Optional<C>(*from java.util*)

```
public final class java.util.Optional<T> {
  public static <T> java.util.Optional<T> empty();
  public static <T> java.util.Optional<T> of(T);
  public static <T> java.util.Optional<T> ofNullable(T);
  public T get();
  public boolean isPresent();
  public boolean isEmpty();
  public void ifPresent(java.util.function.Consumer<? super T>);
  public void ifPresentOrElse(java.util.function.Consumer<? super T>, java.lang.Runnable);
  public java.util.Optional<T> filter(java.util.function.Predicate<? super T>);
  public <U> java.util.Optional<U> map(java.util.function.Function<? super T, ? extends U>);
  public <U> java.util.Optional<U> flatMap(java.util.function.Function<? super T, ? extends java.util.Optional<? extends U>>);
  public java.util.Optional<T> or(java.util.function.Supplier<? extends java.util.Optional<? extends T>>);
  public java.util.stream.Stream<T> stream();
  public T orElse(T);
  public T orElseGet(java.util.function.Supplier<? extends T>);
  public T orElseThrow();
  public <X extends java.lang.Throwable> T orElseThrow(java.util.function.Supplier<? extends X>) throws X;
  public boolean equals(java.lang.Object);
  public int hashCode();
  public java.lang.String toString();
  static {};
}
```

✅ What is Optional in Java?

Optional is a **container object** introduced in Java 8, which may or may not contain a **non-null value.**

It is used to **avoid NullPointerException** in your code by **clearly specifying that a value may be absent.**

It's a **container** that can hold either:

- **a value of type T**, or
- **nothing (empty)**

It was introduced in **Java 8** to help reduce the risk of NullPointerException.

ASHISH

📦 How to create Optional objects

1. Optional.of(value) – when you are sure the value is not null

```
Optional<String> name = Optional.of("Ashi");
```

2. Optional.ofNullable(value) – when value may be null

```
Optional<String> name = Optional.ofNullable(null); //allowed
```

3. Optional.empty() – create an empty Optional

```
Optional<String> name = Optional.empty();
```

Methods :

✅ Optional<T> Methods Summary Table (Detailed)

Method	Return Type	Description	When to Use	Example
empty()	Optional<T>	Returns an empty Optional	To represent absence of value	Optional<String> opt = Optional.empty();
of(T value)	Optional<T>	Wraps a non-null value in an Optional	When value is guaranteed not null	Optional.of("Ashi")

ASH_23

Method	Return Type	Description	When to Use	Example
ofNullable(T value)	Optional<T>	Wraps a value that may be null	When value might be null	Optional.ofNullable(null)
get()	T	Returns the value if present, else throws NoSuchElementException	Only when you're 100% sure value is present	opt.get()
isPresent()	boolean	Returns true if value is present	To check for value before calling get() (Old style)	if(opt.isPresent()) {...}
isEmpty()	boolean	Returns true if value is not present	Opposite of isPresent()	if(opt.isEmpty()) {...}
ifPresent(Consumer<T>)	void	Executes a block if value exists	Preferred over isPresent() + get()	opt.ifPresent(val -> System.out.println(val))
ifPresentOrElse(Consumer<T>, Runnable)	void	Executes one action if value exists, another if not	Useful for clean conditional branching	opt.ifPresentOrElse(val -> ..., () -> ...);

Method	Return Type	Description	When to Use	Example
filter(Predicate<T>)	Optional<T>	Keeps value only if it passes condition	For validating contained value	opt.filter(val -> val.length() > 5)
map(Function<T, U>)	Optional<U>	Transforms the value (if present) to another type	Basic transformation (e.g., String → length)	opt.map(String::length)
flatMap(Function<T, Optional<U>>)	Optional<U>	Like map(), but avoids nested Optionals	For chaining methods that return Optional	opt.flatMap(this::getDetails)
or(Supplier<Optional<T>>)	Optional<T>	If empty, returns alternate Optional from supplier	For fallback logic	opt.or(() -> Optional.of("default"))
orElse(T other)	T	Returns value or provided default (always evaluates default)	When default is cheap and simple	opt.orElse("default")
orElseGet(Supplier<T>)	T	Returns value or default	When default is	opt.orElseGet(() -> getFromDB())

