else` block executes when none of the above conditions are met, printing "The number is less than 5.".	`else {`
	`    System.out.println("The number is less than 5.");`
	`}`

Description	Example

Closing curly braces to end the `main` method and class definition.	`    }`
	 `}`

Explanation: Since `number` is exactly `5`, the program prints `"The number is equal to 5."`.

---

switch statement

A `switch` statement checks a single variable against multiple values.

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
A variable `day` of type `int` is declared and initialized with the value `3`.	`int day = 3;`

The `switch` statement checks the value of `day` and compares it against the `case` labels. If `day` is `3`, it prints `"Wednesday"`.	`switch (day) {`
	 `    case 1:`
	 `        System.out.println("Monday");`
	 `        break;`
	 `    case 2:`
	 `        System.out.println("Tuesday");`
	 `        break;`
	 `    case 3:`
	 `        System.out.println("Wednesday");`
	 `        break;`
	 `    case 4:`
	 `        System.out.println("Thursday");`
	 `        break;`

Description	Example

	`        case 5:`
	`            System.out.println("Friday");`
	`            break;`
	`        default:`
	`            System.out.println("Weekend");`
	`    }`

Closing curly braces to end the `main` method and class definition.	`        }`
	`    }`

---

default case in a switch statement

When using a `switch` statement, it's a good practice to specify a `default` case. This case runs if none of the specified cases match, acting as a fallback option.

Description	Example
A `switch` statement checks the value of a variable `color`.	`    switch (color) {`

A `case` checks if `color` is `"red"`, printing `"Color is red."`.	`        case "red":`
	`            System.out.println("Color is red.");`
	`            break;`

Description	Example

A `case` checks if `color` is `"blue"`, printing `"Color is blue."`.	
	```
	        case "blue":
	            System.out.println("Color is blue.");
	            break;
	```

The `default` case prints `"Unknown color."` if `color` doesn't match `"red"` or `"blue"`.	
	```
	        default:
	            System.out.println("Unknown color.");
	    }
	```

Nested Conditional Statements

You can place conditional statements within each other to create more complex decisions. The process of placing conditional statements within other conditional statements is called nesting.

Description	Example
A Java class named `Main` with a `main` method. The `main` method is the entry point of the program.	`public class Main {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
A variable `age` of type `int` is declared and initialized with the value `20`.	`int age = 20;`

The `if` statement checks if `age` is greater than or equal to `18`. If `true`, it prints `"You are an adult."`.	
	```
	if (age >= 18) {
	    System.out.println("You are an adult.");
	}
	```

Description	Example

Another `if` statement checks if `age` is greater than or equal to `65`, printing `"You are a senior citizen."` if true.	
	```
	if (age >= 65) {
	    System.out.println("You are a senior citizen.");
	}
	```

The `else` block executes if `age` is less than `18`, printing `"You are a minor."`.	
	```
	else {
	    System.out.println("You are a minor.");
	}
	```

Closing curly braces to end the `main` method and class definition.	
	```
	    }
	}
	```

Introduction to Loops in Java

for Loop

The for loop is used when the number of iterations is known beforehand. It consists of three parts:

- Initialization: This sets a counter variable.
- Condition: This checks if the loop should continue executing.
- Increment/Decrement: This updates the counter variable after each iteration.

Description	**Example**
A Java class named `ForLoopExample` with a `main` method. The `main` method is the entry point of the program.	`public class ForLoopExample {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
A for loop is initialized with `int i = 1`, which starts the counter at 1. The counter variable `i` is incremented by `i++` after each iteration.	`for (int i = 1; i <= 5; i++) {`
The loop checks if `i <= 5`, and if true, it prints the value of `i`.	`System.out.println(i);`
Close the `for` loop.	`}`
Close the `main` method.	`}`
Close the `ForLoopExample` class.	`}`

Description	Example

while Loop

The while loop is used when the number of iterations is not known in advance. It continues executing as long as the specified condition remains true.

Description	Example
A Java class named `WhileLoopExample` with a `main` method. The `main` method is the entry point of the program.	`public class WhileLoopExample {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
A variable `i` is initialized to `1`.	`int i = 1;`
The while loop continues as long as `i <= 5`.	`while (i <= 5) {`

Inside the loop, the value of `i` is printed, then incremented by `i++`.	`System.out.println(i);`
	`i++;`

The `main` method and class are closed with curly braces.	`}` `}`

Description	Example

do-while Loop

The do-while loop is similar to the while loop but guarantees that the code block executes at least once before checking the condition.

