Factory Method

Definition

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Frequency of use (*in JavaScript*):

1 2 3 4 5
🟥🟥🟥🟥🟨

medium high

Summary

A Factory Method creates new objects as instructed by the client. One way to create objects in JavaScript is by invoking a constructor function with the new operator.

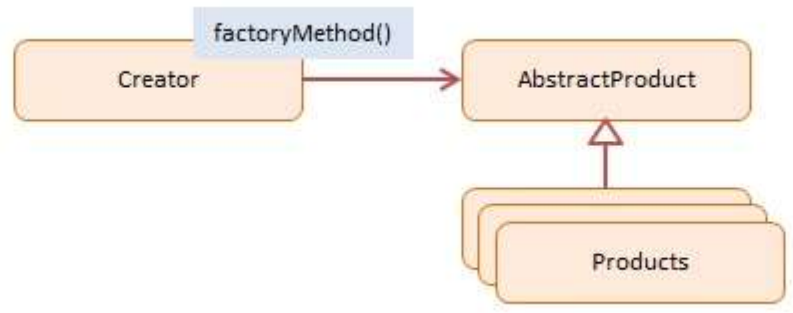
There are situations however, where the client does not, or should not, know which one of several candidate objects to instantiate.

The Factory Method allows the client to delegate object creation while still retaining control over which type to instantiate.

The key objective of the Factory Method is extensibility. Factory Methods are frequently used in applications that manage, maintain, or manipulate collections of objects that are different but at the same time have many characteristics (i.e. methods and properties) in common.

An example would be a collection of documents with a mix of Xml documents, Pdf documents, and Rtf documents.

Diagram



Participants

The objects participating in this pattern are:

- **Creator** -- In sample code: **Factory**
    - the 'factory' object that creates new products
    - implements 'factoryMethod' which returns newly created products
-
- **AbstractProduct** -- not used in JavaScript
    - declares an interface for products
-
- **ConcreteProduct** -- In sample code: **Employees**
    - the product being created
    - all products support the same interface (properties and methods)

Sample code in JavaScript

In this JavaScript example the Factory object creates four different types of employees. Each employee type has a different hourly rate. The createEmployee method is the actual Factory Method. The client instructs the factory what type of employee to create by passing a type argument into the Factory Method.

The AbstractProduct in the diagram is not implemented because Javascript does not support abstract classes or interfaces. However, we still need to ensure that all employee types have the same interface (properties and methods).

Four different employee types are created; all are stored in the same array. Each employee is asked to say what they are and their hourly rate.

The log function is a helper which collects and displays results.

```
function Factory() {
    this.createEmployee = function (type) {
        var employee;

        if (type === "fulltime") {
            employee = new FullTime();
        } else if (type === "parttime") {
            employee = new PartTime();
        } else if (type === "temporary") {
                employee = new Temporary();
            } else if (type === "contractor") {
                employee = new Contractor();
            }

            employee.type = type;

            employee.say = function () {
                log.add(this.type + ": rate " + this.hourly + "/hour");
            }

            return employee;
        }
    }

    var FullTime = function () {
        this.hourly = "$12";
    };

    var PartTime = function () {
        this.hourly = "$11";
    };

    var Temporary = function () {
        this.hourly = "$10";
    };

    var Contractor = function () {
        this.hourly = "$15";
    };

    // log helper
    var log = (function () {
        var log = "";

        return {
            add: function (msg) { log += msg + "\n"; },
            show: function () { alert(log); log = ""; }
        }
    })();

    function run() {
        var employees = [];
        var factory = new Factory();

        employees.push(factory.createEmployee("fulltime"));
        employees.push(factory.createEmployee("parttime"));
        employees.push(factory.createEmployee("temporary"));
        employees.push(factory.createEmployee("contractor"));

        for (var i = 0, len = employees.length; i < len; i++) {
            employees[i].say();
        }

        log.show();
    }
```