# Builder Design Pattern

Builder pattern is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

[](http://javapapers.com/wp-content/uploads/2009/11/ComplexObject.jpg)

Complex Object Construction

For example, you can consider construction of a home. Home is the final end product (object) that is to be returned as the output of the construction process. It will have many steps, like basement construction, wall construction and so on roof construction. Finally, the whole home object is returned. Here using the same process, you can build houses with different properties.

GOF says,

*“Separate the construction of a complex object from its representation so that the same construction process can create different representations” [GoF 94]*

**What is the difference between abstract factory and builder pattern?**

[Abstract factory](http://javapapers.com/design-patterns/abstract-factory-pattern/) may also be used to construct a complex object, then what is the difference with builder pattern? In builder pattern emphasis is on ‘step by step’. Builder pattern will have many number of small steps. Every steps will have small units of logic enclosed in it. There will also be a sequence involved. It will start from step 1 and will go on up to step n and the final step is returning the object. In these steps, every step will add some value in construction of the object. That is, you can imagine that the object grows stage by stage. Builder will return the object in last step. But in abstract factory how complex the built object might be, it will not have step by step object construction.

Ads by Google

**Sample builder design pattern implementation in Java API**

[DocumentBuilderFactory](http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/parsers/DocumentBuilderFactory.html) , StringBuffer, StringBuilder are some examples of builder pattern usage in java API.

**Sample Java Source Code for Builder Pattern**

Following is the interface, that will be returned as the product from the builder.

|  |
| --- |
| package com.javapapers.sample.designpattern.builder;    public interface HousePlan {      public void setBasement(String basement);      public void setStructure(String structure);      public void setRoof(String roof);      public void setInterior(String interior);  } |

Following is the interface for which the factory implementation should be done. In turn all abstract factory will return this type.

|  |
| --- |
| package com.javapapers.sample.designpattern.abstractfactory;    public interface AnimalFactory {    public Animal createAnimal();  } |

Concrete class for the above interface. The builder constructs an implementation for the following class.

|  |
| --- |
| package com.javapapers.sample.designpattern.builder;    public class House implements HousePlan {      private String basement;    private String structure;    private String roof;    private String interior;      public void setBasement(String basement) {      this.basement = basement;    }      public void setStructure(String structure) {      this.structure = structure;    }      public void setRoof(String roof) {      this.roof = roof;    }      public void setInterior(String interior) {      this.interior = interior;      }    } |

Builder interface. We will have multiple different implementation of this interface in order to facilitate, the same construction process to create different representations.

|  |
| --- |
| package com.javapapers.sample.designpattern.builder;    public interface HouseBuilder {      public void buildBasement();      public void buildStructure();      public void bulidRoof();      public void buildInterior();      public House getHouse();  } |

First implementation of a builder.

|  |
| --- |
| package com.javapapers.sample.designpattern.builder;    public class IglooHouseBuilder implements HouseBuilder {      private House house;      public IglooHouseBuilder() {      this.house = new House();    }      public void buildBasement() {      house.setBasement("Ice Bars");    }      public void buildStructure() {      house.setStructure("Ice Blocks");    }      public void buildInterior() {      house.setInterior("Ice Carvings");    }      public void bulidRoof() {      house.setRoof("Ice Dome");    }      public House getHouse() {      return this.house;    }  } |

Second implementation of a builder. Tipi is a type of eskimo house.

|  |
| --- |
| package com.javapapers.sample.designpattern.builder;    public class TipiHouseBuilder implements HouseBuilder {    private House house;      public TipiHouseBuilder() {      this.house = new House();    }      public void buildBasement() {      house.setBasement("Wooden Poles");    }      public void buildStructure() {      house.setStructure("Wood and Ice");    }      public void buildInterior() {      house.setInterior("Fire Wood");    }      public void bulidRoof() {      house.setRoof("Wood, caribou and seal skins");    }      public House getHouse() {      return this.house;    }    } |

Following class constructs the house and most importantly, this maintains the building sequence of object.