Description	Example
A Java class named `DoWhileLoopExample` with a `main` method. The `main` method is the entry point of the program.	`public class DoWhileLoopExample {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
A variable `i` is initialized to `1`.	`int i = 1;`
The do-while loop starts and executes at least once before checking if `i <= 5`.	`do {`

Inside the loop, the value of `i` is printed, then incremented by `i++`.	`System.out.println(i);`
	`i++;`

Description	Example
The condition `i <= 5` is checked after each iteration.	`} while (i <= 5);`
The `main` method and class are closed with curly braces.	`}` `}`

Nested loops

You can also use loops inside other loops, known as nested loops. This is useful for working with multi-dimensional data structures, like arrays or matrices.

Description	Example
A Java class named `NestedLoopsExample` with a `main` method. The `main` method is the entry point of the program.	`public class NestedLoopsExample {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
The outer loop controls the rows, running 10 times.	`for (int i = 1; i <= 10; i++) {`
The inner loop controls the columns, also running 10 times for each row.	`for (int j = 1; j <= 10; j++) {`

Description	Example
The product of `i * j` is printed for each combination of rows and columns.	`System.out.print(i * j + "\t");`
After the inner loop, a newline is printed to separate the rows.	`System.out.println();`

The `main` method and class are closed with curly braces.	`}`
	 `}`

break statement

The break statement is used to terminate a loop immediately, regardless of the loop's condition. This can be useful when you want to exit a loop based on a specific condition that may occur during its execution.

Description	Example
A Java class named `BreakExample` with a `main` method. The `main` method is the entry point of the program.	`public class BreakExample {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`
An array `numbers` is declared and initialized.	`int[] numbers = {1, 3, 5, 7, 9, 2, 4};`

Description	Example
The loop iterates through the array, checking if any number is greater than 5.	`for (int num : numbers) {`

When a number greater than 5 is found, the loop exits with the `break` statement.	`if (num > 5) {`
	`    break;`

The `main` method and class are closed with curly braces.	`}` `}`

continue statement

The continue statement is used to skip the current iteration of a loop and move to the next iteration. It's useful when you want to skip certain conditions but continue executing the rest of the loop.

Description	Example
A Java class named `ContinueExample` with a `main` method. The `main` method is the entry point of the program.	`public class ContinueExample {`
The `main` method is declared using `public static void main(String[] args)`. This method is required for execution in Java programs.	`public static void main(String[] args) {`

Description	Example
The loop iterates through the numbers from 1 to 10.	`for (int i = 1; i <= 10; i++) {`

When `i == 5`, the `continue` statement is executed, skipping the `System.out.println(i);` statement for that iteration.	`if (i == 5) {`
	 `    continue;`

The value of `i` is printed for all numbers except 5.	`System.out.println(i);`

The `main` method and class are closed with curly braces.	`}`
	 `}`

Working with Strings in Java

Creating strings

You can create a string in Java in two main ways:

Using string literals: This means writing the string directly inside double quotes.

Description	Example
Create a string using string literal.	`String greeting = "Hello, World!";`

In this example, we created a string called greeting that contains "Hello, World!".

Using the new Keyword: This method involves using the new keyword to create a string object.

Description	Example
Create a string using the new keyword.	`String message = new String("Hello, World!");`

Although this works, it's more common to use string literals because it's simpler.

String length

To find out how many characters are in a string, you can use the length() method. This method tells you the total number of characters in the string.

Description	Example
Create a string and use length() to get the number of characters.	`String text = "Java Programming";`
Use the length() method to find the string length.	`int length = text.length();`
Print the length of the string.	`System.out.println("Length of the string: " + length); // Output: 16`

Here, we created a string called text and then checked its length. The output tells us that there are 16 characters in "Java Programming".

Accessing characters

If you want to look at individual characters in a string, you can use the charAt() method. This method allows you to get a character at a specific position in the string.

Description	Example
Create a string and access a character using charAt().	`String word = "Java";`
Access the first character of the string.	`char firstChar = word.charAt(0);`
Print the first character of the string.	`System.out.println("First character: " + firstChar); // Output: J`

In this example, we accessed the first character of the string "Java". Remember that counting starts at 0, so charAt(0) gives us 'J'.

Concatenating strings

Sometimes you might want to combine two or more strings together. You can do this easily using the + operator or the concat() method.

