Module java.base
Package java.util

Class Optional<T>

java.lang.Object
    java.util.Optional<T>

Type Parameters:

T - the type of value

```
public final class Optional<T>
extends Object
```

A container object which may or may not contain a non-`null` value. If a value is present, `isPresent()` returns `true`. If no value is present, the object is considered *empty* and `isPresent()` returns `false`.

Additional methods that depend on the presence or absence of a contained value are provided, such as `orElse()` (returns a default value if no value is present) and `ifPresent()` (performs an action if a value is present).

This is a value-based class; programmers should treat instances that are equal as interchangeable and should not use instances for synchronization, or unpredictable behavior may occur. For example, in a future release, synchronization may fail.

API Note:

`Optional` is primarily intended for use as a method return type where there is a clear need to represent "no result," and where using `null` is likely to cause errors. A variable whose type is `Optional` should never itself be `null`; it should always point to an `Optional` instance.

Since:

1.8

Method Summary

All Methods Static Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
static <T> **Optional**<T>	**empty**()	Returns an empty `Optional` instance.
boolean	**equals**(**Object** obj)	Indicates whether some other object is "equal to" this `Optional`.
Optional<T>	**filter**(**Predicate**<? super T> predicate)	If a value is present, and the value matches the given predicate, returns an `Optional` describing the value, otherwise returns an empty `Optional`.

<U> **Optional**<U>	**flatMap**(**Function**<? super **T**,? extends **Optional**<? extends U>> mapper)	If a value is present, returns the result of applying the given Optional-bearing mapping function to the value, otherwise returns an empty Optional.
T	**get**()	If a value is present, returns the value, otherwise throws NoSuchElementException.
int	**hashCode**()	Returns the hash code of the value, if present, otherwise 0 (zero) if no value is present.
void	**ifPresent**(**Consumer**<? super **T**> action)	If a value is present, performs the given action with the value, otherwise does nothing.
void	**ifPresentOrElse**(**Consumer**<? super **T**> action, **Runnable** emptyAction)	If a value is present, performs the given action with the value, otherwise performs the given empty-based action.
boolean	**isEmpty**()	If a value is not present, returns true, otherwise false.
boolean	**isPresent**()	If a value is present, returns true, otherwise false.
<U> **Optional**<U>	**map**(**Function**<? super **T**,? extends U> mapper)	If a value is present, returns an Optional describing (as if by ofNullable(T)) the result of applying the given mapping function to the value, otherwise returns an empty Optional.
static <T> **Optional**<T>	**of**(T value)	Returns an Optional describing the given non-null value.
static <T> **Optional**<T>	**ofNullable**(T value)	Returns an Optional describing the given value, if non-null, otherwise returns an empty Optional.
Optional<**T**>	**or**(**Supplier**<? extends **Optional**<? extends **T**>> supplier)	If a value is present, returns an Optional describing the value, otherwise returns an

		Optional produced by the supplying function.
T	orElse(T other)	If a value is present, returns the value, otherwise returns other.
T	orElseGet(Supplier<? extends T> supplier)	If a value is present, returns the value, otherwise returns the result produced by the supplying function.
T	orElseThrow()	If a value is present, returns the value, otherwise throws NoSuchElementException.
<X extends Throwable> T	orElseThrow(Supplier<? extends X> exceptionSupplier)	If a value is present, returns the value, otherwise throws an exception produced by the exception supplying function.
Stream<T>	stream()	If a value is present, returns a sequential Stream containing only that value, otherwise returns an empty Stream.
String	toString()	Returns a non-empty string representation of this Optional suitable for debugging.

Methods declared in class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Method Details

empty

```
public static <T> Optional<T> empty()
```

Returns an empty Optional instance. No value is present for this Optional.

API Note:

Though it may be tempting to do so, avoid testing if an object is empty by comparing with == or != against instances returned by Optional.empty(). There is no guarantee that it is a singleton. Instead, use isEmpty() or isPresent().

Type Parameters:

T - The type of the non-existent value

Returns:

an empty `Optional`

of

`public static <T> Optional<T> of(T value)`

Returns an `Optional` describing the given non-`null` value.

Type Parameters:

`T` - the type of the value

Parameters:

`value` - the value to describe, which must be non-`null`

Returns:

an `Optional` with the value present

Throws:

`NullPointerException` - if value is `null`

ofNullable

`public static <T> Optional<T> ofNullable(T value)`

Returns an `Optional` describing the given value, if non-`null`, otherwise returns an empty `Optional`.

Type Parameters:

`T` - the type of the value

Parameters:

`value` - the possibly-`null` value to describe

Returns:

an `Optional` with a present value if the specified value is non-`null`, otherwise an empty `Optional`

get

`public T get()`

If a value is present, returns the value, otherwise throws `NoSuchElementException`.

API Note:

The preferred alternative to this method is `orElseThrow()`.

Returns:

the non-`null` value described by this `Optional`

Throws:

`NoSuchElementException` - if no value is present

isPresent

```
public boolean isPresent()
```

If a value is present, returns `true`, otherwise `false`.

Returns:

`true` if a value is present, otherwise `false`

isEmpty

```
public boolean isEmpty()
```

If a value is not present, returns `true`, otherwise `false`.

Returns:

`true` if a value is not present, otherwise `false`

Since:

11

ifPresent

```
public void ifPresent(Consumer<? super T> action)
```

If a value is present, performs the given action with the value, otherwise does nothing.

Parameters:

`action` - the action to be performed, if a value is present

Throws:

`NullPointerException` - if value is present and the given action is `null`

ifPresentOrElse

```
public void ifPresentOrElse(Consumer<? super T> action,
                            Runnable emptyAction)
```

If a value is present, performs the given action with the value, otherwise performs the given empty-based action.

