## Strings

### Immutability

Once a String object is created, it is not allowed to change. It cannot be made larger or smaller, and you cannot change one of the characters inside it.

You can think of a string as a storage box you have perfectly full and whose sides can’t bulge. There’s no way to add objects, nor can you replace objects without disturbing the entire arrangement. The trade-off for the optimal packing is zero fl exibility.

Mutable is another word for changeable. Immutable is the opposite—an object that can’t be changed once it’s created. On the OCA exam, you need to know that String is immutable.

Also, immutable classes in Java are fi nal, and subclasses can’t add mutable behavior.

String s1 = "1";

String s2 = s1.concat("2");

s2.concat("3");

System.out.println(s2);

Output-> 12

**String Pool** Since strings are everywhere in Java, they use up a lot of memory. In some production appli cations, they can use up 25–40 percent of the memory in the entire program. Java realizes that many strings repeat in the program and solves this issue by reusing common ones. The string pool, also known as the intern pool, is a location in the Java virtual machine (JVM) that collects all these strings.

## Ch-3 Core Java APIs

Important String Methods

length()

charAt()

indexOf()

substring()

toLowerCase() and toUpperCase()

equals() and equalsIgnoreCase()

startsWith() and endsWith()

contains()

replace()

trim()

Method Chaining

### ****Equality of Strings****

There are two types of equality checks in Java:

1. **Reference Equality** (using ==)
2. **Value Equality** (using .equals())

#### ****Example of Reference Equality (****==****):****

This checks if two string references point to the same object in memory.

java

Copy code

public class StringReferenceEquality {

public static void main(String[] args) {

String str1 = "Hello";

String str2 = "Hello";

String str3 = new String("Hello");

System.out.println(str1 == str2); // true (both point to the same object in the string pool)

System.out.println(str1 == str3); // false (str3 is a new object in heap memory)

}

}

#### ****Example of Value Equality (****.equals()****):****

This checks if two strings have the same sequence of characters.

java

Copy code

public class StringValueEquality {

public static void main(String[] args) {

String str1 = "Hello";

String str2 = "Hello";

String str3 = new String("Hello");

System.out.println(str1.equals(str2)); // true (same character sequence)

System.out.println(str1.equals(str3)); // true (same character sequence, even if str3 is a different object)

}

}

 **Immutability:**

* Operations like concatenation, replacing, or converting to uppercase do not modify the original string; they create a new string.

 **Equality:**

* Use == for checking if two string references point to the same memory location.
* Use .equals() for checking if two strings have the same content.

### Using the StringBuilder Class

StringBuilder one = new StringBuilder();

StringBuilder two = new StringBuilder();

StringBuilder three = one.append("a");

System.out.println(one == two); // false

System.out.println(one == three); // true

### Key Points:

* The append() method modifies the existing StringBuilder and returns the same instance.

### 1. ****Compile-Time Evaluation****

These are operations or values that are resolved and fixed during the compilation of the program.

#### Examples:

* **Constants (final values):**

java

Copy code

final int x = 10; // Known at compile-time

* **String Literals:**

java

Copy code

String str = "Hello World"; // Added to the string pool at compile-time

* **Compile-Time Constants:**
  + Any expression involving only constants:

java

Copy code

final int x = 5;

final int y = 10;

int z = x + y; // Evaluated at compile-time

* **Constant Expressions in Strings:**

java

Copy code

String str = "Hello" + " " + "World"; // "Hello World" resolved at compile-time

### 2. ****Runtime Evaluation****

These are operations or values that require execution of the program to determine their result.

#### Examples:

* **Operations Involving Variables:**

int x = 5;

int y = 10;

int z = x + y; // Evaluated at runtime

* **Method Calls:**

String trimmed = " Hello ".trim(); // The result of trim() is resolved at runtime

* **String Concatenation with Non-Constants:**

String dynamic = "Hello";

String result = dynamic + " World"; // Resolved at runtime

* **Values Dependent on User Input or External Factors:**

Scanner scanner = new Scanner(System.in);

int userInput = scanner.nextInt(); // Known only at runtime

### Key Differences:

|  |  |
| --- | --- |
| **Compile-Time** | **Runtime** |
| Resolved during compilation. | Determined during program execution. |
| Must involve constants or literals. | Can involve variables, user input, or method calls. |
| Errors (like syntax or type mismatch) detected here. | Errors (like NullPointerException) occur during execution. |
| Efficient, as the result is precomputed. | Slight overhead due to dynamic evaluation. |

