Overview of GOF Design Patterns - Core Design Patterns

CHAPTER 2

PRESENTATION BY:
RMN

spring

Chapter Two

We will cover these skills

- Introducing the power of design patterns
- Common GOF Design Patterns overview
- Using the power of the POJO pattern
- Core design patterns
  - Creational design patterns
  - Structural design patterns
  - Behavioral design patterns
- J2EE design patterns
  - Design patterns at presentation layer
  - Design patterns at business layer
  - Design patterns at integration layer
- Some best practices for Spring application development

Introducing
the Power of Design Patterns

A design pattern is a software engineering concept describing recurring solutions to common problems in software design.

Design patterns also represent the best practices used by experienced object-oriented software developers."

A design pattern has three main characteristics:

- A Design pattern is specific to a particular scenario rather than a specific platform.
- Design patterns have been evolved to provide the best solutions to certain problems faced during software development.
- Design patterns are the remedy for the problems under consideration.

Common GoF
Design Pattern Overview

The Gang of Four (GoF:
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides)
patterns are 23 classic software design patterns providing recurring solutions to common problems in software design.

These patterns are categorized into two main categories:

- Core Design Patterns
- J2EE Design Patterns
  - Design pattern at the **presentation layer**
  - Design pattern at the **business layer**
  - Design pattern at the **integration layer**

Categories of
Core Design Patterns

Creational Design Pattern: Patterns under this category provide a way to construct objects when constructors will not serve your purpose.

Structural Design Pattern: Patterns under this category deal with the composition of classes or objects.

Behavioral Design Pattern: Patterns under this category, characterize the ways in which classes or objects interact and distribute responsibility

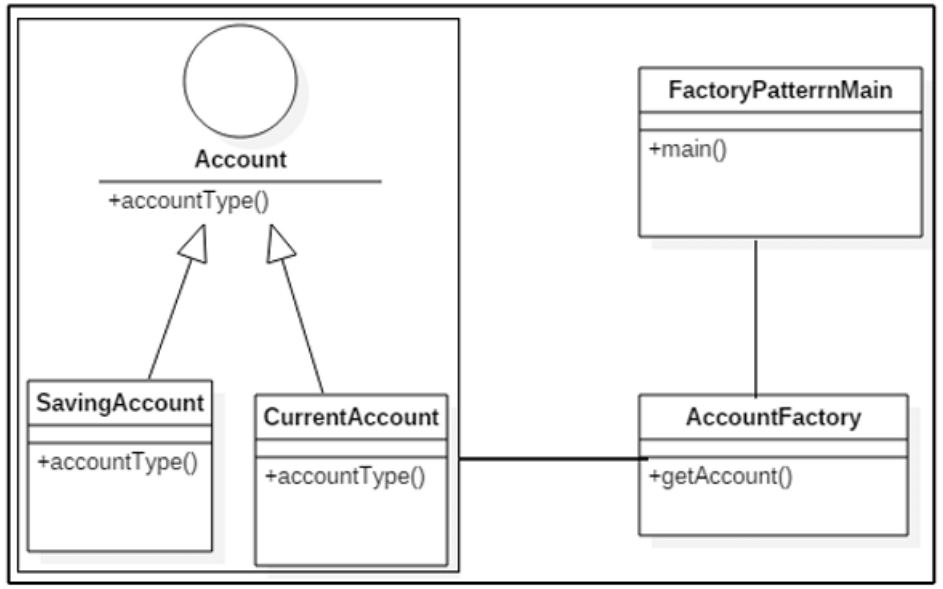
Factory
Design Pattern

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory Method lets a class defer instantiation to subclasses.

According to this design pattern, you get an object of a class without exposing the underlying logic to the client.

The Factory pattern is one of the most-used design patterns in Java.



Factory
Design Pattern (Contd.)

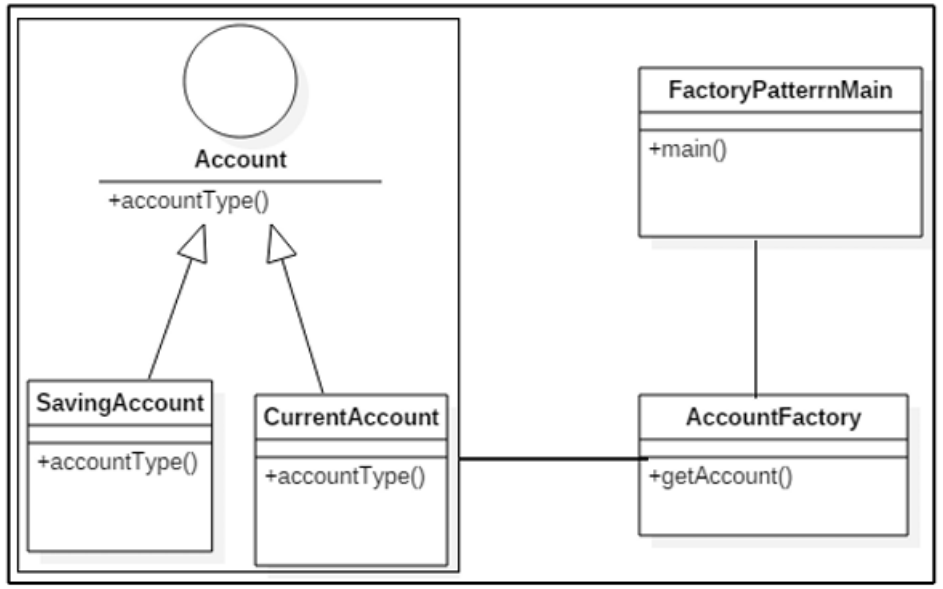
```
package com.packt.patterninspring.chapter2.factory;
public interface Account {
  void accountType();
}
```