Description	Example
Combine two strings using the + operator.	`String firstName = "John";`
Combine two strings using the + operator.	`String lastName = "Doe";`
Concatenate first and last names using the + operator.	`String fullName = firstName + " " + lastName;`

Description	Example
Print the full name.	`System.out.println("Full Name: " + fullName); // Output: John Doe`

Here, we combined firstName and lastName using the + operator and added a space between them.

You can also use the concat() method:

Description	Example
Combine strings using the concat() method.	`String anotherFullName = firstName.concat(" ").concat(lastName);`
Print the concatenated string.	`System.out.println("Another Full Name: " + anotherFullName); // Output: John Doe`

String comparison

When you want to check if two strings are the same, use the equals() method. This checks if both strings have identical content.

Description	Example
Create three strings to compare.	`String str1 = "Hello";`
Create another string to compare.	`String str2 = "Hello";`

Description	Example
Create a third string to compare.	`String str3 = "World";`
Check if str1 is equal to str2.	`boolean isEqual = str1.equals(str2);`
Print comparison result.	`System.out.println("str1 equals str2: " + isEqual); // Output: true`
Check if str1 is equal to str3.	`boolean isNotEqual = str1.equals(str3);`
Print comparison result.	`System.out.println("str1 equals str3: " + isNotEqual); // Output: false`

In this example, isEqual returns true because both strings are "Hello". However, isNotEqual returns false since "Hello" and "World" are different.

String immutability

One important thing to know about strings in Java is that they are immutable. This means that once a string is created, it cannot be changed. If you try to change it, you will actually create a new string instead.

Description	Example
Create an original string.	`String original = "Hello";`

Description	Example
Add to the string (creates a new string).	`original = original + " World";`
Print the new string.	`System.out.println(original); // Output: Hello World`

In this case, we added "World" to original, but instead of changing the original string, we created a new string that combines both parts.

Finding substrings

You may want to get a smaller part of a string. You can do this using the substring() method, which allows you to specify where to start and where to stop.

Description	Example
Create a string and extract a substring.	`String phrase = "Java Programming";`
Extract a substring from the string.	`String sub = phrase.substring(5, 16);`
Print the extracted substring.	`System.out.println("Substring: " + sub); // Output: Programming`

Description	Example

In this example, we started at index 5 and went up to (but did not include) index 16 to extract "Programming".

String methods

Java has many built-in methods for strings that help you manipulate and process them. Here are some useful ones:

toUpperCase(): This method converts all letters in a string to uppercase.

Description	Example
Create a string.	`String text = "hello";`
Convert the string to uppercase.	`System.out.println(text.toUpperCase()); // Output: HELLO`

toLowerCase(): This converts all letters in a string to lowercase.

Description	Example
Create a string.	`String text = "WORLD";`
Convert the string to lowercase.	`System.out.println(text.toLowerCase()); // Output: world`

trim(): This method removes any extra spaces at the beginning or end of a string.

Description	Example
Create a string with extra spaces and trim it.	`String textWithSpaces = "   Hello   ";`
Remove spaces from the string.	`System.out.println(textWithSpaces.trim()); // Output: Hello`

replace(): If you want to change all instances of one character or substring to another, use this method.

Description	Example
Create a sentence and replace a word.	`String sentence = "I like cats.";`
Replace a word in the sentence.	`String newSentence = sentence.replace("cats", "dogs");`
Print the new sentence.	`System.out.println(newSentence); // Output: I like dogs.`

Splitting strings

You can break a string into smaller pieces using the split() method. This is useful when you have data separated by commas or spaces.

Description	Example
Create a CSV string and split it.	`String csv = "apple,banana,cherry";`

Description	Example
Split the string at each comma.	`String[] fruits = csv.split(",");`

Print each fruit in the array.	`for (String fruit : fruits) {`
	`    System.out.println(fruit);`
	`}`

Output:	`apple`
	`banana`
	`cherry`

Joining strings

If you have an array of strings and want to combine them back into one single string, you can use the String.join() method.

Description	Example
Create an array of strings.	`String[] colors = {"Red", "Green", "Blue"};`
Join the strings with a separator.	`String joinedColors = String.join(", ", colors);`

Description	Example
Print the joined string.	`System.out.println(joinedColors); // Output: Red, Green, Blue`

Using Packages and Imports

Creating a package

To create a package, you simply declare it at the top of your Java source file using the package keyword followed by the package name. For example:

Description	Example
Declare a package at the top of the file.	`package com.example.myapp;`

In this example, com.example.myapp is the name of the package. It's common practice to use a reverse domain name as the package name to ensure uniqueness.