### How to Determine:

1. **Look for Method Calls:**
   * If an operation involves a method (e.g., .trim(), .toUpperCase()), it will be **runtime**.
2. **Check for Variables or External Inputs:**
   * If the result depends on variables or inputs that are not constants, it's **runtime**.
3. **Examine Constants and Literals:**
   * Operations involving only final constants or literals are **compile-time**.
4. **Use Reflection or Debugging Tools:**
   * Advanced debugging or reflection can confirm runtime operations.

### Practical Example:

String a = "Hello" + "World"; // Compile-time

String b = new String("Hello") + "World"; // Runtime

String c = " Hello ".trim(); // Runtime

String d = "Hello".substring(0, 3); // Runtime

### Summary:

* If the value/result can be determined **without running the program**, it's **compile-time**.
* If the value/result depends on program execution or user interaction, it's **runtime**.

**Creating a Multidimensional Array**

Multiple array separators are all it takes to declare arrays with multiple dimensions. You can locate them with the type or variable name in the declaration, just as before:

int[][] vars1; // 2D array

int vars2 [][]; // 2D array

int[] vars3[]; // 2D array

int[] vars4 [], space [][]; // a 2D AND a 3D array

String [][] rectangle = new String[3][2];

**ASYMMETRIC ARRAY**

Another way to create an asymmetric array is to initialize just an array’s first dimension, and define the size of each array component in a separate statement:

int [][] args = new int[4][];

args[0] = new int[5];

args[1] = new int[3];

This technique reveals what you really get with Java: arrays of arrays that, properly managed, offer a multidimensional effect.

**Comparison with Symmetric Arrays:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Symmetric Array** | **Asymmetric Array** |
| **Structure** | All rows have the same length. | Rows can have different lengths. |
| **Example** | int[3][3] (3x3 matrix) | { {1, 2}, {3}, {4, 5, 6} } |
| **Use Case** | Useful for fixed grid-like data. | Useful when row sizes vary (e.g., sparse data). |
| **Memory Usage** | Fixed memory allocation for all rows. | Variable memory allocation based on row size. |

**package** Array.Array2D;

**public** **class** AsymmetricArray {

**public** **static** **void** main(String[] args) {

**int** [][] numbers = **new** **int**[4][];

numbers[0] = **new** **int**[5];

numbers[1] = **new** **int**[3];

System.***out***.println(numbers.length);

}

}

### Key Points:

1. The length of a multidimensional array refers only to the number of rows (outer array size), not the lengths of the inner arrays.
2. Even though numbers[2] and numbers[3] are not initialized (remain null), their presence as part of the outer array contributes to the total length.
3. The numbers.length refers to the length of the outer array, which is 4 because it was defined as new int[4][]. This is independent of the lengths of the inner arrays.

ARRAYLIST

Arraylist methods

add() - The add() methods insert a new value in the ArrayList. The method signatures are as follows:

* boolean add(E element)
* void add(int index, E element)

remove() - The remove() methods remove the first matching value in the ArrayList or remove the element at a specifi ed index. The method signatures are as follows:

* boolean remove(Object object)
* E remove(int index)

set() ­-The set() method changes one of the elements of the ArrayList without changing the size. The method signature is as follows:

* E set(int index, E newElement)

isEmpty() and size()-The isEmpty() and size() methods look at how many of the slots are in use. The method signatures are as follows:

* boolean isEmpty()
* int size()

clear()-The clear() method provides an easy way to discard all elements of the ArrayList. The method signature is as follows:

* void clear()

contains()-The contains() method checks whether a certain value is in the ArrayList. The method signature is as follows:

* boolean contains(Object object)