Parameters:

`action` - the action to be performed, if a value is present

`emptyAction` - the empty-based action to be performed, if no value is present

Throws:

`NullPointerException` - if a value is present and the given action is `null`, or no value is present and the given empty-based action is `null`.

Since:

9

filter

```
public Optional<T> filter(Predicate<? super T> predicate)
```

If a value is present, and the value matches the given predicate, returns an `Optional` describing the value, otherwise returns an empty `Optional`.

Parameters:

`predicate` - the predicate to apply to a value, if present

Returns:

an `Optional` describing the value of this `Optional`, if a value is present and the value matches the given predicate, otherwise an empty `Optional`

Throws:

`NullPointerException` - if the predicate is `null`

map

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

If a value is present, returns an `Optional` describing (as if by `ofNullable(T)`) the result of applying the given mapping function to the value, otherwise returns an empty `Optional`.

If the mapping function returns a `null` result then this method returns an empty `Optional`.

API Note:

This method supports post-processing on `Optional` values, without the need to explicitly check for a return status. For example, the following code traverses a stream of URIs, selects one that has not yet been processed, and creates a path from that URI, returning an `Optional<Path>`:

```
Optional<Path> p =
    uris.stream().filter(uri -> !isProcessedYet(uri))
                 .findFirst()
                 .map(Paths::get);
```

Here, `findFirst` returns an `Optional<URI>`, and then `map` returns an `Optional<Path>` for the desired URI if one exists.

Type Parameters:

`U` - The type of the value returned from the mapping function

Parameters:

`mapper` - the mapping function to apply to a value, if present

Returns:

an `Optional` describing the result of applying a mapping function to the value of this `Optional`, if a value is present, otherwise an empty `Optional`

Throws:

NullPointerException - if the mapping function is null

flatMap

```
public <U> Optional<U> flatMap
(Function<? super T,? extends Optional<? extends U>> mapper)
```

If a value is present, returns the result of applying the given Optional-bearing mapping function to the value, otherwise returns an empty Optional.

This method is similar to map(Function), but the mapping function is one whose result is already an Optional, and if invoked, flatMap does not wrap it within an additional Optional.

Type Parameters:

U - The type of value of the Optional returned by the mapping function

Parameters:

mapper - the mapping function to apply to a value, if present

Returns:

the result of applying an Optional-bearing mapping function to the value of this Optional, if a value is present, otherwise an empty Optional

Throws:

NullPointerException - if the mapping function is null or returns a null result

or

```
public Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)
```

If a value is present, returns an Optional describing the value, otherwise returns an Optional produced by the supplying function.

Parameters:

supplier - the supplying function that produces an Optional to be returned

Returns:

returns an Optional describing the value of this Optional, if a value is present, otherwise an Optional produced by the supplying function.

Throws:

NullPointerException - if the supplying function is null or produces a null result

Since:

9

stream

```
public Stream<T> stream()
```

If a value is present, returns a sequential Stream containing only that value, otherwise returns an empty Stream.

API Note:

This method can be used to transform a `Stream` of optional elements to a `Stream` of present value elements:

```
Stream<Optional<T>> os = ..
Stream<T> s = os.flatMap(Optional::stream)
```

Returns:

the optional value as a `Stream`

Since:

9

orElse

`public T orElse(T other)`

If a value is present, returns the value, otherwise returns `other`.

Parameters:

`other` - the value to be returned, if no value is present. May be `null`.

Returns:

the value, if present, otherwise `other`

orElseGet

`public T orElseGet(Supplier<? extends T> supplier)`

If a value is present, returns the value, otherwise returns the result produced by the supplying function.

Parameters:

`supplier` - the supplying function that produces a value to be returned

Returns:

the value, if present, otherwise the result produced by the supplying function

Throws:

`NullPointerException` - if no value is present and the supplying function is `null`

orElseThrow

`public T orElseThrow()`

If a value is present, returns the value, otherwise throws `NoSuchElementException`.

Returns:

the non-`null` value described by this `Optional`

Throws:

`NoSuchElementException` - if no value is present

Since:

10

orElseThrow

```
public <X extends Throwable> T orElseThrow
(Supplier<? extends X> exceptionSupplier)
                                    throws X
```

If a value is present, returns the value, otherwise throws an exception produced by the exception supplying function.

API Note:

A method reference to the exception constructor with an empty argument list can be used as the supplier. For example, `IllegalStateException::new`

Type Parameters:

`X` - Type of the exception to be thrown

Parameters:

`exceptionSupplier` - the supplying function that produces an exception to be thrown

Returns:

the value, if present

Throws:

`X` - if no value is present

`NullPointerException` - if no value is present and the exception supplying function is `null`

equals

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this `Optional`. The other object is considered equal if:

- it is also an `Optional` and;
- both instances have no value present or;
- the present values are "equal to" each other via `equals()`.

Overrides:

`equals` in class `Object`

Parameters:

`obj` - an object to be tested for equality

Returns:

`true` if the other object is "equal to" this object otherwise `false`

See Also:

`Object.hashCode()`, `HashMap`

hashCode

```
public int hashCode()
```

Returns the hash code of the value, if present, otherwise `0` (zero) if no value is present.

Overrides:

`hashCode` in class `Object`

Returns:

hash code value of the present value or `0` if no value is present

See Also:

`Object.equals(java.lang.Object)`,
`System.identityHashCode(java.lang.Object)`

toString

```
public String toString()
```

Returns a non-empty string representation of this `Optional` suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions.

Overrides:

`toString` in class `Object`

Implementation Requirements:

If a value is present the result must include its string representation in the result. Empty and present `Optional`s must be unambiguously differentiable.

Returns:

the string representation of this instance