Description	Example

Define a class inside the package.	`public class MyClass {`
	`    // Class code here`
	`}`

Creating and using a package

Description	Example
Define a class inside the shapes package.	`package shapes;`

Description	Example

Create the Circle class with a constructor and a method.	
	```
	public class Circle {
	    private double radius;

	    public Circle(double radius) {
	        this.radius = radius;
	    }

	    public double area() {
	        return Math.PI * radius * radius;
	    }
	}
	```

To use this class in another Java file, you need to import it.

Importing classes

To import a specific class from a package, use the following syntax:

Description	Example
Import a specific class from a package.	`import package_name.ClassName;`

Importing all classes from a package

You can also import all classes from a package using an asterisk (*).

Description	Example
Import all classes from the shapes package.	`import shapes.*;`

This imports all classes in the shapes package, allowing you to use any class without needing to import them individually.

Implementing Functions and Methods

Function structure

Description	Example

The structure of a function in Java.	`returnType functionName(parameter1Type parameter1, parameter2Type parameter2) {`
	`    // code to be executed`
	`    return value; // optional`
	`}`

Example of a Simple Function

Let's create a simple function that adds two numbers:

Description	Example

Create a function that adds two numbers.	`public static int add(int a, int b) {`
	`    return a + b;`
	`}`

Call the add function in the main method and print the result.	`int sum = add(5, 3);`
	`System.out.println("The sum is: " + sum);`

Example of a simple method

Let's create a method within a class:

Description	Example
Define a multiply method inside the Calculator class.	`public class Calculator`
The multiply method takes two integers and returns their product.	`public int multiply(int x, int y) {`

Description	Example
Close the multiply method.	`}`
Define the main method, which is the program's entry point.	`public static void main(String[] args) {`
Create an instance of the Calculator class in the main method.	`Calculator calc = new Calculator();`
Call the multiply method with 4 and 5 and store the result.	
int product = calc.multiply(4, 5)	
Print the result to the screen.	`System.out.println("The product is: " + product);`

Parameters and arguments

Parameters are the inputs to functions or methods, while arguments are the values passed when calling these functions or methods.

Description	Example

Define a method with multiple parameters.	`public void greet(String name, int age) {`
	`    System.out.println("Hello " + name + ", you are " + age + " years old.");`
	`}`

Description	Example

Call the greet method with arguments in the main method.	`Greeting greeting = new Greeting();`
	`greeting.greet("Alice", 30);`

Return values

Functions and methods can return values or perform actions without returning anything.

Description	Example

Define a method that returns a value.	`public double area(double length, double width) {`
	`    return length * width;`
	`}`

Call the area method and print the returned value.	`Rectangle rect = new Rectangle();`
	`double area = rect.area(4.5, 3.0);`
	`System.out.println("The area of the rectangle is: " + area);`

Overloading methods

Java allows defining multiple methods with the same name but different parameters. This is known as method overloading.

Description	Example
Define the `Display` class.	`public class Display {`
Define an overloaded method that takes an `int`.	`    public void show(int number) {`

Description	**Example**
Print the number.	`        System.out.println("Number: " + number);`
Close the first method.	`    }`
Define an overloaded method that takes a `String`.	`    public void show(String text) {`
Print the text.	`        System.out.println("Text: " + text);`
Close the second method.	`    }`
Close the `Display` class.	`}`

Description	Example
Define the `main` method to call the overloaded methods.	`public static void main(String[] args) {`
Create a `Display` object.	`Display display = new Display();`

Call `show(int)` and `show(String)`.	`display.show(10);`
	`display.show("Hello World");`

Close the `main` method.	`}`

Scope of identifiers

The scope of an identifier refers to the part of the program where the identifier can be accessed.

Description	Example
Local Scope: Identifiers are accessible only within the method or block.	`int x = 10; // x is local to this block`
Instance Scope: Variables are accessible by all methods in the class.	`private int x; // x is accessible by all methods`

Description	Example
Static Scope: Static variables belong to the class and are accessible throughout the class.	`private static int count;`

Void methods

A void method does not return a value.

Description	Example

Define a void method that prints a message.	`public void printMessage() {`
	`    System.out.println("Hello, World!");`
	`}`

Empty parameter lists

An empty parameter list means the method does not take any parameters.