**WRAPPER CLASSES**

|  |  |  |
| --- | --- | --- |
| **Primitive Type** | **Wrapper Class** | **Example of Constructing** |
| boolean | Boolean | new Boolean(true) |
| byte | Byte | new Byte((byte) 1) |
| short | Short | new Short((short) 1) |
| int | Integer | new Integer(1) |
| long | Long | new Long(1) |
| float | Float | new Float(1.0) |
| double | Double | new Double(1.0) |
| char | Character | new Character('c') |

valueOf method takes only String

|  |  |  |
| --- | --- | --- |
| **Wrapper Class** | **Converting String to Primitive** | **Converting String to Wrapper Class** |
| **Boolean** | Boolean.parseBoolean("true"); | Boolean.valueOf("TRUE"); |
| **Byte** | Byte.parseByte("1"); | Byte.valueOf("2"); |
| **Short** | Short.parseShort("1"); | Short.valueOf("2"); |
| **Integer** | Integer.parseInt("1"); | Integer.valueOf("2"); |
| **Long** | Long.parseLong("1"); | Long.valueOf("2"); |
| **Float** | Float.parseFloat("1"); | Float.valueOf("2.2"); |
| **Double** | Double.parseDouble("1"); | Double.valueOf("2.2"); |
| **Character** | **None** | **None** |

**Using getClass().getSimpleName()**

The getClass().getSimpleName() method returns the simple name of the class of the object.

**Convert Array to ArrayList**

To convert an **array** to an **ArrayList** in Java, you can use the Arrays.asList() method from the java.util.Arrays utility class. Here's how you can do it:

### ****1. Using**** Arrays.asList()

This method converts the array into a fixed-size List. You can then pass the List to an ArrayList if you need a resizable list.

#### ****Example:****

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

public class ArrayToArrayList {

public static void main(String[] args) {

String[] array = {"hawk", "robin", "sparrow"};

// Convert array to fixed-size List

List<String> fixedList = Arrays.asList(array);

// Convert fixed-size List to ArrayList

ArrayList<String> arrayList = new ArrayList<>(fixedList);

System.out.println("ArrayList: " + arrayList);

}

}

**Output:**

ArrayList: [hawk, robin, sparrow]

**Notes:**

* Arrays.asList() returns a fixed-size list, so you cannot add or remove elements from it.
* To modify the list (e.g., add or remove elements), wrap it in an ArrayList as shown above.

### ****2. Using**** Collections.addAll()

You can use Collections.addAll() to directly add elements from the array into an ArrayList.

#### ****Example:****

import java.util.ArrayList;

import java.util.Collections;

public class ArrayToArrayList {

public static void main(String[] args) {

String[] array = {"hawk", "robin", "sparrow"};

// Convert array to ArrayList

ArrayList<String> arrayList = new ArrayList<>();

Collections.addAll(arrayList, array);

System.out.println("ArrayList: " + arrayList);

}

}

**Output:**

ArrayList: [hawk, robin, sparrow]

### ****3. Using a Loop****

For greater control or when working with primitive types, you can manually loop through the array and add elements to an ArrayList.

#### ****Example:****

import java.util.ArrayList;

public class ArrayToArrayList {

public static void main(String[] args) {

String[] array = {"hawk", "robin", "sparrow"};

// Convert array to ArrayList using a loop

ArrayList<String> arrayList = new ArrayList<>();

for (String element : array) {

arrayList.add(element);

}

System.out.println("ArrayList: " + arrayList);

}

}

**Output:**

ArrayList: [hawk, robin, sparrow]

### ****Converting Primitive Arrays to ArrayList****

If you have a **primitive array** (e.g., int[]), you need to first convert it to a wrapper type (e.g., Integer[]) using Java Streams or a loop.

#### ****Using Streams (Java 8+)****

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

public class PrimitiveArrayToArrayList {

public static void main(String[] args) {

int[] numbers = {1, 2, 3};

// Convert primitive array to wrapper array

List<Integer> arrayList = new ArrayList<>();

for (int number : numbers) {

arrayList.add(number);

}

System.out.println("ArrayList: " + arrayList);

}

}

### Summary Table:

|  |  |  |
| --- | --- | --- |
| **Method** | **Resizability** | **Additional Notes** |
| Arrays.asList() | Fixed size | Modifications like add/remove are not allowed. |
| Collections.addAll() | Resizable | Directly adds elements to an ArrayList. |
| **Loop** | Resizable | Useful for custom transformations or primitive arrays. |

**WORKING WITH DATE AND TIMES**

import java.time.\*; // import time classes

**LocalDate** - Contains just a date—no time and no time zone. A good example of LocalDate is your birthday this year. It is your birthday for a full day regardless of what time it is.

**LocalTime** - Contains just a time—no date and no time zone. A good example of LocalTime is midnight. It is midnight at the same time every day.

