`Optional` class

The `Optional` class in Java, introduced in Java 8, is a container object that may or may not contain a non-null value. It is primarily used to avoid `NullPointerException` and provides a more graceful way to handle potentially null values.

Why Use `Optional`?

Traditionally, checking for null values in Java often involves null checks, which can lead to verbose and error-prone code:

```
if (object != null) {
    // do something with object
}
```

With `Optional`, you can represent optional (or absent) values more elegantly, avoiding null checks and providing better readability.

Creating an `Optional` Object

There are several ways to create an `Optional` object:

1. **`Optional.of()`**: Creates an `Optional` with a non-null value. If the value is null, it throws a `NullPointerException`.

    ```
    Optional<String> optional = Optional.of("Hello, World!");
    ```

2. **`Optional.ofNullable()`**: Creates an `Optional` that may hold a null value. If the value is null, it creates an empty `Optional`.

    ```
    Optional<String> optional = Optional.ofNullable(null);
    ```

3. **`Optional.empty()`**: Creates an empty `Optional` (i.e., an Optional that holds no value).

```java
Optional<String> optional = Optional.empty();
```

Methods in `Optional`

The `Optional` class provides several methods to work with the contained value:

1. `isPresent()` and `isEmpty()`

- `isPresent()`: Returns `true` if a value is present, otherwise `false`.
- `isEmpty()`: Returns `true` if no value is present (i.e., the `Optional` is empty), otherwise `false`.

  ```java
  Optional<String> optional = Optional.of("Hello");
  if (optional.isPresent()) {
      System.out.println(optional.get()); // Output: Hello
  }
  ```

2. `get()`

Returns the value if present; otherwise, it throws `NoSuchElementException`. Use this cautiously, as it can lead to exceptions if not handled correctly.

~~~
```java
String value = optional.get(); // Only use if you are sure the value is present.
```
~~~

3. `orElse()`

Returns the value if present; otherwise, returns the default value provided.

~~~
```java
String value = optional.orElse("Default Value");
```
~~~

4. `orElseGet()`

Similar to `orElse()`, but takes a `Supplier` functional interface. The `Supplier` is only called if the value is not present, making it more efficient when computing default values.

~~~
```java
String value = optional.orElseGet(() -> "Default Value");
```
~~~

5. `orElseThrow()`

Returns the value if present; otherwise, throws an exception. You can pass a custom exception using a `Supplier`.

~~~
```java
String value = optional.orElseThrow(() -> new IllegalArgument
Exception("Value not present"));
```
~~~

6. `ifPresent()` and `ifPresentOrElse()`

- `ifPresent()`: Executes a block of code if a value is present.

  ```
  optional.ifPresent(value -> System.out.println(value));
  ```

- `ifPresentOrElse()`: Executes a block of code if a value is present; otherwise, executes a different block of code.

  ```
  optional.ifPresentOrElse(
      value -> System.out.println(value),
      () -> System.out.println("Value not present")
  );
  ```

7. `map()`

Applies a function to the value if present and returns an `Optional` of the result. If no value is present, it returns an empty `Optional`.

```java
Optional<String> optional = Optional.of("hello");
Optional<String> upperCase = optional.map(String::toUpperCase);
upperCase.ifPresent(System.out::println); // Output: HELLO
```

8. `flatMap()`

Similar to `map()`, but the function should return an `Optional`. It's useful for chaining multiple `Optional` operations.

```java
Optional<String> optional = Optional.of("hello");
Optional<String> flatMapped = optional.flatMap(value -> Optional.of(value.toUpperCase()));
flatMapped.ifPresent(System.out::println); // Output: HELLO
```

9. `filter()`

Filters the value using a predicate. If the value matches the predicate, it returns the same `Optional`; otherwise, it returns an empty `Optional`.

```java
Optional<String> optional = Optional.of("hello");
Optional<String> filtered = optional.filter(value -> value.startsWith("h"));
filtered.ifPresent(System.out::println); // Output: hello
```

When to Use `Optional`

1. **Return Type for Methods**: Use `Optional` as a return type for methods where the result may be null. For example, fetching a value from a database that might not exist.

   ```
   public Optional<User> findUserById(int id) {
       // If user is found, return Optional.of(user)
       // If not found, return Optional.empty()
   }
   ```

2. **Avoid Passing `Optional` as Parameters**: It's generally not recommended to use `Optional` as method parameters. Instead, pass non-null values or handle null checks within the method.
3. **Avoid Using `Optional` in Fields**: `Optional` is not designed to be used for class fields. It's better to handle null checks directly or use non-null fields.

Drawbacks of `Optional`

- **Performance Overhead**: Creating `Optional` objects adds some performance overhead, which might be unnecessary in performance-critical applications.
- **Misuse in Fields and Parameters**: While `Optional` is useful as a return type, using it in fields or method parameters is generally discouraged.

Conclusion

The `Optional` class in Java provides a robust way to handle null values without the risk of `NullPointerException`. By using its various methods like `orElse()`, `map()`, `filter()`, and `ifPresent()`, you can create clean and concise code that gracefully handles optional values.