# PROTOTYPE

Also known as: Clone

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

☹ Problem

Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.

Nice! But there's a catch. Not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.
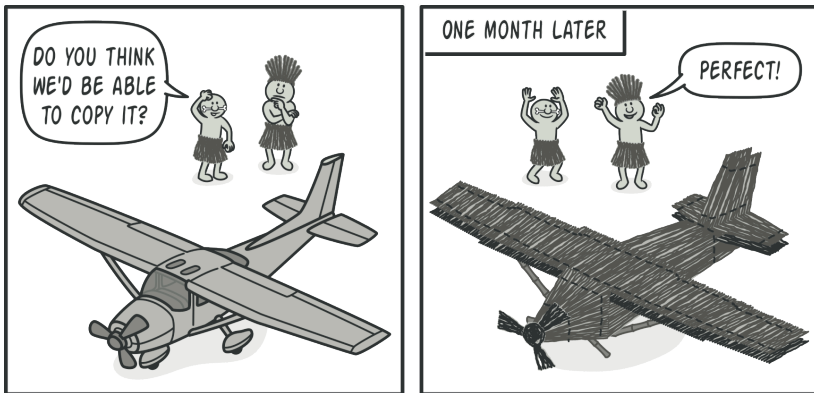
*Copying an object "from the outside" **isn't** always possible.*

There's one more problem with the direct approach. Since you have to know the object's class to create a duplicate, your code becomes dependent on that class. If the extra dependency doesn't scare you, there's another catch. Sometimes you only know the interface that the object follows, but not its concrete

class, when, for example, a parameter in a method accepts any objects that follow some interface.

😊 Solution

The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single `clone` method.

The implementation of the `clone` method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.


Pre-built prototypes can be an alternative to subclassing.

An object that supports cloning is called a *prototype*. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.

Here's how it works: you create a set of objects, configured in various ways. When you need an object like the one you've configured, you just clone a prototype instead of constructing a new object from scratch.

🚗 Real-World Analogy

In real life, prototypes are used for performing various tests before starting mass production of a product. However, in this case, prototypes don't participate in any actual production, playing a passive role instead.
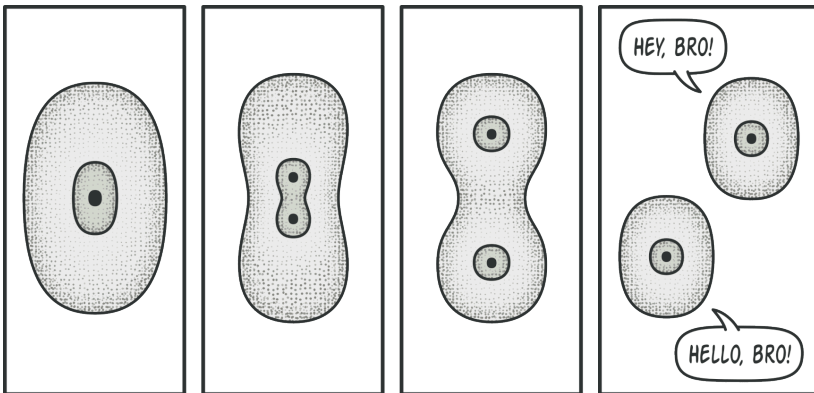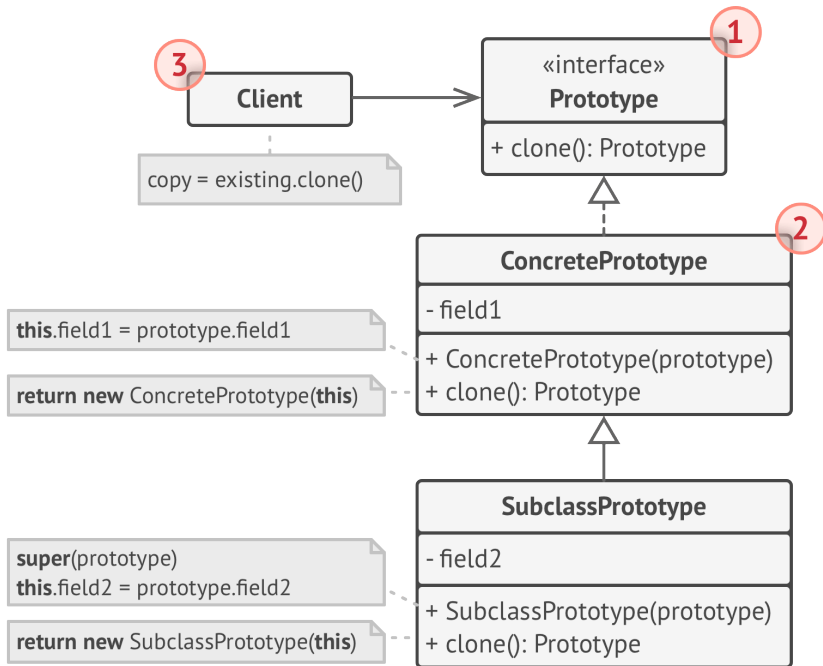

