Introduce Null Object

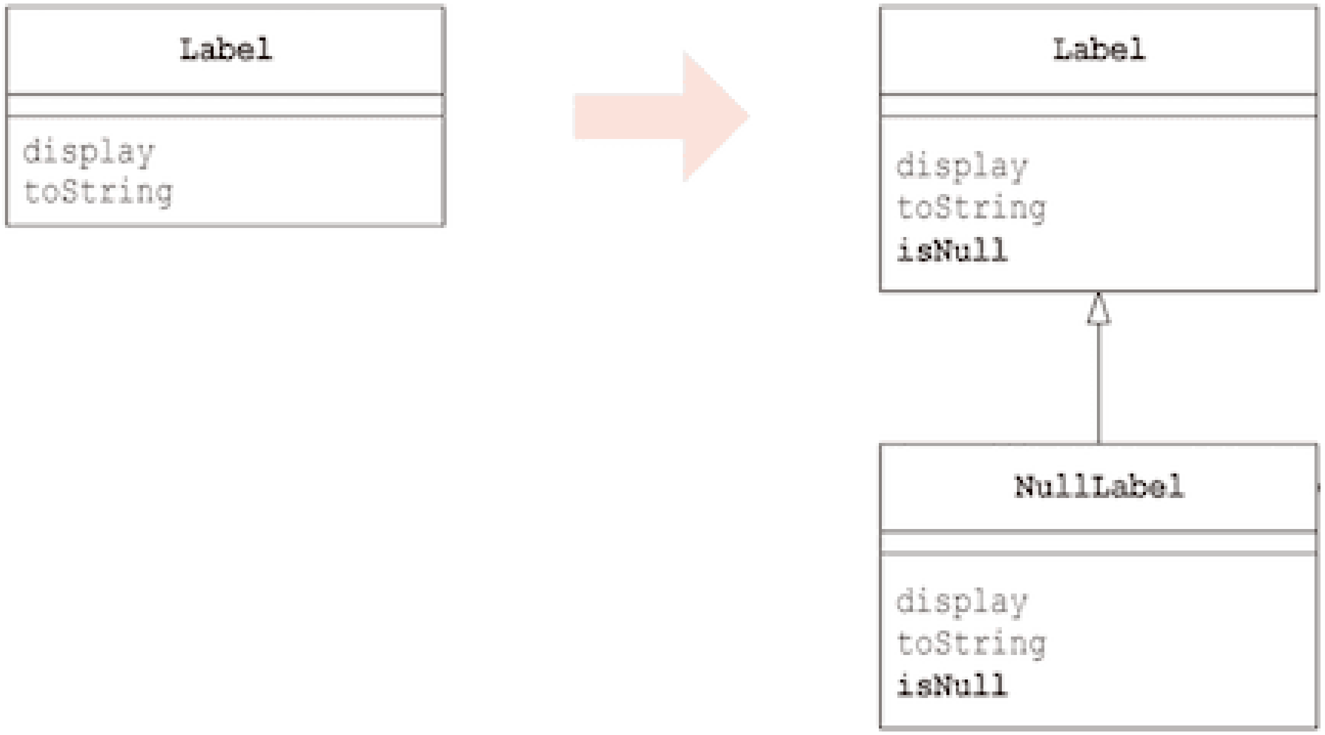
Replace `null` with an object that provides **neutral default behavior** to avoid null checks.

Code Smell

```
# Too much null check: In Python, None is null.
if _name is not None:
    _name.display()

=>

_name.display()
```


Label
display
toString
Label
display
toString
isNull
NullLabel
display
toString
isNull

Example: Null Object

We have a Person object that has the Label and Mail

```python
class Label:
    def __init__(self, label: str):
        self.label = label
    def display(self):
        print(f"display: {self.label}")
    def __str__(self):
        return f'"{self.label}"'
class Mail:
    ...
```

Because name and mail can be None, we should check before using them.

```
class Person:
    def __init__(self, name: Label, mail: Optional[Label] = None):
        self.name = name
        self.mail = mail

    def display(self):
        def __init__(self, name: Label,
            mail: Optional[Label] = None):
        self.name = name
        """Display person info with null checks"""
        if self.name is not None:
            self.name.display()
        if self.mail is not None:
            self.mail.display()
        ...
```

This makes the code ugly and hard to read.

```python
def __str__(self):
    """String representation with null checks"""
    result = "[ Person:"

    result += " name="
    if self.name is None: result += '"(none)"'
    else: result += str(self.name)

    result += " mail="
    if self.mail is None: result += '"(none)"'
    else: result += str(self.mail)

    result += " ]"

    return result
```

After: Introduce a NullLabel

```
class Label:
    ...
    def is_null(self) -> bool:
        return False

class NullLabel(Label):
    def __init__(self):
        super().__init__("(none)")
    def display(self):
        pass
    def is_null(self) -> bool:
        return True
```

self.mail becomes NullLabel when None is assigned

```python
class Person:
    def __init__(self, name: Label, mail: Label = None):
        self.name = name
        self.mail = mail if mail is not None else NullLabel()
```

The code becomes much simpler.

```python
def display(self):
    # no null check due to the Null Object
    self.name.display()
    self.mail.display()

def __str__(self):
    return f"[ Person: name={self.name} mail={self.mail} ]"
```

Run Unit Test

```python
from Label import Label
from Person import Person

class MainTest(StandardOutputTest):
    def test_label(self):
        alice = Label("Alice")
        expected = '"Alice"'
        actual = str(alice)
        self.assertEqual(expected, actual)
...
```

It should return all pass!

Refactoring to use Singleton & Factory

- Singleton
  - There is only one NullLabel object, so we can use the singleton.

```python
class NullLabel(Label):
    _instance = None
    ...
    @classmethod
    def get_instance(cls):
        if cls._instance is None: cls._instance = cls()
        return cls._instance
```

```
NullLabel()

=>

NullLabel.get_instance()
```

- Factory
    - We can use classmethod (or static method) to use the factory method.

```python
class Label:
    ...
    @staticmethod
    def new_null():
        """Factory method for null object"""
        return NullLabel.get_instance()
```

- The Person class needs to import only Label.
- We can specify the name and mail as private fields.

```python
from Label_singleton import Label

class Person:
    def __init__(self, name, mail=None):
        self._name = name
        self._mail = mail
          if mail is not None else Label.new_null()
```

Run Unit Test Again

We make sure the refactoring does not change external behavior.

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Discussion

We should implement Null Object Class in these conditions

- **Implement the same interface** as the real object
- **Provide neutral/default behavior** for all methods
- **Return sensible default values (not Null)**
- **Avoid throwing exceptions** when possible

Benefits of Null Object

1. **Eliminates null checks** - no need for repetitive null checking
2. **Reduces errors** - prevents null pointer exceptions
3. **Simplifies code** - cleaner, more readable code
4. **Polymorphic behavior** - null object implements same interface