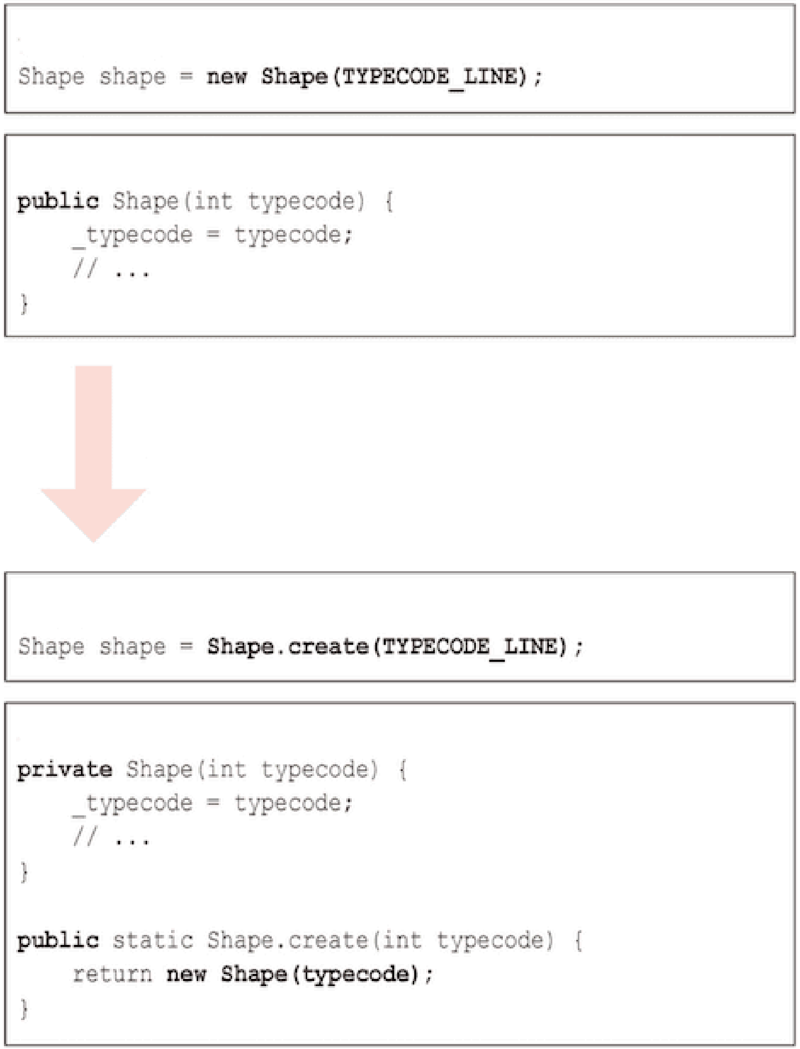
Replace Constructor with Factory Method

Replace direct **constructor calls** with a **factory method** for flexible object creation.

Complex constructors can be **confusing with many parameters**, **cannot have descriptive names**, **cannot return subclasses**, and make **object creation inflexible**.

- We already discussed the importance of factory method.

```
s = Something() # not flexible
s = Something.create() # flexible
```