Now let's create SavingAccount.java, which will implement the Account interface:

```
package com.packt.patterninspring.chapter2.factory;
public class SavingAccount implements Account{
@Override
  public void accountType() {
    System.out.println("SAVING ACCOUNT");
  }
}
```

Same with CurrentAccount.java, it will also implement the Account interface:

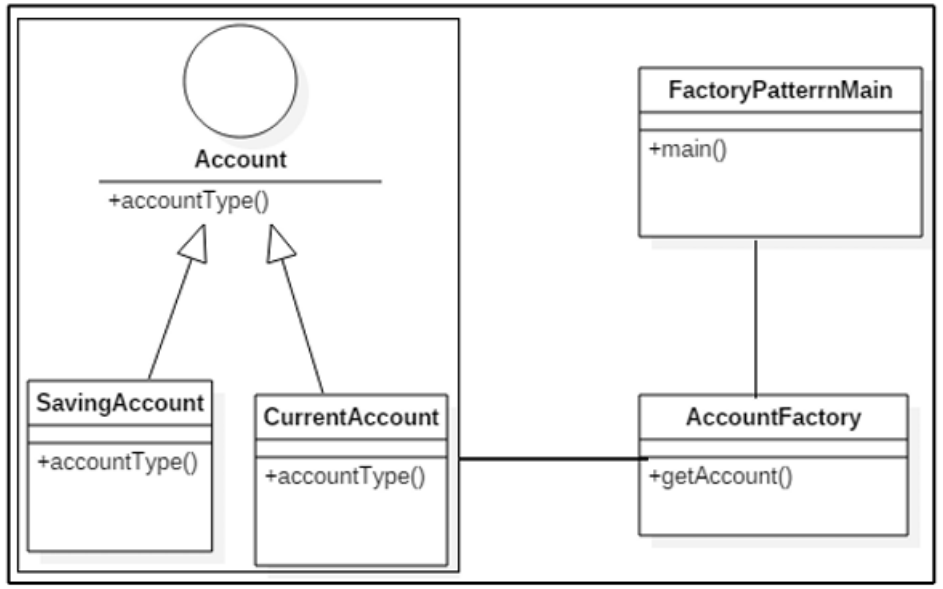
```
package com.packt.patterninspring.chapter2.factory;
public class CurrentAccount implements Account {
  @Override
  public void accountType() {
    System.out.println("CURRENT ACCOUNT");
  }
}
```



Factory
Design Pattern (Contd.)

AccountFactory.java is a Factory to produce the Account type object:

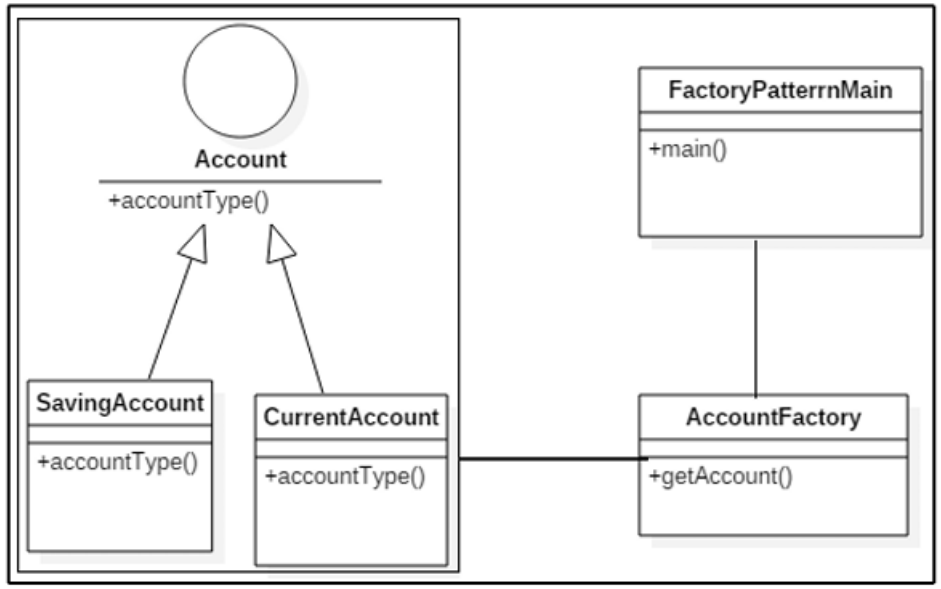
```
package com.packt.patterninspring.chapter2.factory.pattern;
import com.packt.patterninspring.chapter2.factory.Account;
import com.packt.patterninspring.chapter2.factory.CurrentAccount;
import com.packt.patterninspring.chapter2.factory.SavingAccount;
public class AccountFactory {
  final String CURRENT_ACCOUNT = "CURRENT";
  final String SAVING_ACCOUNT = "SAVING";
  //use getAccount method to get object of type Account
  //It is factory method for object of type Account
  public Account getAccount(String accountType){
    if(CURRENT_ACCOUNT.equals(accountType)) {
      return new CurrentAccount();
    }
    else if(SAVING_ACCOUNT.equals(accountType)){
      return new SavingAccount();
    }
    return null;
  }
}
```



