# Creational Patterns

---

Overview of creational patterns

- Abstract the instantiation process
- Help make a system independent of how its objects are created, composed, represented

- Class creational pattern
  - ▶ uses inheritance to vary the class that's instantiated
  - ▶ *Factory Method*

- Object creational pattern
  - ▶ delegates instantiation to another object
  - ▶ *Abstract Factory, Prototype, Singleton, Builder*

Let's start simple...
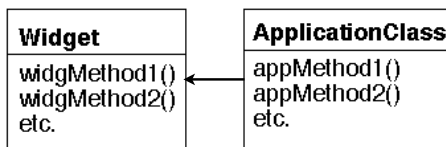

```
class ApplicationClass   {
    Widget a;
    Widget b;

    public appMethod1()   {
        Widget d = new Widget();
        d.widgetMethod1();
        // . . .
        Widget e = new Widget();
        // . . .
    }

    public appMethod2()   {
        // . . .
        Widget f = new Widget();
        f.widgetMethod1();
        // . . .
        Widget g = new Widget();
        // . . .
    }
    // etc. etc . . .
```

We can modify the internal `Widget` code without modifying the `ApplicationClass`

## Problems with Changes

What happens when we discover a new widget and would like to use in the `ApplicationClass`?

Multiple coupling between `Widget` and `ApplicationClass`
- ▶ `ApplicationClass` knows the interface of `Widget`

- ▶ `ApplicationClass` explicitly uses the `Widget` type
  - ◆ hard to change because Widget is a concrete class

- ▶ `ApplicationClass` explicitly creates new Widgets in many places
  - ◆ if we want to use the new `Widget` instead of the initial one, changes are spread all over the code

Apply "Program to an Interface"
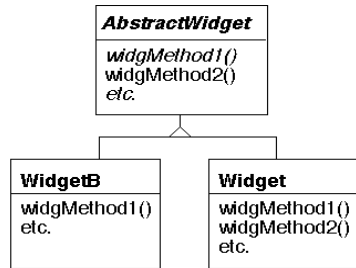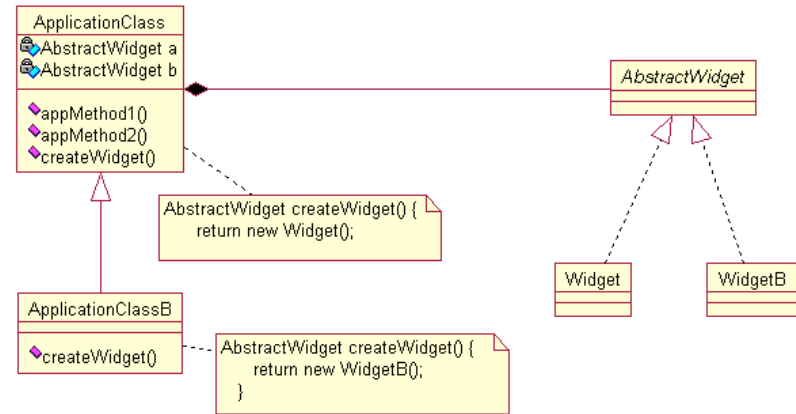


```
class ApplicationClass  {
    AbstractWidget a;
    AbstractWidget b;

    public appMethod1()  {
        AbstractWidget d = new Widget();
        d.widgetMethod1();
        // . . .
        AbstractWidget e = new Widget();
        blah;
    }

  etc.
}
```

- **ApplicationClass** depends now on an (abstract) interface
- But we still have hard coded which widget to create!
  - should I copy-paste? ;-)