The division of a cell.

Since industrial prototypes don't really copy themselves, a much closer analogy to the pattern is the process of mitotic cell division (biology, remember?). After mitotic division, a pair of identical cells is formed. The original cell acts as a prototype and takes an active role in creating the copy.
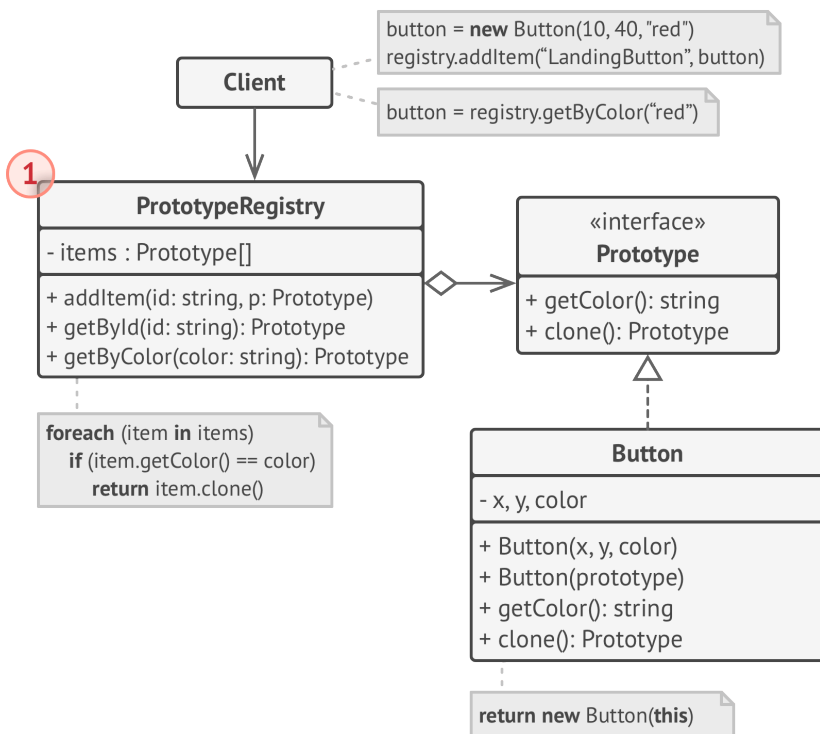
Structure

Basic implementation

1. The **Prototype** interface declares the cloning methods. In most cases, it's a single `clone` method.

2. The **Concrete Prototype** class implements the cloning method. In addition to copying the original object's data to the clone, this method may also handle some edge cases of the cloning process related to cloning linked objects, untangling recursive dependencies, etc.