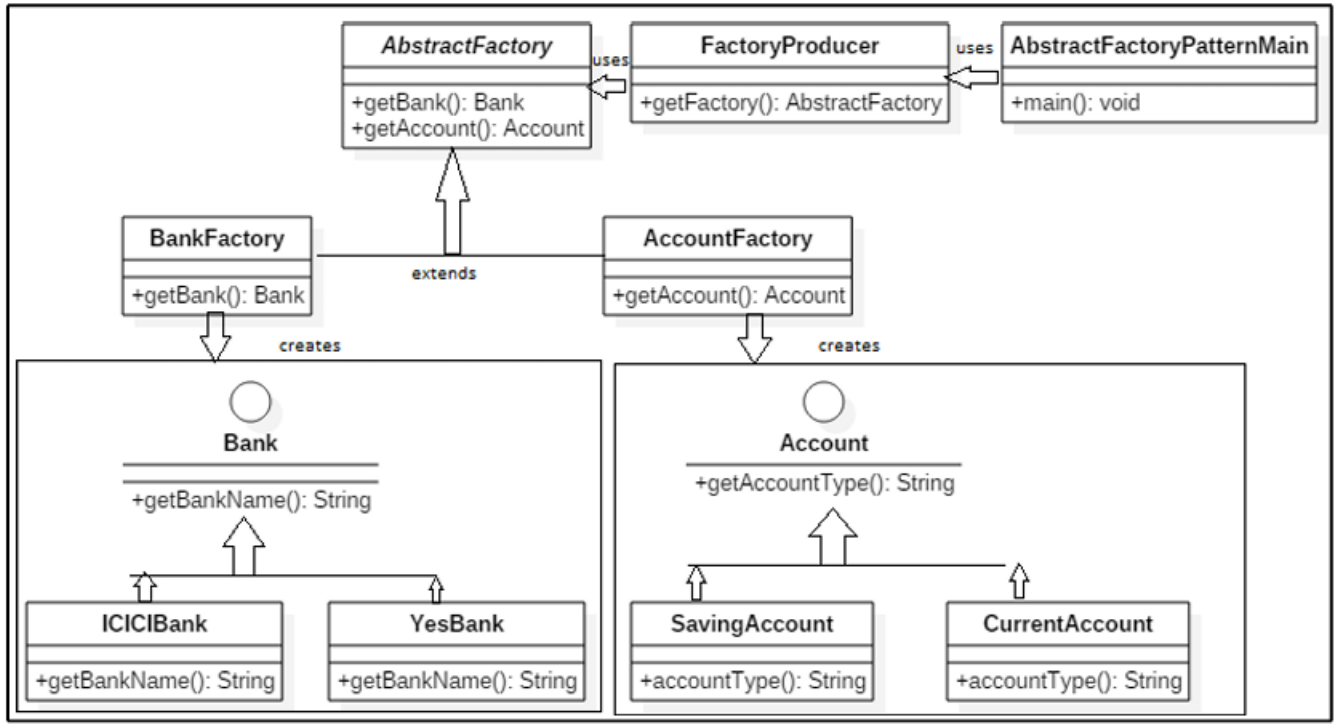
Factory
Design Pattern (Contd.)

Let's create a demo class FactoryPatterMain.java to test the factory method design pattern:

```
package com.packt.patterninspring.chapter2.factory.pattern;
import com.packt.patterninspring.chapter2.factory.Account;
public class FactoryPatterMain {
  public static void main(String[] args) {
    AccountFactory accountFactory = new AccountFactory();
    //get an object of SavingAccount and call its accountType()
method.
    Account savingAccount = accountFactory.getAccount("SAVING");
    //call accountType method of SavingAccount
    savingAccount.accountType();
    //get an object of CurrentAccount and call its
accountType()method.
    Account currentAccount = accountFactory.getAccount("CURRENT");
    //call accountType method of CurrentAccount
    currentAccount.accountType();
  }
}
```