Use a Factory Method

```
class ApplicationClass  {
    AbstractWidget a;
    AbstractWidget b;

    // If C++ make this a virtual function
    // and use pointers to ApplicationClass obj.

    public AbstractWidget createWidget() {
        return new Widget();
    }

    public appMethod1()  {
        AbstractWidget d = createWidget();
        d.widgetMethod1();
        // . . .
        AbstractWidget e = createWidget();
        // . . .
    }
    // . . .
}
class ApplicationClassB extends ApplicationClass {
    public AbstractWidget createWidget() {
        return new WidgetB();
    }
}
```
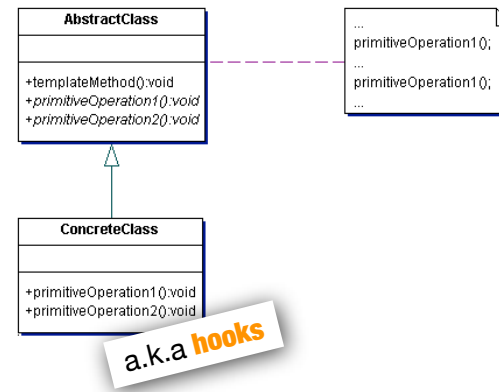
Elsewhere ...

```
ApplicationClass test = new ApplicationClassB();
test.appMethod1();
```

Factory Method is a Creational **Template Method**...


a.k.a **hooks**

Evaluation of Factory Method Solution

- Explicit creation of **Widget** objects is not anymore dispersed
  - ‣ easier to change

- Functional methods in **ApplicationClass** are decoupled from various concrete implementations of widgets

- Avoid ugly code duplication in **ApplicationClassB**
  - ‣ subclasses reuse the functional methods, just implementing the concrete Factory Method needed

- Disadvantages
  - ‣ create a subclass only to override the factory-method
  - ‣ can't change the **Widget** at run-time

Factory Method

Basic Aspects

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

- Also Known As
  - ‣ Virtual Constructor

- Applicability
  - ‣ A class can't anticipate the class of objects it must create
  - ‣ A class wants its subclasses to specify the objects it creates
  - ‣ Classes delegate responsibility to one of several helper subclasses

---

Structure
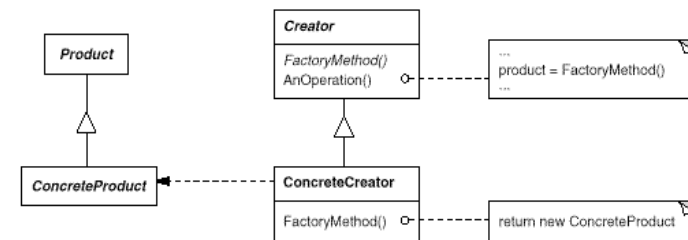
Participants & Collaborations

- Product
  - ‣ defines the interface of objects that will be created by the FM
  - ‣ Concrete Product implements the interface

- Creator
  - ‣ declares the FM, which returns a product of type Product.
    - ◆ may define a default implementation of the FM
    - ◆ may call the FM to create a product

- ConcreteCreator
  - ‣ overrides FM to provide an instance of ConcreteProduct

Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct

---

Consequences

- Eliminate binding of application specific classes into your code.
  - ‣ creational code only deals with the Product interface

- Provide hooks for subclassing
  - ‣ subclasses can change this way the product that is created

- Clients might have to subclass the Creator just to create a particular ConcreteProduct object.

Implementation Issues

- Varieties of Factory Methods
  - ‣ Creator class is **abstract**
    - ◆ does not provide an implementation for the FM it declares
    - ◆ requires subclasses
  - ‣ Creator is a **concrete** class
    - ◆ provides default implementation
    - ◆ FM used for flexibility
    - ◆ Create objects in a separate operation so that subclasses can override it
- Parametrization of Factory Methods
  - ‣ A variation on the pattern lets the factory method create multiple kinds of products
  - ‣ a parameter identifies the type of Product to create
  - ‣ all created objects share the Product interface

Parameterizing the Factory

```
class Creator {
    public Product create(productId) {
      if (id == MINE) return new MyProduct;
      if (id == YOURS) return new YourProduct;
    }
}

class MyCreator extends Creator {
  public Product create(productId) {
    if (id == MINE) return new YourProduct;
    if (id == YOURS) return new MyProduct;
    if (id == THEIRS) return new TheirProduct;
    return super.create(id); // called if others fail
}
```