**LocalDateTime** - Contains both a date and time but no time zone. A good example of LocalDateTime is “the stroke of midnight on New Year’s.” Midnight on January 2 isn’t nearly as special, and clearly an hour after midnight isn’t as special either.

System.out.println(LocalDate.now());

System.out.println(LocalTime.now());

System.out.println(LocalDateTime.now());

**OUTPUT**

2015-01-20

12:45:18.401

2015-01-20T12:45:18.401

why did this not require new keyword while creating a LocalTime instance

The reason you don't need the new keyword when creating an instance of LocalDate, LocalTime, or LocalDateTime in the code you provided is because **these classes are part of the Java java.time package and are immutable objects**. Instead of using constructors, they use static factory methods to create instances.

### ****Why No**** new ****Keyword?****

1. **Immutable Classes:** Classes like LocalDate, LocalTime, and LocalDateTime are immutable. Instead of allowing direct instantiation with constructors, the Java designers provided static factory methods (like of() and now()) to create new instances of these classes. This design promotes thread safety and ensures immutability.
2. **Factory Methods Over Constructors:** Static factory methods are often preferred for creating instances in immutable classes because they:
   * Provide more meaningful names (e.g., of, now, from) compared to constructors.
   * Can return precomputed instances or cached values for optimization (e.g., avoiding duplicate instances of the same data).
   * Allow the implementation to change internally without affecting external usage.

### ****How**** LocalDate****,**** LocalTime****, and**** LocalDateTime ****Work:****

* **Static Methods for Object Creation:**
  + LocalDate.of(int year, int month, int dayOfMonth)
  + LocalTime.of(int hour, int minute, int second, int nanoOfSecond)
  + LocalDateTime.of(LocalDate date, LocalTime time)

These methods internally handle all the complexities of object creation and validation.

* **No Public Constructors:** If you inspect these classes, you'll notice that their constructors are private. This prevents direct instantiation using the new keyword.

For example, here's the LocalDate constructor:

java

Copy code

private LocalDate(int year, int month, int dayOfMonth) {

// Constructor logic

}

Because it’s private, it can only be accessed by methods within the same class, such as the of() method.

### ****Example Breakdown:****

Here’s how your code works:

LocalDate date = LocalDate.of(1997, 10, 27);

* Calls the of() method in the LocalDate class.
* Creates and returns an immutable LocalDate object representing 1997-10-27.

LocalTime time = LocalTime.of(10, 23, 23, 0);

* Calls the of() method in the LocalTime class.
* Creates and returns an immutable LocalTime object representing 10:23:23.

LocalDateTime dateTime1 = LocalDateTime.of(date, time);

* Combines the LocalDate and LocalTime objects into a new LocalDateTime instance.

### ****Advantages of Factory Methods:****

1. **Readability:** The of() method makes the code more readable and self-explanatory.
2. **Validation:** Factory methods can include additional logic to validate input values.
3. **Flexibility:** They can return cached or optimized instances when possible.

### ****Conclusion:****

The java.time API classes use static factory methods instead of public constructors for creating instances, promoting immutability, thread safety, and a clean design. This is why the new keyword is not required for these classes.

4o

##### You said:

give some example of Immutable classes like LocalDate which do not require new keyword

2/2

###### ChatGPT said:

ChatGPT

Here are some examples of **immutable classes** in Java, like LocalDate, that do not require the new keyword for instantiation because they use **static factory methods** for object creation:

### ****1.**** String

* Although String is not created via factory methods, it’s an immutable class where you don’t necessarily need the new keyword because of string literals.

java

Copy code

String str = "Hello"; // No `new` keyword

### ****2.**** java.math.BigInteger

* BigInteger is immutable and provides factory methods for object creation.

BigInteger bigInt = BigInteger.valueOf(12345); // Using a static factory method

BigInteger bigInt2 = BigInteger.ONE; // Predefined constants

### ****3.**** java.math.BigDecimal

* Like BigInteger, BigDecimal is immutable and uses factory methods.

BigDecimal bigDec = BigDecimal.valueOf(123.45); // Using a static factory method

BigDecimal bigDec2 = BigDecimal.ZERO; // Predefined constant

### ****4.**** java.util.Optional

* Optional is immutable and provides factory methods like of, ofNullable, and empty.