Abstract
Design Pattern



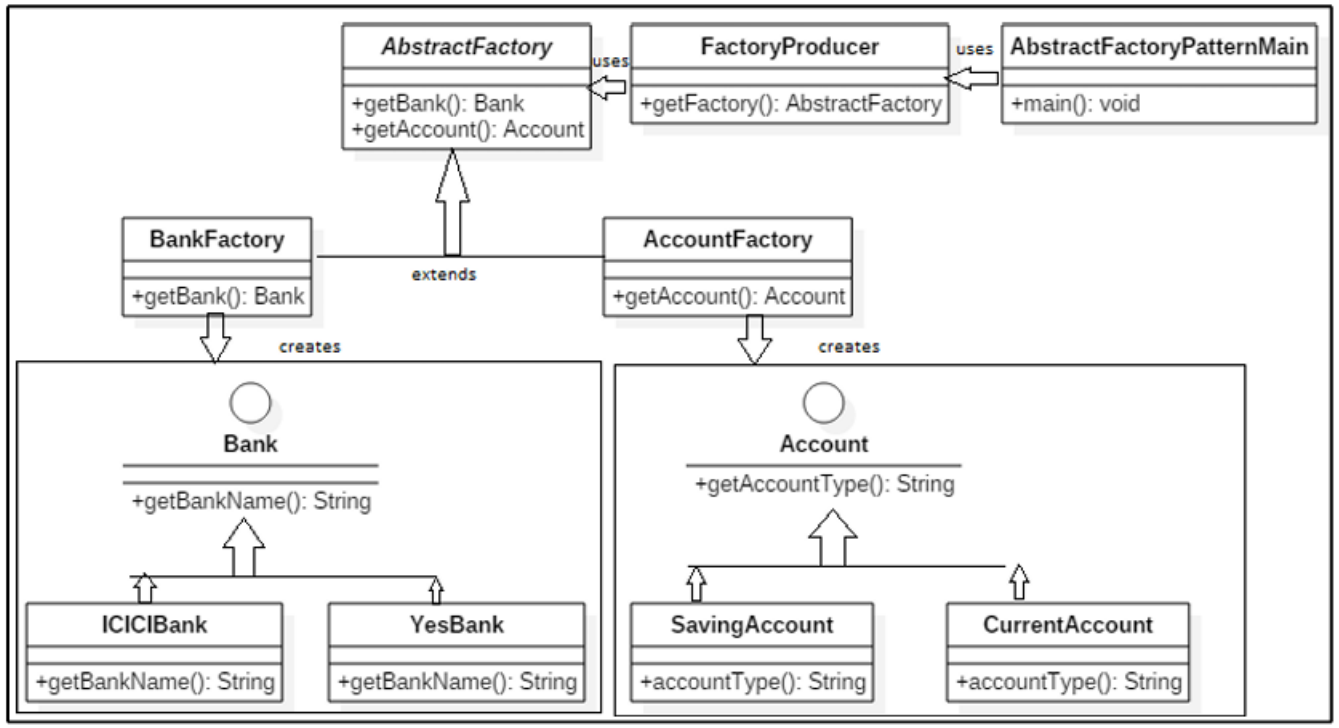
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

It is a high-level design pattern compared to the factory method design pattern.

In the Abstract Factory pattern, an interface is responsible for creating a factory of related objects without explicitly specifying their classes.

```
package
com.packt.patterninspring.chapter2.model;
public interface Bank {
  void bankName();
}
```

Abstract
Design Pattern (Contd.)



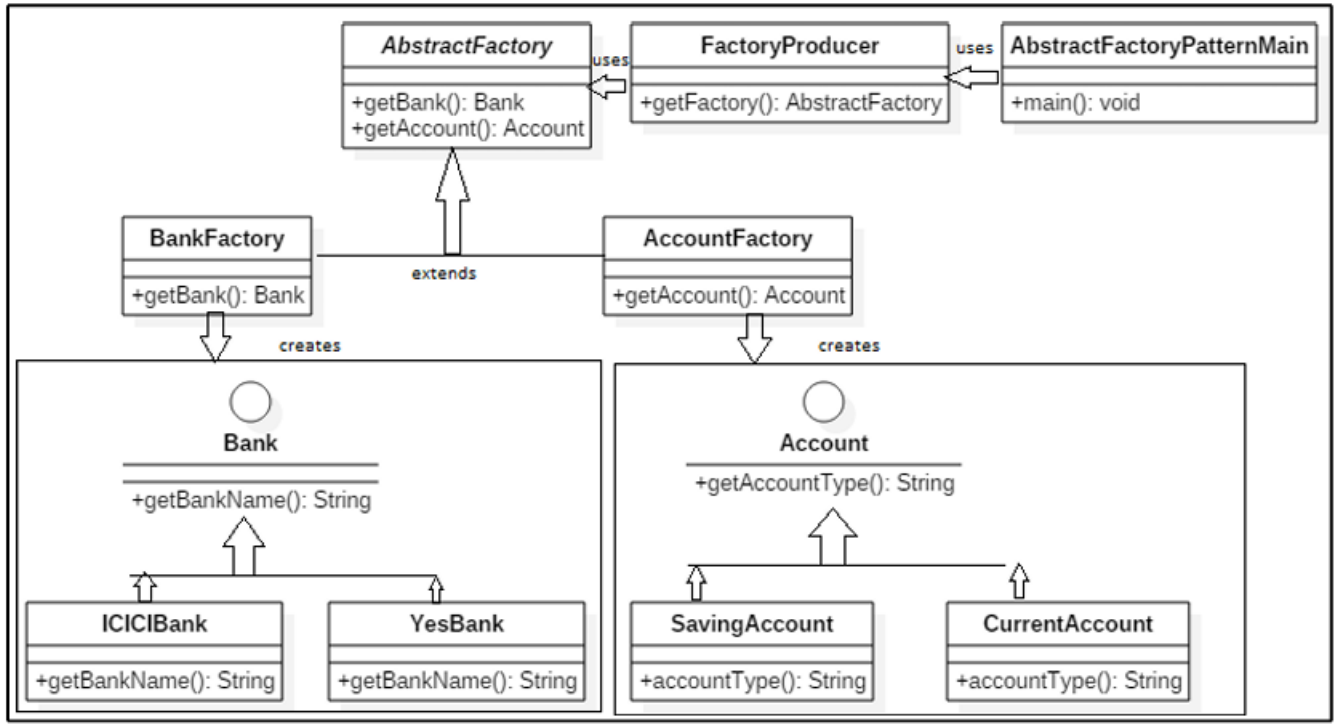
Now let's create ICICIBank.java, which implements the Bank interface:

```
package
com.packt.patterninspring.chapter2.model;
public class ICICIBank implements Bank {
  @Override public void bankName() {
    System.out.println("ICICI Bank Ltd.");
  }
}
```