- selectively extend or change products that get created

Java: `forName` and Factory Methods

```java
class Creator {
    public Product FactoryMethod(String productType) {
        Class productClass = Class.forName(productType);
        return (Product) productClass.newInstance();
    }
}


Product theBest = new Creator().FactoryMethod("ProductA");
```

```java
import java.util.*;

class AbstractFactory {
    public Product make(String c) {
      try {
          Class prod = Class.forName(c);
          return (Product) prod.newInstance();
      }
      catch(Exception e) {
          System.out.println("Error");
          System.exit(1);
          return null;
      }
    }
}

class Main {
    public static void main(String[] args) {
      AbstractFactory af = new AbstractFactory();

      af.make(args[0]).doSomething();
    }
}
```

```java
abstract class Product {
    abstract public void doSomething();
}

class ProductA extends Product {
    public void doSomething() {
      System.out.println("ProductA");
    }
}

class ProductB extends Product {
    public void doSomething() {
      System.out.println("ProductB");
    }
}
```
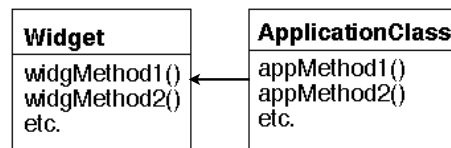
Another solution for the example:
Factory Method in Product

Widget	ApplicationClass
widgMethod1() widgMethod2() etc.	appMethod1() appMethod2() etc.

Solution 2.1: Product Creates Itself

Provide the **Widgets** with a polymorphic creational method

AbstractWidget

create()
widgMethod1()
widgMethod2()
etc.

WidgetB

create()
widgMethod1()
etc.

Widget

create()
widgMethod1()
widgMethod2()
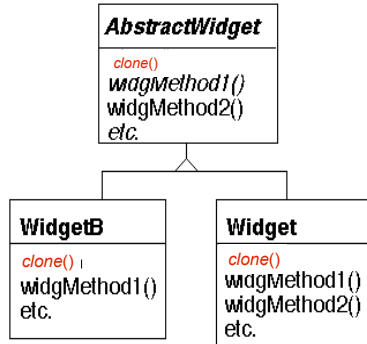etc.
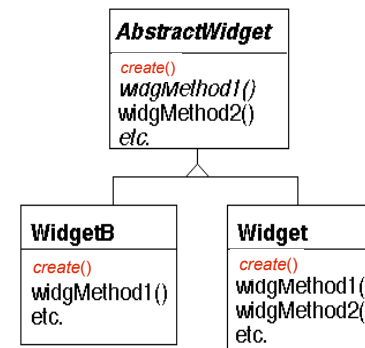
```java
class Widget {
    int data;

    public WidgetB create() {
      return new WidgetB();
    }
}
```

Solution 2.2: Product Clones Itself

Provide the **Widgets** with a clone method
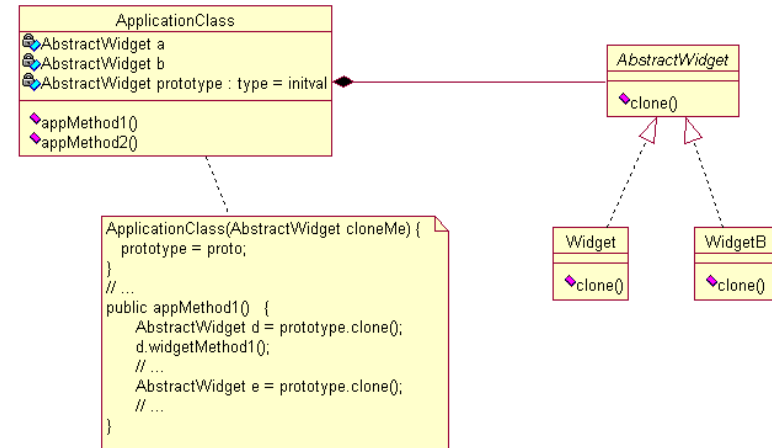
▸ make a copy of an existing Widget object


```
class Widget {
    int data;

    public Widget clone() {
        Widget aCopy = new Widget();
        aCopy.data = data;
        return aCopy;
    }
}
```

