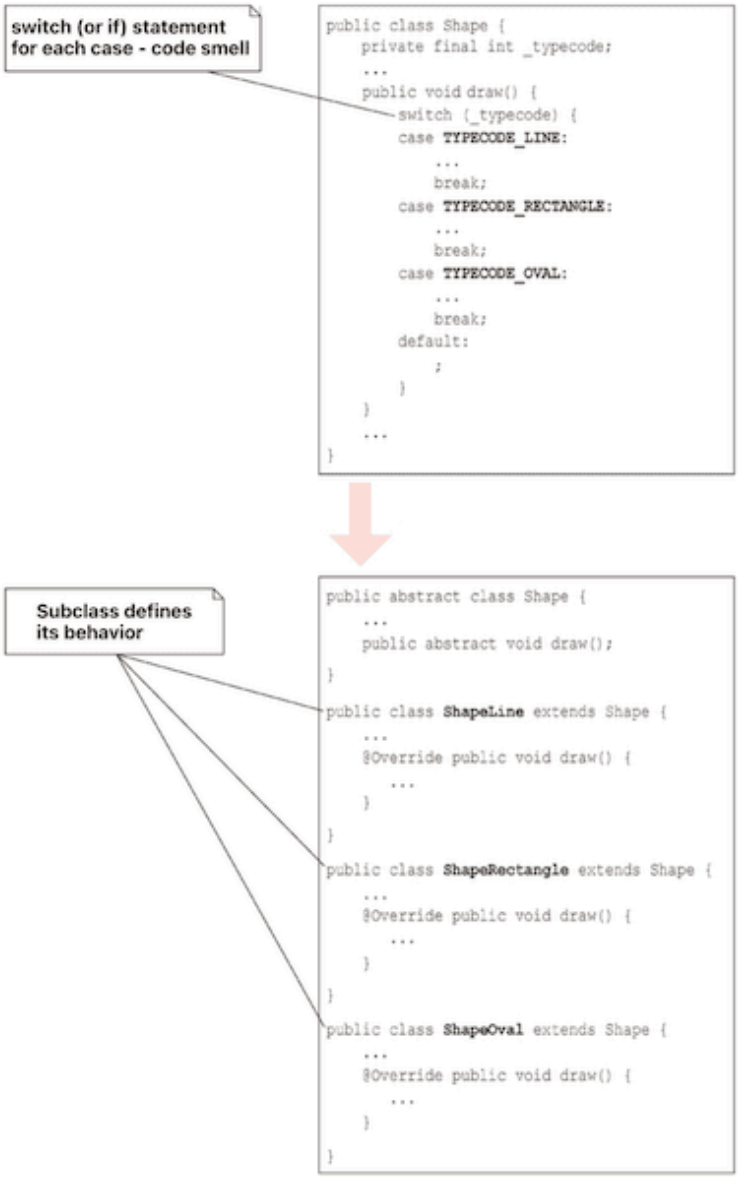
Replace Type Code with SubClass

Replace a **type code that affects behavior** with **subclasses**, each implementing the specific behavior.

The **type code affects the object's behavior** and you have **conditional logic** that varies based on the type code.

- When we have to add `if` statement when we add new features, it's a code smell.
- We introduce polymorphism to change object behavior, which solves this issue.


switch (or if) statement
for each case - code smell
public class Shape {
private final int _typecode;
...
public void draw() {
switch (_typecode) {
case TYPECODE_LINE:
...
break;
case TYPECODE_RECTANGLE:
...
break;
case TYPECODE_OVAL:
...
break;
default:
;
}
}
...
}
Subclass defines
its behavior
public abstract class Shape {
...
public abstract void draw();
}
public class ShapeLine extends Shape {
...
@Override public void draw() {
...
}
}
public class ShapeRectangle extends Shape {
...
@Override public void draw() {
...
}
}
public class ShapeOval extends Shape {
...
@Override public void draw() {
...
}
}

Example: Shape

Before:

- We have a class Shape that represents different shapes.

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

- When we use type code, we need to add if statements to extend the feature.

```
def get_name(self) -> str:
    if self.typecode == Shape.TYPECODE_LINE:
        return "LINE"
    elif self.typecode == Shape.TYPECODE_RECTANGLE:
        return "RECTANGLE"
    ...
def draw(self):
    if self.typecode == Shape.TYPECODE_LINE:
        self._draw_line()
    elif self.typecode == Shape.TYPECODE_RECTANGLE:
        self._draw_rectangle()
    ...
```

- The Shape uses constructor (`__init__`) to instantiate the Shape object.

```
line = Shape(Shape.TYPECODE_LINE, 0, 0, 100, 00)
rectangle = Shape(Shape.TYPECODE_RECTANGLE, 0, 20, 30, 40)
oval = Shape(Shape.TYPECODE_OVAL, 100, 200, 300, 400)
```

After:

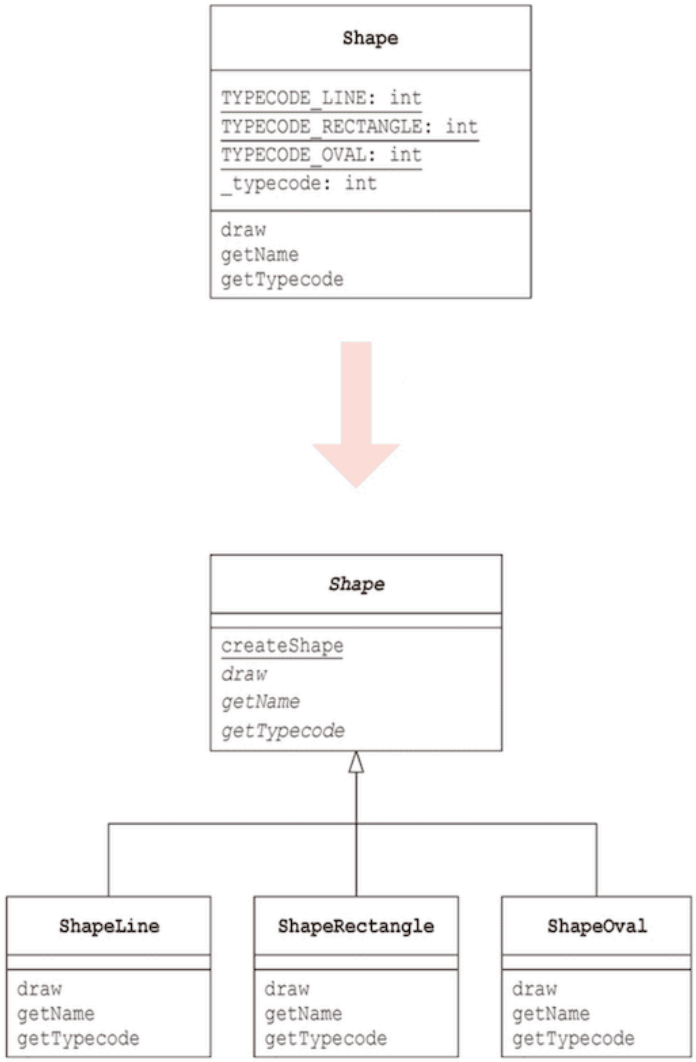
- We use an abstract class and its factory method to create subclass objects.

```
class Shape(ABC):
    TYPECODE_LINE = 0
    TYPECODE_RECTANGLE = 1
    @staticmethod
    def create_shape(typecode: int,
      startx: int, starty: int, endx: int, endy: int)
        if typecode == Shape.TYPECODE_LINE:
            from ShapeLine import ShapeLine
            return ShapeLine(startx, starty, endx, endy)
```

Subclasses

- Each subclass implements the abstract super class.

```
from Shape import Shape
class ShapeLine(Shape):
    def __init__(self, startx: int, starty: int, endx: int, endy: int):
        super().__init__(startx, starty, endx, endy)
    def get_typecode(self) -> int:
        return Shape.TYPECODE_LINE
    def get_name(self) -> str:
        return "LINE"
    def draw(self):
        self._draw_line()
    def _draw_line(self):
        print(f"drawLine: {self}")
```


Shape
TYPECODE_LINE: int
TYPECODE_RECTANGLE: int
TYPECODE_OVAL: int
_typecode: int
draw
getName
getTypecode
Shape
createShape
draw
getName
getTypecode
ShapeLine
draw
getName
getTypecode
ShapeRectangle
draw
getName
getTypecode
ShapeOval
draw
getName
getTypecode

Refactoring: Use Factory Method

- Instead of using constructor, we use the create_shape factory method.

```
line = Shape(
  Shape.TYPECODE_LINE, 0, 0, 100, 00)
rectangle = Shape(
  Shape.TYPECODE_RECTANGLE, 0, 20, 30, 40)

=>

line = Shape.create_shape(
  Shape.TYPECODE_LINE, 0, 0, 100, 200)
rectangle = Shape.create_shape(
  Shape.TYPECODE_RECTANGLE, 10, 20, 30, 40)
```

Refactoring: Use Factory Method for SubClasses

- We can remove the type entirely by introducing a factory method for each subclass.
- In this case, we should duplicate the create static method.

The TypeCode is not used anymore:

```
class ShapeOval(Shape):
    @staticmethod
    def create(startx: int, starty: int,
        endx: int, endy: int)
        return ShapeOval(startx, starty, endx, endy)
...
class ShapeLine(Shape):
    @staticmethod
    def create(startx: int, starty: int,
        endx: int, endy: int)
        return ShapeLine(startx, starty, endx, endy)

# Usage
line = ShapeLine.create(0, 0, 100, 200)
rectangle = ShapeRectangle.create(10, 20, 30, 40)
oval = ShapeOval.create(100, 200, 300, 400)
```

Discussion

Benefits of Type Code with Subclass

1. **Eliminates conditionals** - no more if/switch statements on type
2. **Polymorphism** - behavior varies automatically by type
3. **Open/Closed Principle** - can add new types without changing existing code
4. **Type safety** - compiler ensures correct method calls
5. **Clearer intent** - each subclass has a single, clear purpose

Potential Drawback

Cannot change type at runtime - once an object is created as a specific subclass, it cannot change to another type. If you need to change types dynamically, use Replace Type Code with Strategy instead.