Let's create another YesBank.java, an implementing Bank interface:

```
package
com.packt.patterninspring.chapter2.model;
public class YesBank implements Bank {
  @Override public void bankName() {
    System.out.println("Yes Bank Pvt. Ltd.");
  }
}
```

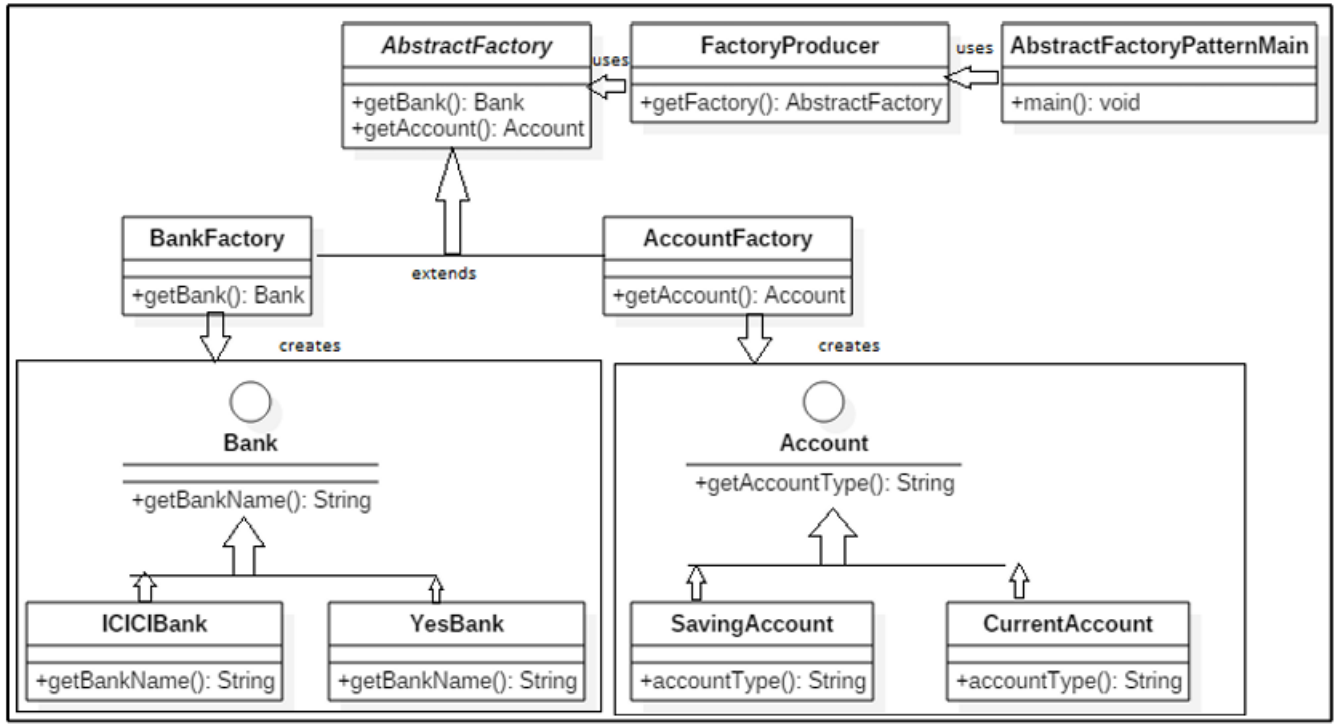
Abstract
Design Pattern (Contd.)