Using the Clone

```
class ApplicationClass   {
    AbstractWidget a;
    AbstractWidget b;
    AbstractWidget prototype;

    public ApplicationClass(AbstractWidget cloneMe ) {
        prototype = cloneMe;
    }

    public appMethod1()   {
        AbstractWidget d = prototype.clone();
        d.widgetMethod1();
        // . . .
        AbstractWidget e = prototype.clone();
        // . . .
    }

    // ...etc. etc...
}
```

Elsewhere:
```
ApplicationClass test =
    new ApplicationClass( new Widget() );

ApplicationClass testB =
    new ApplicationClass( new WidgetB() );
```


Advantages

- Classes to instantiate may be specified dynamically
  - ▸ client can install and remove prototypes at run-time

- We avoided subclassing of **ApplicationClass**
  - ▸ **Remember**: Favor Composition over Inheritance! :-)

- Totally hides concrete product classes from clients
  - ▸ Reduces implementation dependencies

The Prototype Pattern

Basic Aspects

- Intent
  - ‣ Specify the kinds of objects to create using a prototypical instance
  - ‣ Create new objects by copying this prototype

- Applicability
  - ‣ when a client class should be independent of how its products are created, composed, and represented **and**
  - ‣ when the classes to instantiate are specified at run-time

---

Structure
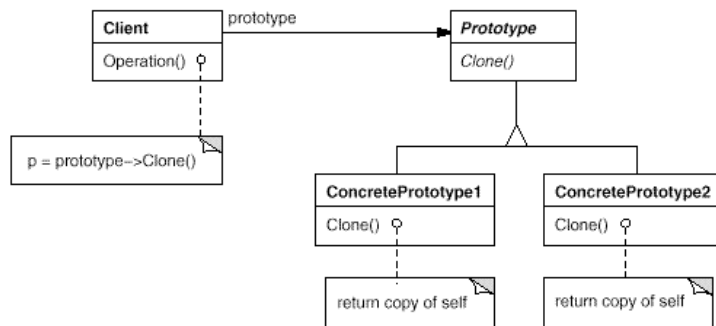
Consequences

- Adding and removing products at run-time
- Reduced subclassing
  - ‣ avoid parallel hierarchy for creators

- Each subclass of Prototype must implement `clone`
  - ‣ difficult when classes already exist or
 - ‣ internal objects don't support copying or have circular references

Implementation Issues

- Using a Prototype manager
  - number of prototypes isn't fixed
    - keep a registry → **prototype manager**
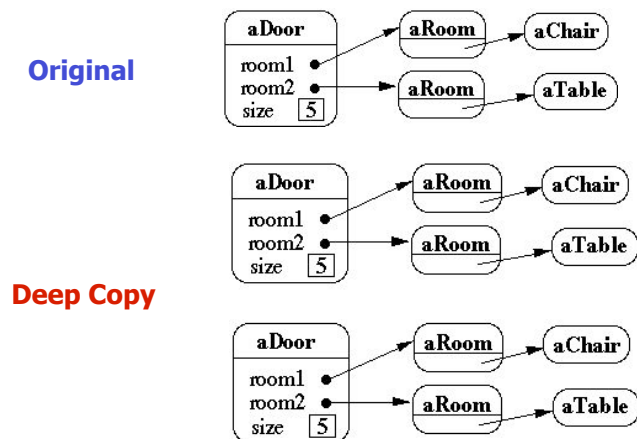 - clients instead of knowing the prototype know a manager
    - associative store

- Initializing clones
  - heterogeneity of initialization methods
  - write an **Initialize** method