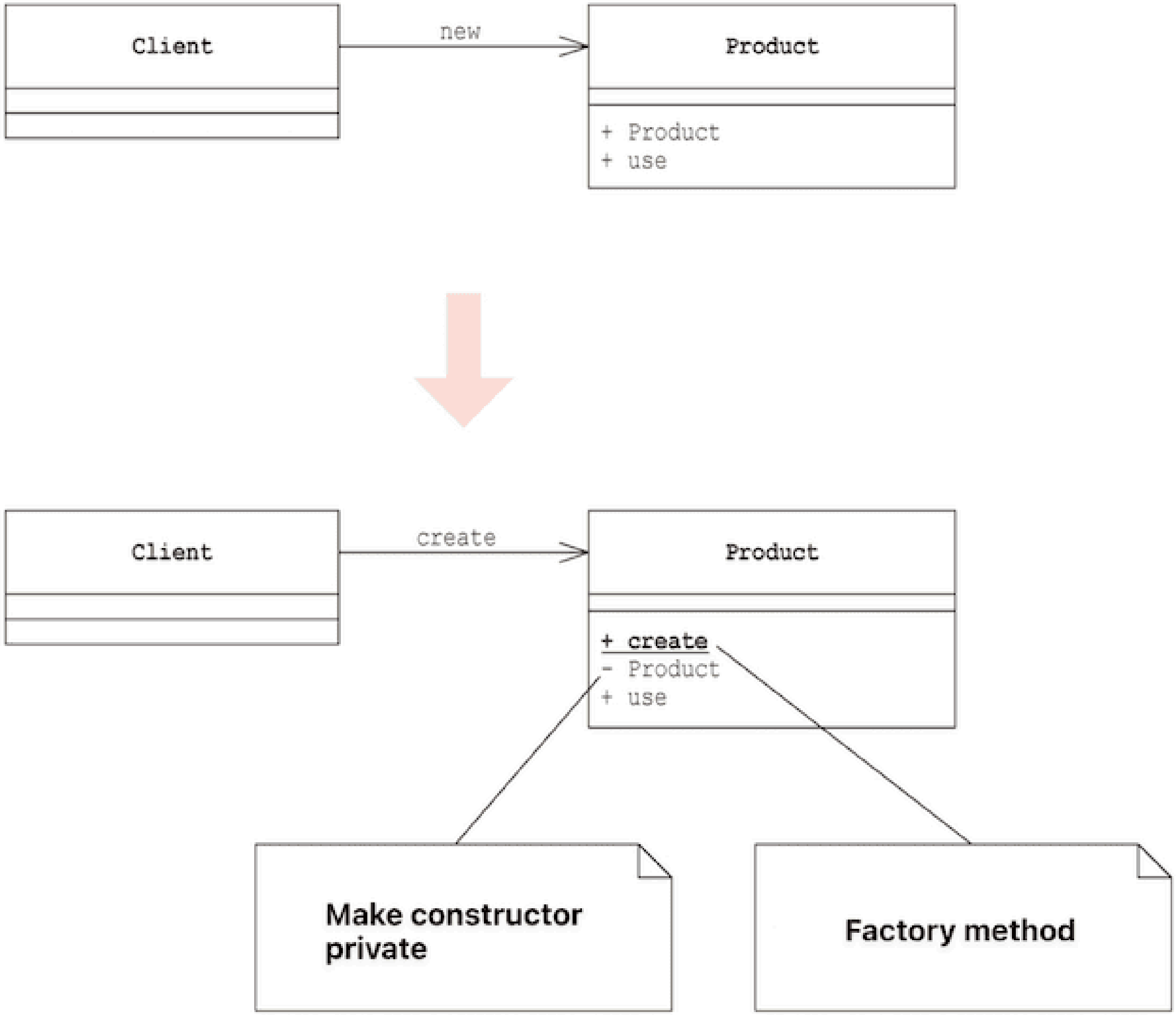
```
Shape shape = new Shape(TYPECODE_LINE);
```

```
public Shape(int typecode) {
    _typecode = typecode;
    // ...
}
```

```
Shape shape = Shape.create(TYPECODE_LINE);
```

```
private Shape(int typecode) {
    _typecode = typecode;
    // ...
}

public static Shape.create(int typecode) {
    return new Shape(typecode);
}
```



Make constructor private

Factory method

Example: Shape

- In this example, we refactor the Shape class using two steps.
  - In the first step, we introduce the factory.
  - In the second step, we remove the type code using polymorphism.

- The constructor uses typecode to identify the shape.

```
class Shape:
    TYPECODE_LINE = 0
    TYPECODE_RECTANGLE = 1
    TYPECODE_OVAL = 2
    def __init__(self, typecode: int,
      startx: int, starty: int,
      endx: int, endy: int):
        self.typecode = typecode
        self.startx = startx
        self.starty = starty
        self.endx = endx
        self.endy = endy
```

- Using type code is a code smell that can be removed with polymorphism.