Optional<String> optional = Optional.of("Hello");

Optional<String> optionalEmpty = Optional.empty();

### ****5.**** java.time ****Classes****

All the classes in the java.time package are immutable and use static factory methods:

* LocalDate
* LocalTime
* LocalDateTime
* ZonedDateTime

Examples:

LocalDate date = LocalDate.of(2024, 12, 11);

LocalTime time = LocalTime.now();

LocalDateTime dateTime = LocalDateTime.of(date, time);

### ****6.**** java.util.Collections

The Collections utility class provides methods to create immutable collections.

List<String> immutableList = Collections.unmodifiableList(Arrays.asList("A", "B", "C"));

Set<String> immutableSet = Set.of("A", "B", "C"); // Java 9+

Map<String, String> immutableMap = Map.of("key1", "value1", "key2", "value2"); // Java 9+

### ****7.**** java.nio.file.Path

The Path class is immutable and uses factory methods from Paths or FileSystems.

Path path = Paths.get("example.txt");

Path absolutePath = Path.of("/home/user/documents");

### ****8.**** java.util.UUID

The UUID class represents immutable universally unique identifiers.

UUID uuid = UUID.randomUUID(); // Generate a random UUID

UUID uuidFromString = UUID.fromString("123e4567-e89b-12d3-a456-426614174000");

### ****9.**** Currency ****(from**** java.util****)****

The Currency class represents currency codes (e.g., USD, INR) and is immutable.

Currency currency = Currency.getInstance("USD");

### ****10.**** java.awt.Color

Color is immutable and has static factory methods for creating standard colors.

Color color = Color.RED; // Predefined constants

Color customColor = new Color(255, 0, 0); // Uses a constructor, but it's still immutable

### ****11.**** java.util.Locale

The Locale class is immutable and uses factory methods like forLanguageTag.

Locale locale = Locale.forLanguageTag("en-US");

Locale defaultLocale = Locale.getDefault();

### ****12.**** java.net.URI

The URI class represents immutable URIs and has factory methods for object creation.

URI uri = URI.create("https://example.com");

### ****13.**** Pattern ****(from**** java.util.regex****)****

The Pattern class is immutable and uses the static compile() method.

Pattern pattern = Pattern.compile("[a-z]+");

### ****Why Immutable Classes Avoid**** new ****Keyword?****

1. **Thread Safety:** Immutability ensures objects can be shared between threads without synchronization.
2. **Optimization:** Factory methods can cache frequently used objects.
3. **Readability:** Static methods like of, from, or valueOf are more descriptive than constructors.
4. **Validation:** Factory methods allow validating input before creating an object.

These immutable classes follow similar principles, making them efficient and thread-safe while simplifying object creation.

4o

##### You said:

**but collections require new keyword, so why you have mentioned above**

###### ChatGPT said:

ChatGPT

You are absolutely right—**collections like ArrayList, HashSet, and HashMap require the new keyword** when creating mutable instances. However, the examples I provided in the context of **immutable collections** specifically use factory methods or utility methods, such as Collections.unmodifiableList or Set.of, which do not require the new keyword.

Let me clarify this:

### ****Mutable Collections (Require**** new ****Keyword)****

If you are creating a mutable collection like ArrayList, HashSet, or HashMap, you need to use the new keyword:

java

Copy code

List<String> mutableList = new ArrayList<>();

Set<String> mutableSet = new HashSet<>();

Map<String, String> mutableMap = new HashMap<>();

### ****Immutable Collections (Do Not Require**** new ****Keyword)****

If you want to create immutable collections (which are unmodifiable and thread-safe), you can use **static factory methods** introduced in Java or utility methods from Collections. These do **not** require the new keyword:

#### Examples:

1. **Using Collections.unmodifiableList (Utility Method):**

java

Copy code

List<String> immutableList = Collections.unmodifiableList(Arrays.asList("A", "B", "C"));

1. **Using Java 9+ Factory Methods (List.of, Set.of, Map.of):**

java

Copy code

List<String> immutableList = List.of("A", "B", "C"); // Immutable

Set<String> immutableSet = Set.of("A", "B", "C"); // Immutable

Map<String, String> immutableMap = Map.of("key1", "value1", "key2", "value2");

### Why Include Collections?

I included immutable collections in the list because they do not require the new keyword **when using their factory methods**. They behave similarly to other immutable classes like LocalDate or BigInteger in this context.