AbstractFactory.java is an abstract class that is used to get factories for Bank and Account objects:

```
package
com.packt.patterninspring.chapter2.abstractfactory.
pattern;
import
com.packt.patterninspring.chapter2.model.Account;
import
com.packt.patterninspring.chapter2.model.Bank;
public abstract class AbstractFactory {
  abstract Bank getBank(String bankName);
  abstract Account getAccount(String accountType);
}
```

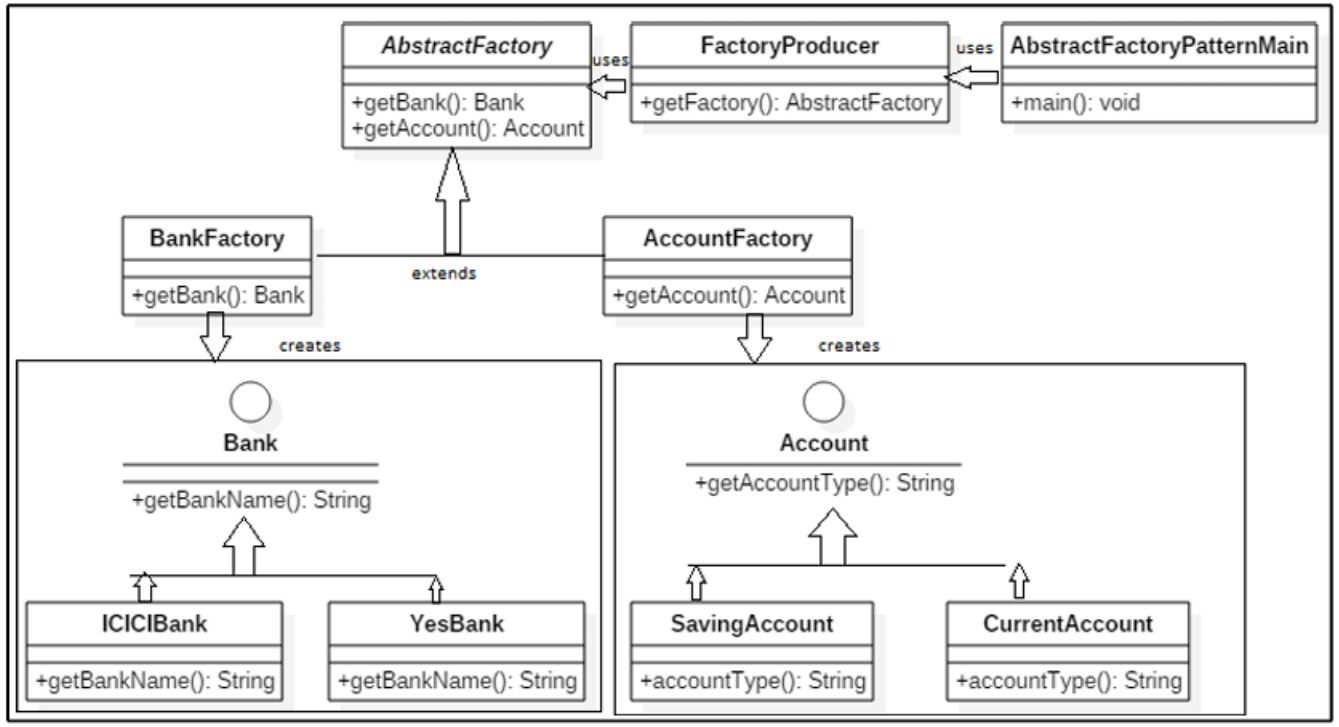
Abstract
Design Pattern (Contd.)



BankFactory.java is a factory class extending AbstractFactory to generate an object of the concrete class based on the given information:

```
package
com.packt.patterninspring.chapter2.abstractfactory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.Bank;
import
com.packt.patterninspring.chapter2.model.CurrentAccount;
import
com.packt.patterninspring.chapter2.model.SavingAccount;
public class AccountFactory extends AbstractFactory {
  final String CURRENT_ACCOUNT = "CURRENT";
  final String SAVING_ACCOUNT = "SAVING";
  @Override Bank getBank(String bankName) {
    return null;
  }
  //use getAccount method to get object of type
  Account
  //It is factory method for object of type Account
  @Override public Account getAccount(String accountType){
    if(CURRENT_ACCOUNT.equals(accountType)) {
      return new CurrentAccount();
    } else if(SAVING_ACCOUNT.equals(accountType)) {
      return new SavingAccount();
    }
    return null;
  }
}
```