- Implementing the **clone** operation
  - shallow vs. deep copy

Shallow Copy vs. Deep Copy

Original
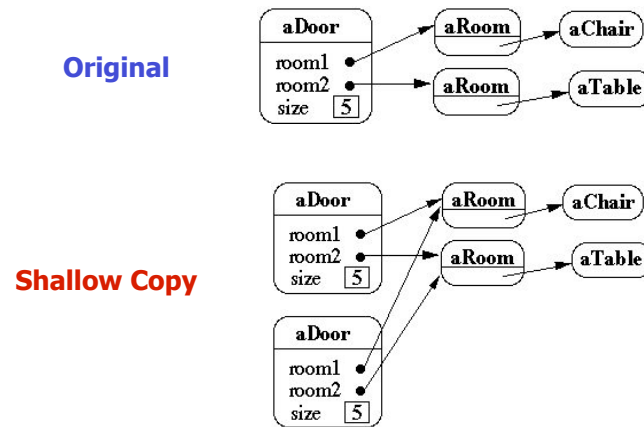
Shallow Copy

Shallow Copy vs. Deep Copy (2)

Original

Deep Copy

Cloning in Java – **Object clone()**

protected Object clone() throws CloneNotSupportedException
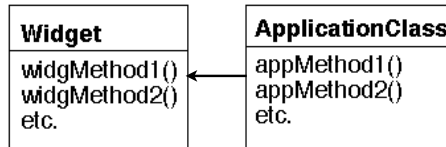
- Creates a clone of the object.
  - allocate a new instance and,
  - place a *bitwise clone* of the current object in the new object.

```
class Device implements Cloneable  {
  public void Initialize( Widget a, Widget b) {
    w1 = a; w2 = b;
  }

  public Object clone() throws CloneNotSupportedException {
        return super.clone();
  }
  Widget w1, w2;
}
```
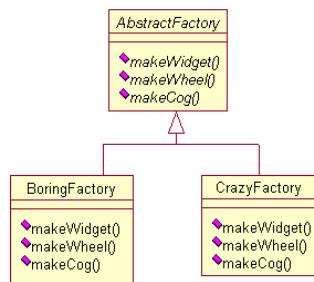
Slide 202

More Changes...
What if ApplicationClass uses other "products" too...Wheels, Cogs, etc.

```
Widget                  ApplicationClass
widgMethod1()    ◄───   appMethod1()
widgMethod2()           appMethod2()
etc.                    etc.
```

Slide 203

More Changes

- Each one of these stays for an object family
  - i.e. all of these have subclasses

- Assume that there are restrictions on what type of Widget can be used with which type of Wheel or Cog

- Factory Methods or Prototypes can handle each type of product but it get hard to insure the wrong types of items are not used together

Slide 204

Solution: Create an Abstract Factory

```
AbstractFactory
◆makeWidget()
◆makeWheel()
◆makeCog()
       △
   ┌───┴───┐
BoringFactory   CrazyFactory
◆makeWidget()   ◆makeWidget()
◆makeWheel()    ◆makeWheel()
◆makeCog()      ◆makeCog()
```

```java
class ApplicationClass   {
    AbstractWidget a;
    AbstractCog b;
    AbstractFactory partFactory;

    public ApplicationClass(AbstractFactory aFactory)
    {
        partFactory = aFactory;
    }

    public appMethod1()
    {
        AbstractWidget d = partFactory.makeWidget();
        d.widgetMethod1();
        // ...
        AbstractCog e = partFactory.makeCog();
    } // ....
}
```

Slide 205

Abstract Factory

Basic Aspects

- Intent
  - Provide an interface for creating **families of related or dependent objects** without specifying their concrete classes

- Applicability
  - System should be independent of how its products are created, composed and represented
  - System should be configured with one of multiple families of products
  - Need to **enforce** that a family of product objects is used together