Method	Return Type	Description	When to Use	Example
		computed lazily	expensive or needs logic	
orElseThrow()	T	Throws NoSuchElementException if empty	For mandatory values	opt.orElseThrow()
orElseThrow(Supplier<Throwable>)	T	Throws custom exception if empty	Best for business rules	opt.orElseThrow(() -> new Exception())
stream() (Java 9+)	Stream<T>	Returns a stream of 0 or 1 element	Useful in stream pipelines	opt.stream().map(...).forEach(...)
equals(Object o)	boolean	Compares Optionals by value	Used internally and in test cases	opt1.equals(opt2)
hashCode()	int	Returns hash of value if present	For collections/maps	map.put(opt, value)
toString()	String	String representation: Optional[value] or Optional.empty	Debugging/logging	System.out.println(opt)

Note :

Feature	**orElse()**	**orElseGet()**
When default is called	Always, even if not needed	Only when value is absent
Type of argument	Direct value	Supplier (lambda or method reference)
Performance	May waste resources if default is complex	More efficient for expensive operations

Method	Use Case in Spring Boot
orElse()	Return a quick default object (e.g., static guest user)
orElseGet()	Return a computed/expensive fallback (e.g., default config loaded from DB/file)
orElseThrow()	Throw 404/exception when not found (common in REST controllers & service layer)

ASH

Note :

⚠️ **What You Should Avoid**

❌ Anti-pattern	✅ Instead Use
Optional.get() without check	orElse(), orElseGet(), orElseThrow()
Optional<T> as method parameter	Use nullable param and wrap inside
Optional in entity fields (JPA)	Use regular nullable fields

✅ **Can Optional store single or multiple values?**

❌ No, Optional<T> can store only a single value or nothing (empty).

It is not a collection like List or Set.

🔍 **Think of it like this:**

```
Optional<String> opt = Optional.of("Ashi"); // ✅ one value


Optional<String> empty = Optional.empty();  // ✅ no value
```

You can't do:

```
Optional<List<String>> opt = Optional.of(List.of("Ashi", "Anu"));
// ✅ This is a single List object
```

But remember:

ASHISH

- **The Optional holds one reference — and that reference *could* point to a collection, but Optional itself does not support storing multiple values.**

💡 **Summary Table**

Feature	Supports
One value	✅ Yes (if present)
No value	✅ Yes (empty)
Multiple values	❌ No
Can wrap a List<T>	✅ Yes, but still one object (the list)

✅ **Real-Time Example**

```
// Optional containing a single String
Optional<String> name = Optional.of("Ashi");

// Optional containing a List (still ONE value inside — the list)
Optional<List<String>> names = Optional.of(List.of("Ashi", "Anu", "Alex"));

names.ifPresent(list -> list.forEach(System.out::println));
```

Note : Optional<T> is a container for a single object of type T — it cannot hold multiple separate values.
If you want multiple values, use a collection like List<T> or Set<T> — and you can wrap that collection in an Optional if needed.

ASHISH

Use or() when:

- You want a fallback Optional
- You don’t want to use orElse() or orElseGet() that unwrap values directly
- You want to continue chaining with Optionals

```
Optional<Employee> fallback = emp.or(() -> Optional.of(new Employee()));

System.out.println("Using or(): " + fallback.get());
```

When to use ?

Method	Use When	Returns	Fallback Type
orElse(T)	You want a default value if the Optional is empty (default is always created)	T (value)	Direct value
orElseGet(Supplier<T>)	You want a default value, but only compute it if needed (lazy)	T (value)	Lambda expression
ifPresent(Consumer<T>)	You want to perform an	void	Consumer

Method	Use When	Returns	Fallback Type
	action *only* if the value is present		
ifPresentOrElse(Consumer<T>, Runnable)	You want to perform one action if present, and another if not	void	Consumer + Runnable

Situation	Use This Method
You want to get the value or return a fallback directly	orElse()
You want to get value or compute fallback only if needed	orElseGet()
You want to do something only when value is present	ifPresent()
You want to do one thing if present, another if not	ifPresentOrElse()

Note :

Don't Use	Why Not
orElse() with expensive default	It will create the object even if it's not needed (performance hit)
get()	Will throw NoSuchElementException if empty

❗ **orElse() is eager**

It evaluates its argument immediately, regardless of whether the Optional contains a value.