3. The **Client** can produce a copy of any object that follows the prototype interface.

Prototype registry implementation

button = **new** Button(10, 40, "red")
registry.addItem("LandingButton", button)

Client

button = registry.getByColor("red")

1

PrototypeRegistry

- items : Prototype[]

+ addItem(id: string, p: Prototype)
+ getById(id: string): Prototype
+ getByColor(color: string): Prototype

foreach (item **in** items)
  if (item.getColor() == color)
    return item.clone()

«interface»
Prototype

+ getColor(): string
+ clone(): Prototype

Button

- x, y, color

+ Button(x, y, color)
+ Button(prototype)
+ getColor(): string
+ clone(): Prototype

return new Button(**this**)

1. The **Prototype Registry** provides an easy way to access fre-
    quently-used prototypes. It stores a set of pre-built objects
    that are ready to be copied. The simplest prototype registry is
    a `name → prototype` hash map. However, if you need better
    search criteria than a simple name, you can build a much more
    robust version of the registry.

Pseudocode

In this example, the **Prototype** pattern lets you produce exact
copies of geometric objects, without coupling the code to their
classes.
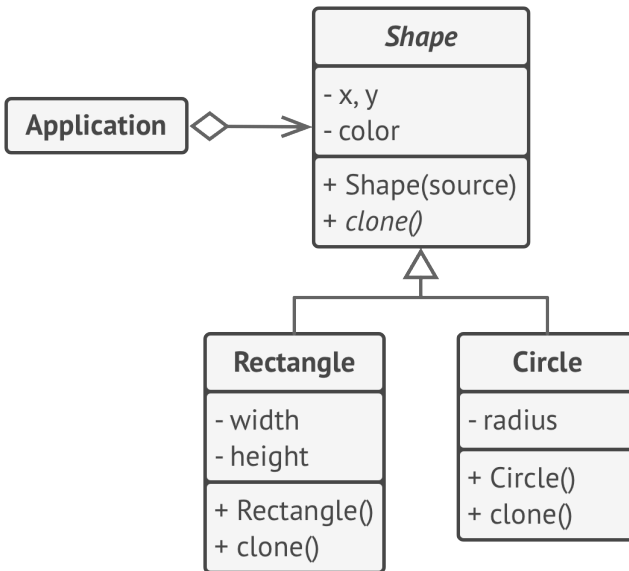
Cloning a set of objects that belong to a class hierarchy.

All shape classes follow the same interface, which provides
a cloning method. A subclass may call the parent's cloning

method before copying its own field values to the resulting object.

```
1   // Base prototype.
2   abstract class Shape is
3     field X: int
4     field Y: int
5     field color: string
6
7     // A regular constructor.
8     constructor Shape() is
9       // ...
10
11    // The prototype constructor. A fresh object is initialized
12    // with values from the existing object.
13    constructor Shape(source: Shape) is
14      this()
15      this.X = source.X
16      this.Y = source.Y
17      this.color = source.color
18
19    // The clone operation returns one of the Shape subclasses.
20    abstract method clone():Shape
21
22
23  // Concrete prototype. The cloning method creates a new object
24  // and passes it to the constructor. Until the constructor is
25  // finished, it has a reference to a fresh clone. Therefore,
26  // nobody has access to a partly-built clone. This keeps the
27  // cloning result consistent.
28  class Rectangle extends Shape is
```

```
29    field width: int
30    field height: int
31
32    constructor Rectangle(source: Rectangle) is
33      // A parent constructor call is needed to copy private
34      // fields defined in the parent class.
35      super(source)
36      this.width = source.width
37      this.height = source.height
38
39    method clone():Shape is
40      return new Rectangle(this)
41
42
43  class Circle extends Shape is
44    field radius: int
45
46    constructor Circle(source: Circle) is
47      super(source)
48      this.radius = source.radius
49
50    method clone():Shape is
51      return new Circle(this)
52
53
54  // Somewhere in the client code.
55  class Application is
56    field shapes: array of Shape
57
58    constructor Application() is
59      Circle circle = new Circle()
60      circle.X = 10
```

```
61        circle.Y = 10
62        circle.radius = 20
63        shapes.add(circle)
64
65        Circle anotherCircle = circle.clone()
66        shapes.add(anotherCircle)
67        // The `anotherCircle` variable contains an exact copy
68        // of the `circle` object.
69
70        Rectangle rectangle = new Rectangle()
71        rectangle.width = 10
72        rectangle.height = 20
73        shapes.add(rectangle)
74
75    method businessLogic() is
76        // Prototype rocks because it lets you produce a copy of
77        // an object without knowing anything about its type.
78        Array shapesCopy = new Array of Shapes.
79
80        // For instance, we don't know the exact elements in the
81        // shapes array. All we know is that they are all
82        // shapes. But thanks to polymorphism, when we call the
83        // `clone` method on a shape the program checks its real
84        // class and runs the appropriate clone method defined
85        // in that class. That's why we get proper clones
86        // instead of a set of simple Shape objects.
87        foreach (s in shapes) do
88           shapesCopy.add(s.clone())
89
90        // The `shapesCopy` array contains exact copies of the
91        // `shape` array's children.
```