---

Structure
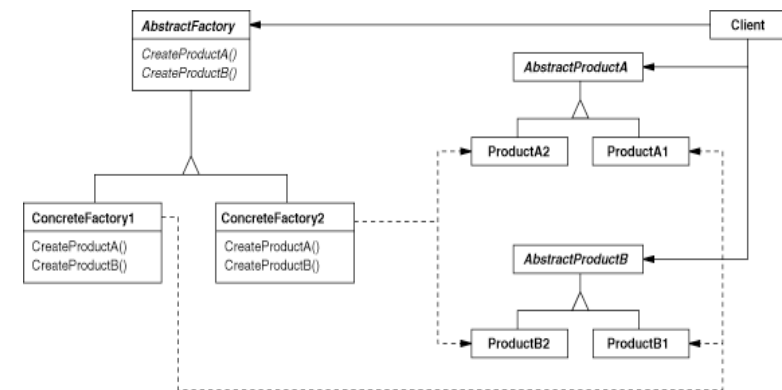
Participants & Collaborations

- Abstract Factory
  - declares an interface for operations to create abstract products
- ConcreteFactory
  - implements the operations to create products
- AbstractProduct
  - declares an interface for a type of product objects
- ConcreteProduct
  - declares an interface for a type of product objects
- Client
  - uses only interfaces decl. by AbstractFactory and AbstractProduct

- A single instance of a ConcreteFactory created.
  - create products having a particular implementation

Consequences

- Isolation of concrete classes
  - appear in `ConcreteFactories` not in client's code

- Exchanging of product families becomes easy
  - a `ConcreteFactory` appears only in one place
    - easy to change

- Promotes consistency among products
  - all products in a family change **at once**, and change **together**

- Supporting new kinds of products is difficult
  - requires a change in the interface of AbstractFactory
  - ... and consequently all subclasses

Implementation Issues

- Factories as Singletons
  - ▸ to assure that only one **ConcreteFactory** per product family is created

- Creating the Products
  - ▸ collection of Factory Methods
  - ▸ can be also implemented using Prototype
    - ◆ define a prototypical instance for each product in **ConcreteFactory**

- Defining Extensible Factories
  - ▸ a single factory method with parameters
  - ▸ more flexible, less safe!

Creating Products...

- ...using own factory methods

```
abstract class WidgetFactory {
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
}

            class MacWidgetFactory extends WidgetFactory {
                public Window createWindow()
                    { return new MacWindow() }
                public Menu createMenu()
                    { return new MacMenu() }
                public Button createButton()
                    { return new MacButton()  }
            }
```

Creating Products...

- ... using product's factory methods
  - ▸ subclass just provides the concrete products in the constructor
  - ▸ spares the re-implementation of FM's in subclasses

```
abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    protected WidgetFactory(Window w, Menu m, Button b) {
        windowFactory = w; menuFactory = m; buttonFactory = b;
    }
    public Window createWindow()
        { return windowFactory.createWindow() }
    public Menu createMenu();
        { return menuFactory.createMenu() }
    public Button createButton()
        { return buttonFactory.createButton() }
}
            class MacWidgetFactory extends WidgetFactory {
                public MacWidgetFactory() {
                    super(new MacWindow(), new MacMenu(), new MacButton();
                }
            }
```
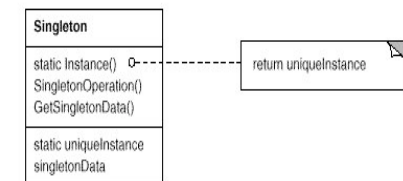

Singletons

```
public class Singleton {
    protected Singleton() {
        // ... }

    static private Singleton _instance = null;

    static public Singleton instance() {
        if(null == _instance) {
            _instance = new Singleton();
        }
        return _instance;
    }

}
```


Singleton
static Instance()   →   return uniqueInstance
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData

Why Use Singletons?

- Controlled access to sole instance

- Permits refinement of operations and representation

- Permits a variable (but precise) number of instances

One Singleton for many Instances ;-)

```java
public interface SingletonFactoryMethod {
    public Singleton createInstance();
}


public class SingletonWrapper {
    static private SingletonFactoryMethod _factory = null;

    static private Singleton _instance = null;

    static public Singleton instance() {
        if(null == _instance)
            if(null == _factory) _instance = new Singleton();
            else _instance = _factory.createInstance();
        }
        return _instance;
    }

    static public void setFactory(SingletonFactoryMethod factory) {
        _factory = factory; _instance = null;}
}
```