Even if the Optional **is not empty**, the object passed to orElse() **will be created.**

✅ **orElseGet() is lazy**

It means the fallback **code is not executed** unless the Optional is **empty.**

✅ **orElseThrow() is also LAZY** ✅

It behaves similarly to orElseGet() — the **exception is only created and thrown if the Optional is empty.**

1. ✅ orElseThrow() – *no arguments (Java 10+)*

```
Optional<String> opt = Optional.of("Ashi");
String value = opt.orElseThrow(); // ☑ Lazy – throws only if empty
```

- Uses NoSuchElementException internally.
- Does **not throw** or create the exception if value is present.

ASHISH_23

🔒 Hidden Secrets & Pro Tips for Optional<T>

✅ 1. Never Use Optional as a Field in Entities

- ❌ @Entity fields like Optional<String> email; are not supported by JPA.
- ✅ Use String email; instead, and wrap it in Optional at **service/controller layer.**

✅ 2. Avoid optional.get() unless you're 100% sure

- optional.get() will **throw NoSuchElementException** if empty.
- ✅ Always prefer:

```
optional.orElse("default");

optional.orElseGet(() -> compute());

optional.orElseThrow(() -> new RuntimeException("Not found"));
```

✅ 3. orElse() is eager — avoid with expensive calls

```
// BAD — compute() runs even if optional has value
optional.orElse(compute());


// GOOD — compute() runs only if needed
optional.orElseGet(() -> compute());
```

✅ Use orElseGet() for **lazy evaluation.**

ASHISH_23

✅ **4. Prefer flatMap() when method returns Optional**

```
// BAD – nested optional: Optional<Optional<Address>>
optional.map(User::getAddress);

// GOOD – flat: Optional<Address>
optional.flatMap(User::getAddress);
```

✅ **5. Avoid chaining Optional when logic becomes messy**

```
// Don't abuse optional like a logic controller
optional.map(...).filter(...).flatMap(...).orElse(...)
```

✅ If you go beyond 2–3 chains – it's better to use regular if-else or service methods.

✅ **6. Optional is NOT meant for collections**

```
// ❌ Don't do this
Optional<List<String>> names;
```

Use empty collections instead: List<String> names = new ArrayList<>();

✅ **7. Optional.stream() is powerful in Java 9+**

```
List<String> names = List.of(Optional.of("Ashi"), Optional.empty())
    .stream()
```

ASHISH 23

```
    .flatMap(Optional::stream) // ✅ removes empty
    .collect(Collectors.toList());
```

- ✅ Very clean way to eliminate Optional.empty() values.

✅ 8. Optional is NOT a replacement for null everywhere

It's best used as:

- Return type for method that **may or may not return a value**
- Temporary wrapper for handling logic safely

❌ Not for:

- Class fields
- Method parameters
- Serialization fields

✅ 9. Use Optional for expressive APIs

```
public Optional<User> findUserByEmail(String email);
```

✅ Clear intention: may return user, or may return nothing.

Compare that to:

```
public User findUserByEmail(String email); // what if email not found? Null? Exception?
```

✅ 10. Combining filter() and map() is powerful

```
String name = Optional.of("ashi")
    .filter(n -> n.length() > 3)
```

ASHISH_23

```
    .map(String::toUpperCase)
    .orElse("UNKNOWN");

System.out.println(name); // ASHI
```

🧠 **Bonus: Optional inside Optional (Double wrap issue)**

```
Optional<Optional<String>> doubleOpt =
Optional.of(Optional.of("test"));
```

⚠️ *Bad practice — avoid double-wrapping by using flatMap.*

ASHISH_23