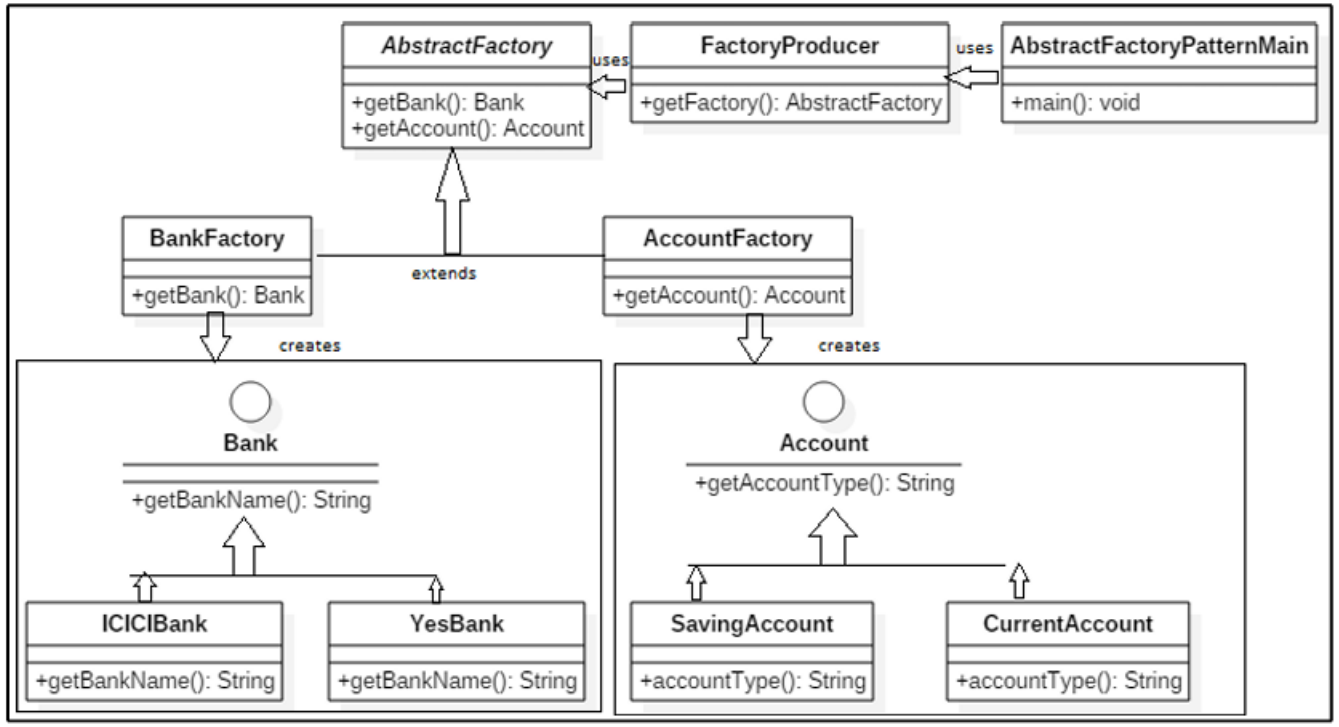
Abstract
Design Pattern (Contd.)



AccountFactory.java is a factory class that extends AbstractFactory.java to generate an object of the concrete class based on the given information:

```
package
com.packt.patterninspring.chapter2.abstractfactory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.Bank;
import
com.packt.patterninspring.chapter2.model.CurrentAccount;
import
com.packt.patterninspring.chapter2.model.SavingAccount;
public class AccountFactory extends AbstractFactory {
  final String CURRENT_ACCOUNT = "CURRENT";
  final String SAVING_ACCOUNT = "SAVING";
  @Override Bank getBank(String bankName) {
    return null;
  }
  //use getAccount method to get object of type Account
  //It is factory method for object of type Account
  @Override public Account getAccount(String accountType){
  if(CURRENT_ACCOUNT.equals(accountType)) {
    return new CurrentAccount();
  } else if(SAVING_ACCOUNT.equals(accountType)){
    return new SavingAccount();
  }
  return null;
  }
}
```

Abstract
Design Pattern (Contd.)



FactoryPatterMain.java is a demo class for the Abstract Factory design pattern. FactoryProducer is a class to get AbstractFactory in order to get the factories of concrete classes by passing a piece of information, such as the type:

```
package
com.packt.patterninspring.chapter2.factory.pattern;
import
com.packt.patterninspring.chapter2.model.Account;
public class FactoryPatterMain {
  public static void main(String[] args) {
     AccountFactory accountFactory = new
AccountFactory();
    //get an object of SavingAccount and call its
accountType() method.
    Account savingAccount =
accountFactory.getAccount("SAVING");
    //call accountType method of SavingAccount
    savingAccount.accountType();
    //get an object of CurrentAccount and call its
accountType() method.
    Account currentAccount =
accountFactory.getAccount("CURRENT");
    //call accountType method of CurrentAccount
    currentAccount.accountType();
  }
}
```

Singleton
Design Pattern

Ensure a class has only one instance and provide a global point of access to it

According to the singleton design pattern, the class provides the same single object for each call--that is, it is restricting the instantiation of a class to one object and provides a global point of access to that class.

This is useful when exactly one object is needed to coordinate actions across the system.