Description	Example

Define a method with an empty parameter list.	`public void show() {`
	`    System.out.println("No parameters here.");`
	`}`

An Introduction to Exceptions

Basic exception handling

Description	Example

Java provides a robust mechanism for handling exceptions using the try, catch, and finally blocks.	`public class ExceptionExample {`
	`  public static void main(String[] args) {`
	`    int numerator = 10;`
	`    int denominator = 0; // This will cause an ArithmeticException`
	`    try {`
	`      int result = numerator / denominator; // This line may throw an exception`

Description	Example
	```
	        System.out.println("Result: " + result);
	      } catch (ArithmeticException e) {
	        System.out.println("Error: Cannot divide by zero.");
	    } finally {
	        System.out.println("This block executes regardless of an exception.");
	     }
	    }
	  }
	```

Creating custom exceptions

Description	Example

Sometimes, you may want to create your own exception types. You can do this by extending the Exception class.	
	```
	class MyCustomException extends Exception {
	  public MyCustomException(String message) {
	    super(message); // Pass the message to the parent Exception class
	  }
	}
	public class CustomExceptionExample {
	   public static void main(String[] args) {
	 try {
	   throw new MyCustomException("This is a custom exception message.");
	 } catch (MyCustomException e) {
	   System.out.println(e.getMessage());
	   }
	  }
	}
	```

Using Finally Block

The structure of exception-handling

Description	Example

One important feature of Java is its exception handling mechanism, which includes the try, catch, and finally blocks.	
	```
	try {
	   // Code that may throw an exception
	} catch (ExceptionType e) {
	   // Code to handle the exception
	} finally {
	   // Code that will always execute
	}
	```

Understanding the finally block

Description	Example

The code within the finally block always runs after the try and catch blocks, regardless of whether an exception was thrown or caught. It is commonly used for resource management.	
	```
	public class FinallyExample {
	  public static void main(String[] args) {
	    try {
	      System.out.println("In try block");
	    } catch (ArithmeticException e) {
	      int result = 10 / 0; // This line will throw an exception
	   } catch (ArithmeticException e) {
	      System.out.println("Caught an exception: " + e.getMessage());
	   } finally {
	 System.out.println("Finally block executed");
	   }
	  }
	}
	```

Correct usage of the finally block

Description	Example

Always executing cleanup code: The primary purpose of the finally block is to ensure that the cleanup code runs regardless of whether an exception was thrown or not.	
	```
	public class CorrectFinallyUsage {
	  public static void main(String[] args) {
	    FileReader file = null;
	    try {
	      file = new FileReader("example.txt");
	      // Code to read from the file
	    } catch (IOException e) {
	    System.out.println("Error reading file: " + e.getMessage());
	   } finally {
	       try {
	         if (file != null) {
	           file.close();
	     System.out.println("File closed successfully.");
	         }
	     } catch (IOException e) {
	          System.out.println("Error closing file: " + e.getMessage());
	   }
	  }
	}
	```

Releasing database connections: Using the finally block to close database connections is another correct practice.	
	```
	import java.sql.Connection;
	import java.sql.DriverManager;
	import java.sql.SQLException;
	public class DatabaseConnectionCorrectUsage {
	  public static void main(String[] args) {
	     Connection connection = null;
	     try {
	       connection = DriverManager.getConnection("jdbc//localhost:3306/mydb", "user", "password");
	       // Perform database operations
	     } catch (SQLException e) {
	           System.out.println("Database error: " + e.getMessage());
	     } finally {
	   try {
	     if (connection != null) {
	          connection.close();
	```

Description	Example
	```
	                System.out.println("Database connection closed.");
	            }
	         } catch (SQLException e) {
	            System.out.println("Error closing connection: " + e.getMessage());
	         }
	        }
	      }
	```

Incorrect usage of the finally block

Description	Example

Not handling exceptions in finally: If an exception occurs in the finally block, it may suppress exceptions thrown in the try or catch blocks.	
	```
	public class IncorrectFinallyUsage {
	  public static void main(String[] args) {
	       try {
	          int result = 10 / 0; // This will throw an exception
	       } catch (ArithmeticException e) {
	          System.out.println("Caught an exception: " + e.getMessage());
	       } finally {
	         int x = 10 / 0; // This will throw another exception
	         System.out.println("Finally block executed");
	    }
	  }
	}
	```