If your focus is on mutable collections (like ArrayList), then yes, you would always require the new keyword. But for immutable collections, the approach aligns with the broader examples of factory methods.

Would you like to dive deeper into how immutable collections differ from mutable ones, or how these factory methods work internally? Let me know! 😊

4o

##### You said:

what is the usecase of Using Collections.unmodifiableList (

###### ChatGPT said:

ChatGPT

### Use Case of Collections.unmodifiableList()

The Collections.unmodifiableList() method is used to create an **immutable view** of a given list. This means that any attempt to modify the list (e.g., adding, removing, or updating elements) will result in an UnsupportedOperationException.

Here's a detailed explanation of its **use cases**:

### ****1. Prevent Accidental Modification****

If you have a list that should remain **read-only** after being initialized or passed to another method, using Collections.unmodifiableList() ensures that no one can modify its contents accidentally.

#### Example:

java

Copy code

import java.util.\*;

public class Example {

public static void main(String[] args) {

List<String> originalList = new ArrayList<>();

originalList.add("Apple");

originalList.add("Banana");

List<String> immutableList = Collections.unmodifiableList(originalList);

System.out.println("Immutable List: " + immutableList);

// Attempting to modify will throw UnsupportedOperationException

immutableList.add("Orange"); // Throws exception

}

}

### ****2. Pass a List to Untrusted Code****

When passing a list to third-party libraries or collaborators, wrapping it with Collections.unmodifiableList() ensures that the underlying list remains safe from modification.

#### Example:

java

Copy code

public void processList(List<String> list) {

// Wrap the list to make it unmodifiable before passing it

List<String> immutableList = Collections.unmodifiableList(list);

// Now the library cannot modify the original list

libraryMethod(immutableList);

}

### ****3. Maintain Thread Safety****

If multiple threads share the same list, making it immutable ensures that no thread can modify it, avoiding synchronization issues. However, it’s important to note that Collections.unmodifiableList() itself **does not make the underlying list thread-safe**, so any changes to the original list will still reflect in the unmodifiable view.

#### Example:

java

Copy code

List<String> threadSafeList = Collections.unmodifiableList(Arrays.asList("A", "B", "C"));

// Safe to share with multiple threads because no one can modify it

new Thread(() -> System.out.println(threadSafeList)).start();

### ****4. Create Immutable Constants****

In cases where you want to define a constant list in your program, you can use Collections.unmodifiableList() to ensure that the constant remains immutable.

#### Example:

java

Copy code

import java.util.Collections;

import java.util.List;

public class Constants {

public static final List<String> COLORS = Collections.unmodifiableList(

Arrays.asList("Red", "Green", "Blue")

);

}

### ****5. Avoid Defensive Copying****

Without immutability, you often need to create a **defensive copy** to protect the original list when returning it from a method. Using Collections.unmodifiableList() avoids this need by making the list immutable directly.

#### Example:

java

Copy code

private List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));

public List<String> getList() {

// Instead of returning a new copy every time:

// return new ArrayList<>(list);

// Return an unmodifiable view

return Collections.unmodifiableList(list);

}

### Limitations:

1. **Underlying List Can Still Change**:
   * If the original list is modified, those changes will still reflect in the unmodifiable view. To ensure true immutability, use factory methods like List.of (Java 9+) or copy the list first.
2. **Not Thread-Safe by Itself**:
   * The unmodifiable list does not make the underlying list thread-safe. External synchronization or immutable implementations like CopyOnWriteArrayList may be required.

Would you like more examples or comparisons with other immutability mechanisms? 😊

4o

##### You said:

what are factory methods

###### ChatGPT said:

ChatGPT

### Factory Methods in Java

A **factory method** is a design pattern used to create objects. Unlike constructors, factory methods provide greater flexibility and control over how objects are created. These methods are typically **static methods** that return an instance of a class.

### ****Features of Factory Methods****

1. **Encapsulation**:
   * Factory methods encapsulate the creation logic, which allows greater flexibility in object creation.
2. **Simplifies Object Creation**:
   * For complex objects or objects requiring additional setup, factory methods simplify the process for the caller.
3. **Custom Behavior**:
   * They can include additional logic, such as caching, validation, or using subclasses based on conditions.
4. **Improved Readability**:
   * Descriptive method names make the code easier to understand (e.g., List.of() or Collections.unmodifiableList()).