⚡ Applicability

Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.

This happens a lot when your code works with objects passed to you from 3rd-party code via some interface. The concrete classes of these objects are unknown, and you couldn't depend on them even if you wanted to.

The Prototype pattern provides the client code with a general interface for working with all objects that support cloning. This interface makes the client code independent from the concrete classes of objects that it clones.

Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects. Somebody could have created these subclasses to be able to create objects with a specific configuration.

The Prototype pattern lets you use a set of pre-built objects, configured in various ways, as prototypes.

Instead of instantiating a subclass that matches some configuration, the client can simply look for an appropriate prototype and clone it.

📋 How to Implement

1. Create the prototype interface and declare the `clone` method in it. Or just add the method to all classes of an existing class hierarchy, if you have one.

2. A prototype class must define the alternative constructor that accepts an object of that class as an argument. The constructor must copy the values of all fields defined in the class from the passed object into the newly created instance. If you're changing a subclass, you must call the parent constructor to let the superclass handle the cloning of its private fields.

    If your programming language doesn't support method overloading, you may define a special method for copying the object data. The constructor is a more convenient place to do this because it delivers the resulting object right after you call the `new` operator.

3. The cloning method usually consists of just one line: running a `new` operator with the prototypical version of the constructor. Note, that every class must explicitly override the cloning method and use its own class name along with the `new` operator. Otherwise, the cloning method may produce an object of a parent class.

4. Optionally, create a centralized prototype registry to store a catalog of frequently used prototypes.

You can implement the registry as a new factory class or put it in the base prototype class with a static method for fetching the prototype. This method should search for a prototype based on search criteria that the client code passes to the method. The criteria might either be a simple string tag or a complex set of search parameters. After the appropriate prototype is found, the registry should clone it and return the copy to the client.

Finally, replace the direct calls to the subclasses' constructors with calls to the factory method of the prototype registry.

⚖ Pros and Cons

✔ You can clone objects without coupling to their concrete classes.

✔ You can get rid of repeated initialization code in favor of cloning pre-built prototypes.

✔ You can produce complex objects more conveniently.

✔ You get an alternative to inheritance when dealing with configuration presets for complex objects.

✘ Cloning complex objects that have circular references might be very tricky.

⇄ Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicat-
ed and more customizable via subclasses) and evolve toward
Abstract Factory, **Prototype**, or **Builder** (more flexible, but more
complicated).

- **Abstract Factory** classes are often based on a set of **Facto-
ry Methods**, but you can also use **Prototype** to compose the
methods on these classes.

- **Prototype** can help when you need to save copies of **Com-
mands** into history.

- Designs that make heavy use of **Composite** and **Decorator** can
often benefit from using **Prototype**. Applying the pattern lets
you clone complex structures instead of re-constructing them
from scratch.

- **Prototype** isn't based on inheritance, so it doesn't have its
drawbacks. On the other hand, *Prototype* requires a complicat-
ed initialization of the cloned object. **Factory Method** is based
on inheritance but doesn't require an initialization step.

- Sometimes **Prototype** can be a simpler alternative to **Memen-
to**. This works if the object, the state of which you want to
store in the history, is fairly straightforward and doesn't have
links to external resources, or the links are easy to re-estab-
lish.

- **Abstract Factories**, **Builders** and **Prototypes** can all be imple-
mented as **Singletons**.