|  |
| --- |
| package com.javapapers.sample.designpattern.builder;    public class CivilEngineer {      private HouseBuilder houseBuilder;      public CivilEngineer(HouseBuilder houseBuilder){      this.houseBuilder = houseBuilder;    }      public House getHouse() {      return this.houseBuilder.getHouse();    }      public void constructHouse() {      this.houseBuilder.buildBasement();      this.houseBuilder.buildStructure();      this.houseBuilder.bulidRoof();      this.houseBuilder.buildInterior();    }  } |

Testing the sample builder design pattern.

|  |
| --- |
| package com.javapapers.sample.designpattern.builder;    public class BuilderSample {    public static void main(String[] args) {      HouseBuilder iglooBuilder = new IglooHouseBuilder();      CivilEngineer engineer = new CivilEngineer(iglooBuilder);        engineer.constructHouse();        House house = engineer.getHouse();        System.out.println("Builder constructed: "+house);    }  } |

**Output of the above sample program for builder pattern**

|  |
| --- |
| Builder constructed: com.javapapers.sample.designpattern.builder.House@7d772e |

# Factory Method Design Pattern

05/11/2009

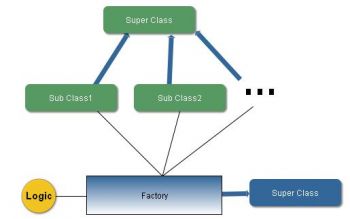
A factory method pattern is a creational pattern. It is used to instantiate an object from one among a set of classes based on a logic.

Assume that you have a set of classes which extends a common super class or interface. Now you will create a concrete class with a method which accepts one or more arguments. This method is our factory method. What it does is, based on the arguments passed factory method does logical operations and decides on which sub class to instantiate. This factory method will have the super class as its return type. So that, you can program for the interface and not for the implementation. This is all about factory method design pattern.

## Sample factory method design pattern implementation in Java API

For a reference of how the factory method design pattern is implemented in Java, you can have a look at SAXParserFactory. It is a factory class which can be used to intantiate SAX based parsers to pares XML. The method [newInstance](http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/parsers/SAXParserFactory.html#newInstance%28%29) is the factory method which instantiates the sax parsers based on some predefined logic.

### Block diagram for The Design Pattern



## Sample Java Source Code for Factory Method Design Pattern

Based on comments received from users, I try to keep my sample java source code as simple as possible for a novice to understand.

Base class:

|  |
| --- |
| package com.javapapers.sample.designpattern.factorymethod;    //super class that serves as type to be instantiated for factory method pattern  public interface Pet {     public String speak();    } |

First subclass:

|  |
| --- |
| package com.javapapers.sample.designpattern.factorymethod;    //sub class 1 that might get instantiated by a factory method pattern  public class Dog implements Pet {     public String speak() {   return "Bark bark...";   }  } |

Ads by Google

Second subclass:

|  |
| --- |
| package com.javapapers.sample.designpattern.factorymethod;    //sub class 2 that might get instantiated by a factory method pattern  public class Duck implements Pet {   public String speak() {   return "Quack quack...";   }  } |

Factory class:

|  |
| --- |
| package com.javapapers.sample.designpattern.factorymethod;    //Factory method pattern implementation that instantiates objects based on logic  public class PetFactory {     public Pet getPet(String petType) {   Pet pet = null;     // based on logic factory instantiates an object   if ("bark".equals(petType))   pet = new Dog();   else if ("quack".equals(petType))   pet = new Duck();   return pet;   }  } |

Using the factory method to instantiate

|  |
| --- |
| package com.javapapers.sample.designpattern.factorymethod;    //using the factory method pattern  public class SampleFactoryMethod {     public static void main(String args[]){     //creating the factory   PetFactory petFactory = new PetFactory();     //factory instantiates an object   Pet pet = petFactory.getPet("bark");     //you don't know which object factory created   System.out.println(pet.speak());   }    } |

### Output of the above sample program for Factory Method Pattern

|  |
| --- |
| Bark bark |