### ****Examples of Factory Methods****

#### ****1. Built-in Factory Methods in Java****

Java provides many factory methods for common tasks. Examples include:

* **Immutable Collection Creation (Java 9+)**:

List<String> list = List.of("A", "B", "C"); // Creates an immutable list

Set<Integer> set = Set.of(1, 2, 3); // Creates an immutable set

Map<String, Integer> map = Map.of("A", 1, "B", 2); // Immutable map

* **Collections Utility**:

List<String> unmodifiableList = Collections.unmodifiableList(new ArrayList<>());

* **Optional**:

Optional<String> optional = Optional.of("Value");

#### ****2. Custom Factory Methods****

Developers often use factory methods in custom code for flexibility and abstraction.

##### Example 1: Simple Factory Method

class ShapeFactory {

public static Shape createCircle(double radius) {

return new Circle(radius);

}

public static Shape createRectangle(double length, double width) {

return new Rectangle(length, width);

}

}

Usage:

Shape circle = ShapeFactory.createCircle(5.0);

Shape rectangle = ShapeFactory.createRectangle(4.0, 6.0);

##### Example 2: Factory Method with Conditional Logic

class DatabaseFactory {

public static Database getDatabase(String dbType) {

if ("MySQL".equalsIgnoreCase(dbType)) {

return new MySQLDatabase();

} else if ("PostgreSQL".equalsIgnoreCase(dbType)) {

return new PostgreSQLDatabase();

} else {

throw new IllegalArgumentException("Unknown database type");

}

}

}

Usage:

Database db = DatabaseFactory.getDatabase("MySQL");

db.connect();

#### ****3. Singleton with Factory Method****

Factory methods are often used to implement the Singleton pattern.

class Singleton {

private static Singleton instance;

private Singleton() {} // Private constructor

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

Usage:

Singleton s1 = Singleton.getInstance();

Singleton s2 = Singleton.getInstance();

System.out.println(s1 == s2); // true

### ****Advantages of Factory Methods****

1. **Decouples Object Creation**:
   * The caller doesn’t need to know the exact implementation details or constructors of the object.
2. **Encapsulation**:
   * The creation logic is hidden, allowing for better maintainability.
3. **Custom Initialization**:
   * Factory methods allow initialization logic to be customized without exposing it to the caller.
4. **Code Reusability**:
   * Shared creation logic can be reused across the application.

### ****Factory Methods vs Constructors****

|  |  |  |
| --- | --- | --- |
| **Feature** | **Factory Methods** | **Constructors** |
| **Flexibility** | Can return objects of any subclass or cached instances. | Always creates a new instance. |
| **Naming** | Can have descriptive names (e.g., createCircle()). | Always matches the class name. |
| **Return Type** | Can return any type (even interfaces or abstract types). | Always returns the exact class type. |
| **Logic** | Can contain complex logic or reuse existing instances. | Limited to basic initialization logic. |

### ****Summary****

Factory methods are powerful tools for object creation, offering more flexibility than constructors. Java itself provides several built-in factory methods (e.g., List.of(), Optional.of(), Collections.unmodifiableList()), and custom factory methods can be tailored to simplify and encapsulate object creation in your application.

4o

**DATE TIME EXCEPTION**

LocalDate.of(2015, Month.JANUARY, 32) // throws DateTimeException

You don’t need to know the exact exception that’s thrown, but it’s a clear one:

Date Manipulation

Here’s the formatted content in a clear tabular layout:

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Can call on LocalDate?** | **Can call on LocalTime?** | **Can call on LocalDateTime?** |
| plusYears / minusYears | Yes | No | Yes |
| plusMonths / minusMonths | Yes | No | Yes |
| plusWeeks / minusWeeks | Yes | No | Yes |
| plusDays / minusDays | Yes | No | Yes |
| plusHours / minusHours | No | Yes | Yes |
| plusMinutes / minusMinutes | No | Yes | Yes |
| plusSeconds / minusSeconds | No | Yes | Yes |
| plusNanos / minusNanos | No | Yes | Yes |

This table effectively separates the methods and shows which ones can be called on LocalDate, LocalTime, or LocalDateTime.

Working with Periods