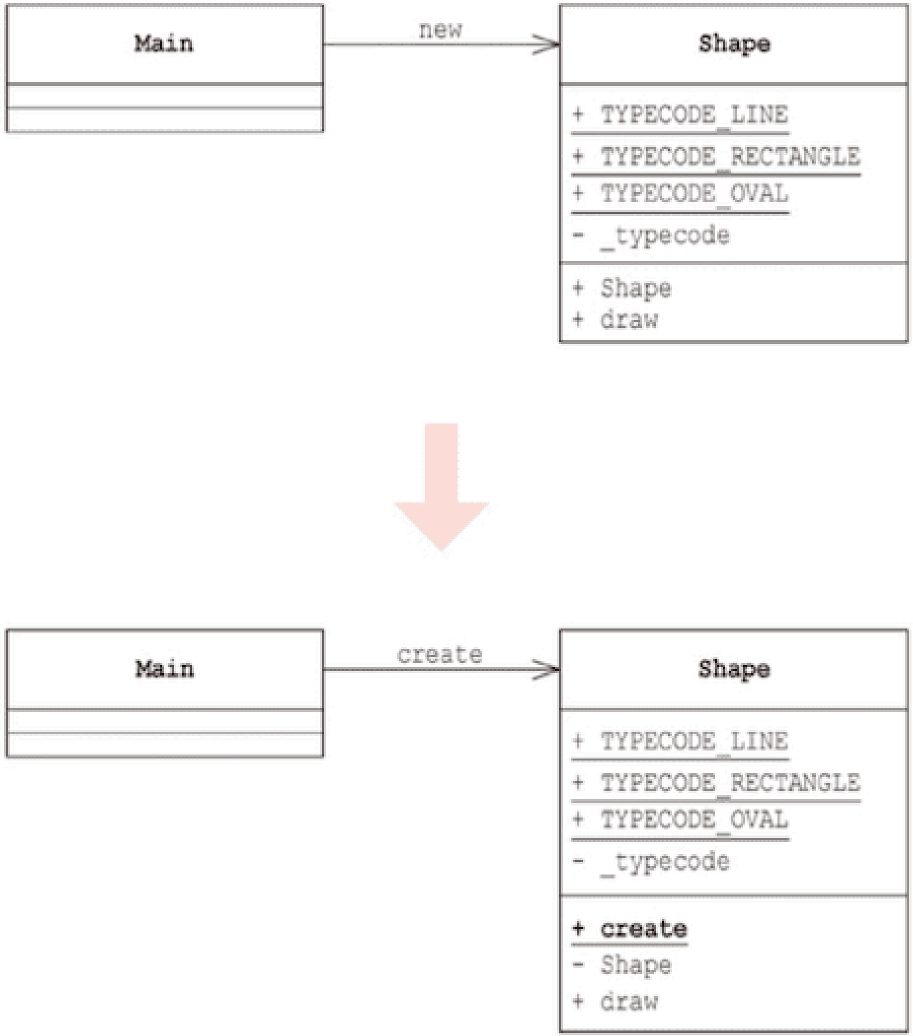
```python
def get_name(self) -> str:
    if self.typecode == Shape.TYPECODE_LINE:
        return "LINE"
    ...
    else:
        return None
def draw(self):
    if self.typecode == Shape.TYPECODE_LINE:
        self._draw_line()
    ...
    elif self.typecode == Shape.TYPECODE_OVAL:
        self._draw_oval()
```

Refactoring: Factory

- In this first refactoring, we make the factory `create`.

```
class Shape:
    TYPECODE_LINE = 0
    TYPECODE_RECTANGLE = 1
    TYPECODE_OVAL = 2

    @staticmethod
    def create(typecode: int,
      startx: int, starty: int,
      endx: int, endy: int) -> 'Shape':
        return Shape(typecode,
          startx, starty, endx, endy)
```