Using return statements in finally: This can lead to unexpected behavior, as it overrides any return statements from the try or catch blocks.	
	```
	public class ReturnInFinally {
	  public static void main(String[] args) {
	      System.out.println(testMethod());
	  }
	  public static int testMethod() {
	     try {
	       return 1; // Return from try block
	     } catch (Exception e) {
	       return 2; // Return from catch block
	     } finally {
	       return 3; // This will override previous returns
	    }
	  }
	}
	```

Assuming finally will not execute: This is incorrect; the finally block will always execute unless the program crashes or is forcibly terminated.	
	```
	public class FinallyAlwaysExecutes {
	  public static void main(String[] args) {
	      try {
	         System.out.println("Trying risky operation...");
	         int result = 10 / 0; // Throws ArithmeticException
	      } catch (ArithmeticException e) {
	         System.out.println("Caught an ArithmeticException.");
	```

Description	Example
	```
	            // Not exiting or terminating the program here
	          }
	        // Assuming finally won't execute (this is wrong)
	      }
	      // The finally block should be here to ensure it executes.
	    }
	```

Using Multiple Try Catch

Basic try-catch structure

Description	Example

Multiple try-catch blocks allow developers to handle different types of exceptions in a structured way. In Java, the basic structure of a try-catch block looks like this.	
	```
	try {
	   // Code that may throw an exception
	} catch (ExceptionType e) {
	   // Code to handle the exception
	}
	```

Multiple try-catch

Description	Example

Multiple try-catch refers to the use of more than one catch block within a single try statement or multiple try-catch statements in the code.	
	```
	public class MultipleCatchExample {
	  public static void main(String[] args) {
	    int[] numbers = {1, 2, 3};
	    int index = 5; // Invalid index
	    try {
	      // Trying to access an invalid index
	    System.out.println("Number: " + numbers[index]);
	    // Trying to divide by zero
	   int result = 10 / 0;
	       } catch (ArrayIndexOutOfBoundsException e) {
	           System.out.println("Error: Index out of bounds.");
	       } catch (ArithmeticException e) {
	           System.out.println("Error: Division by zero.");
	    }
	  }
	}
	```

Throws keyword

Description	Example

The throws keyword is used in method declarations to indicate that a method can throw one or more exceptions.	
	```
	public class ThrowsExample {
	  public static void main(String[] args) {
	    try {
	      readFile("nonexistentfile.txt");
	    } catch (IOException e) {
	      System.out.println("Error: " + e.getMessage());
	    }
	  }
	  // Method that declares an exception
	  static void readFile(String fileName) throws IOException {
	    FileReader file = new FileReader(fileName);
	    BufferedReader fileInput = new BufferedReader(file);
	    // Reading the file
	    System.out.println(fileInput.readLine());
	    fileInput.close();
	  }
	}
	```

Checked and Runtime Exceptions Compared

Checked exceptions

Description	Example

Checked exceptions are exceptions checked at compile time. They usually represent recoverable conditions, such as file not found or network issues.	
	```
	import java.io.File;
	import java.io.FileNotFoundException;
	import java.util.Scanner;
	public class CheckedExceptionExample {
	   public static void main(String[] args) {
	      try {
	         File myFile = new File("nonexistentfile.txt");
	         Scanner myReader = new Scanner(myFile);
	         while (myReader.hasNextLine()) {
	            String data = myReader.nextLine();
	            System.out.println(data);
	         }
	         myReader.close();
	   } catch (FileNotFoundException e) {
	         System.out.println("An error occurred: " + e.getMessage());
	     }
	   }
	}
	```

Runtime exceptions

Description	Example

Runtime exceptions do not need to be explicitly caught or declared. They usually indicate programming errors, such as logic errors or improper use of APIs.	`public class RuntimeExceptionExample {`
	`  public static void main(String[] args) {`
	`    int numerator = 10;`
	`    int denominator = 0;`
	`    try {`
	`      int result = numerator / denominator; // This will cause an ArithmeticException`
	`      System.out.println("Result: " + result);`
	`    } catch (ArithmeticException e) {`
	`        System.out.println("An error occurred: Cannot divide by zero.");`
	`    }`
	`  }`
	`}`

Summary

In addition to viewing this resource here, you can select the printer icon located at the top of the screen to print this information or save this information to a PDF file, based on your computer's capabilities. Also, return to the course itself to access the PDF file within the course for direct downloading.

Author(s)

Ramanujam Srinivasan
Lavanya Thiruvali Sunderarajan