Main

new

Shape

+ TYPECODE_LINE

+ TYPECODE_RECTANGLE

+ TYPECODE_OVAL

- _typecode

+ Shape

+ draw

Main

create

Shape

+ TYPECODE_LINE

+ TYPECODE_RECTANGLE

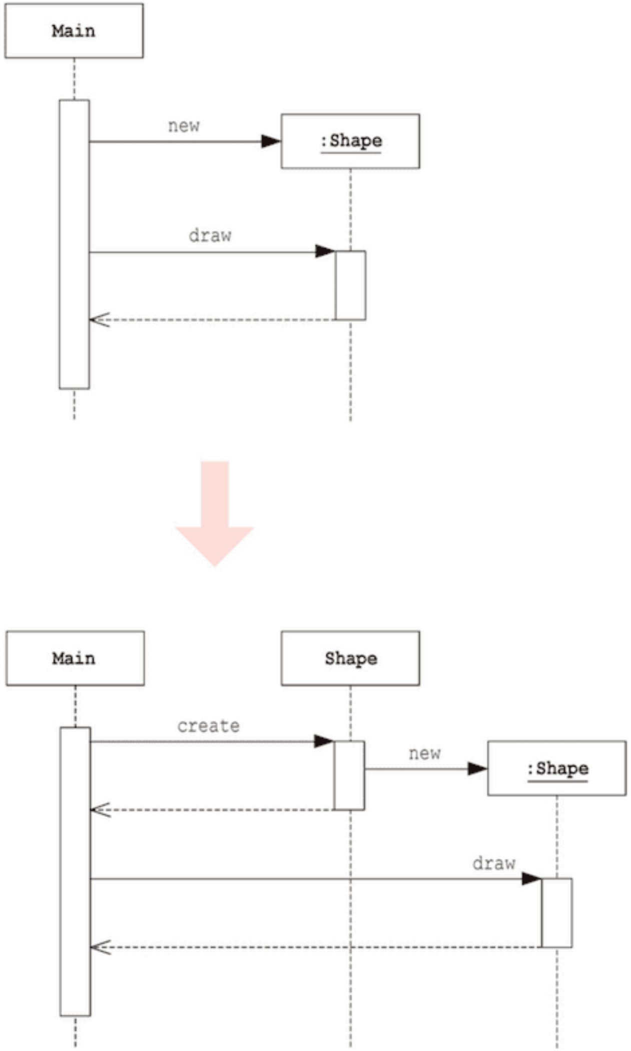
+ TYPECODE_OVAL

- _typecode

+ create

- Shape

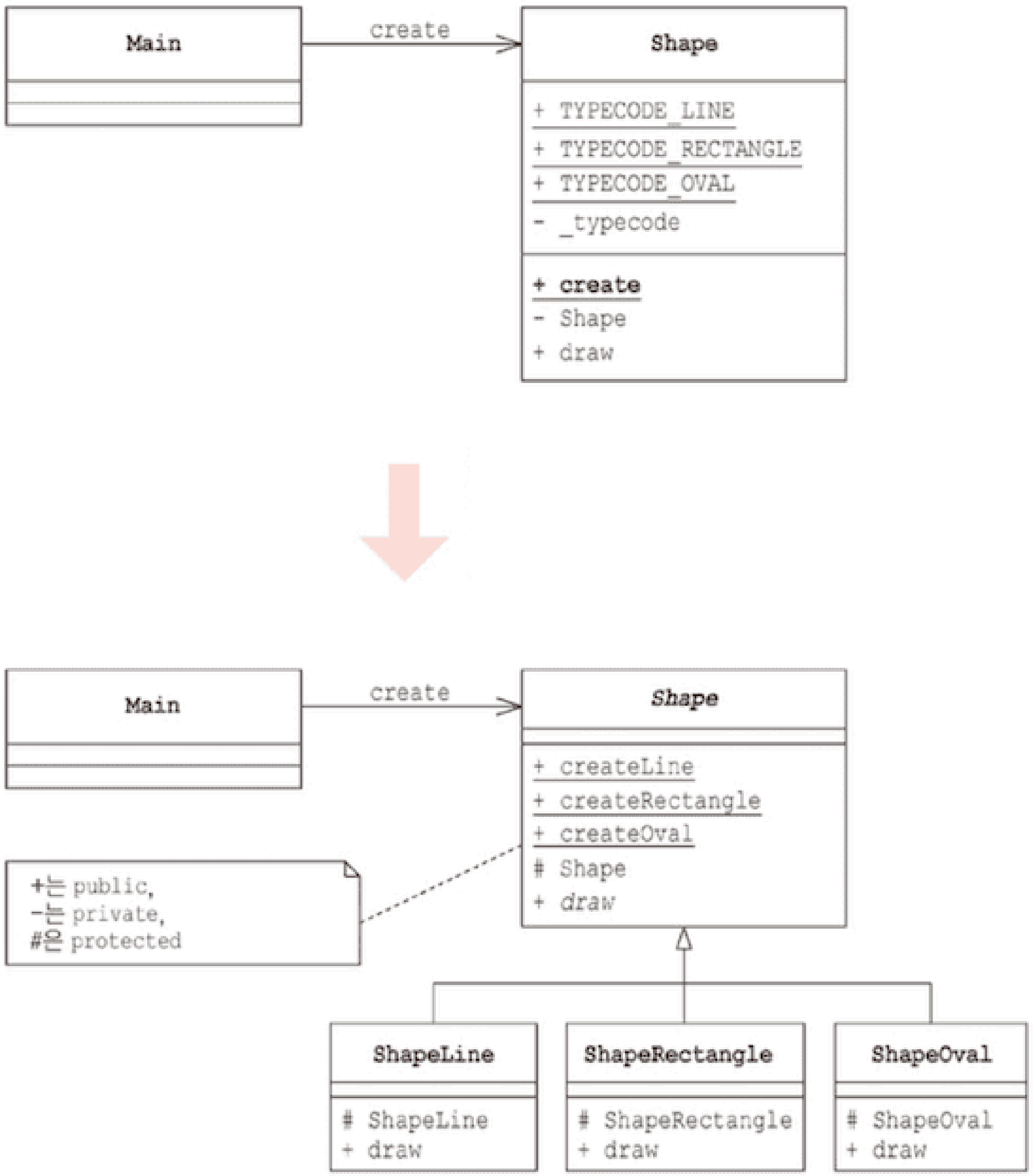
+ draw


Main
new
:Shape
draw
Main
Shape
create
new
:Shape
draw

Refactoring: Remove type code

- In this second refactoring, we use polymorphism to remove the type code and if statements.

```
class ShapeLine(Shape):
    def __init__(self,
      startx: int, starty: int, endx: int, endy: int):
        super().__init__(startx, starty, endx, endy)
    def get_name(self) -> str: ...
    def draw(self): ...
```



Main

create

Shape

+ TYPECODE_LINE
+ TYPECODE_RECTANGLE
+ TYPECODE_OVAL
- _typecode

+ create
- Shape
+ draw

Main

create

Shape

+ createLine
+ createRectangle
+ createOval
Shape
+ *draw*

+는 public,
−는 private,
#은 protected

ShapeLine

ShapeLine
+ draw

ShapeRectangle

ShapeRectangle
+ draw

ShapeOval

ShapeOval
+ draw

Unit tests

- In this example test, we use `assertEqual` to compare the values.

```
class TestShape(unittest.TestCase):
    def test_constructor_creates_line(self):
        shape = Shape(Shape.TYPECODE_LINE, 10, 20, 30, 40)

        self.assertEqual(shape.get_typecode(), Shape.TYPECODE_LINE)
        self.assertEqual(shape.get_name(), "LINE")
        self.assertEqual(shape.startx, 10)
        self.assertEqual(shape.starty, 20)
        self.assertEqual(shape.endx, 30)
        self.assertEqual(shape.endy, 40)
```

Discussion

Benefits of Replace Constructor with Factory

1. **Descriptive names** - method names explain what's being created
2. **Return subclasses** - can return appropriate subclass based on parameters
3. **Control object creation** - can implement caching, pooling, or validation
4. **Multiple creation methods** - different ways to create the same object
5. **Hide complexity** - encapsulate complex initialization logic

Consider the following when designing Factory Method

- **Clear, descriptive names** that explain what's being created
- **Consistent parameter order** across related factory methods
- **Error handling** for invalid parameters
- **Documentation** of what each factory method creates
- **Consider caching** if object creation is expensive