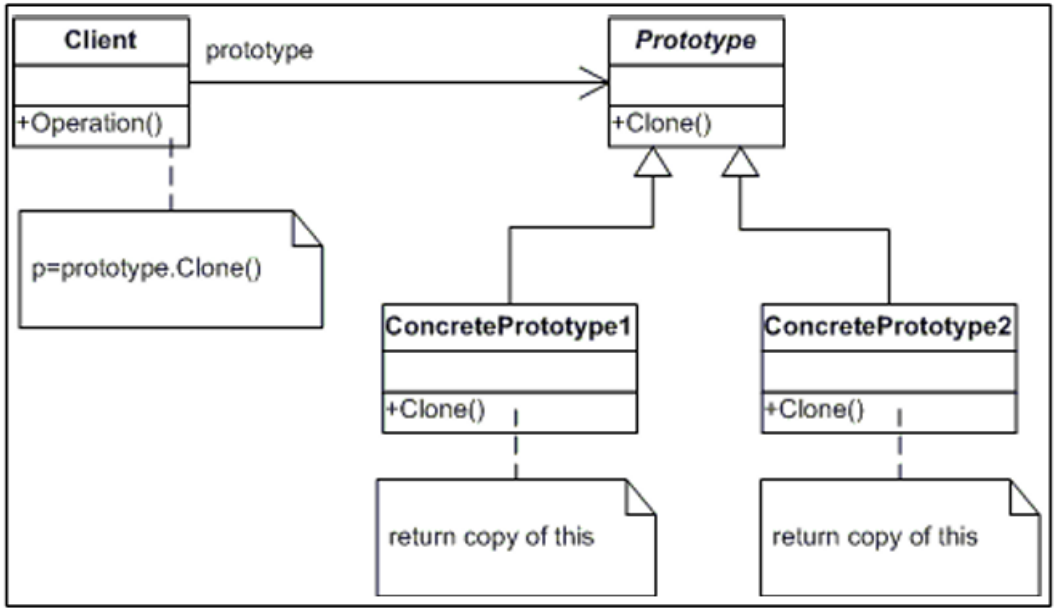
```
package com.packt.patterninspring.chapter2.singleton.pattern;
public class SingletonClass {
  private static SingletonClass instance = null;
  private SingletonClass() { }
  public static SingletonClass getInstance() {
    if (instance == null) {
     synchronized(SingletonClass.class){
       if (instance == null) {
         instance = new SingletonClass();
       }
     }
    }
    return instance;
  }
}
```

Prototype
Design Pattern

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

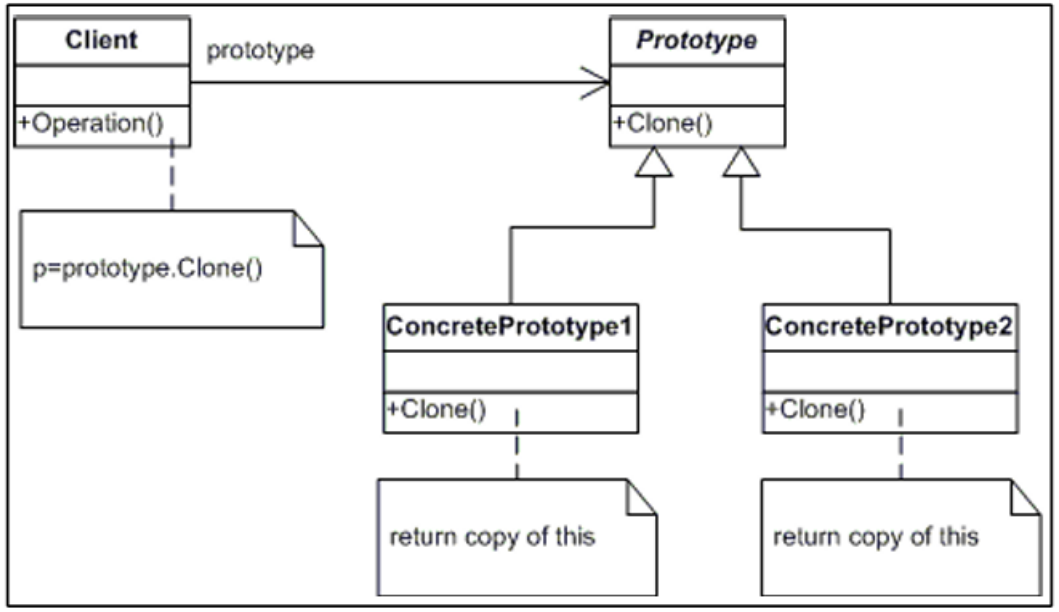
This pattern is used to create the objects by using a clone method of objects.

This pattern is used when the direct creation of the object is costly.



- **Prototype**: The Prototype is an interface. It uses the clone method to create instances of this interface type.
- **ConcretePrototype**: This is a concrete class of the Prototype interface to implement an operation to clone itself.
- **Client**: This is a caller class to create a new object of a Prototype interface by calling a clone method of the prototype interface.

Prototype
Design Pattern (Contd.)



Create an abstract class implementing the Clonable interface.

```
package com.packt.patterninspring.chapter2.prototype.pattern;
public abstract class Account implements Cloneable {
  abstract public void accountType();
  public Object clone() {
    Object clone = null; try {
      clone = super.clone();
    }
   catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    return clone;
  }
}
```

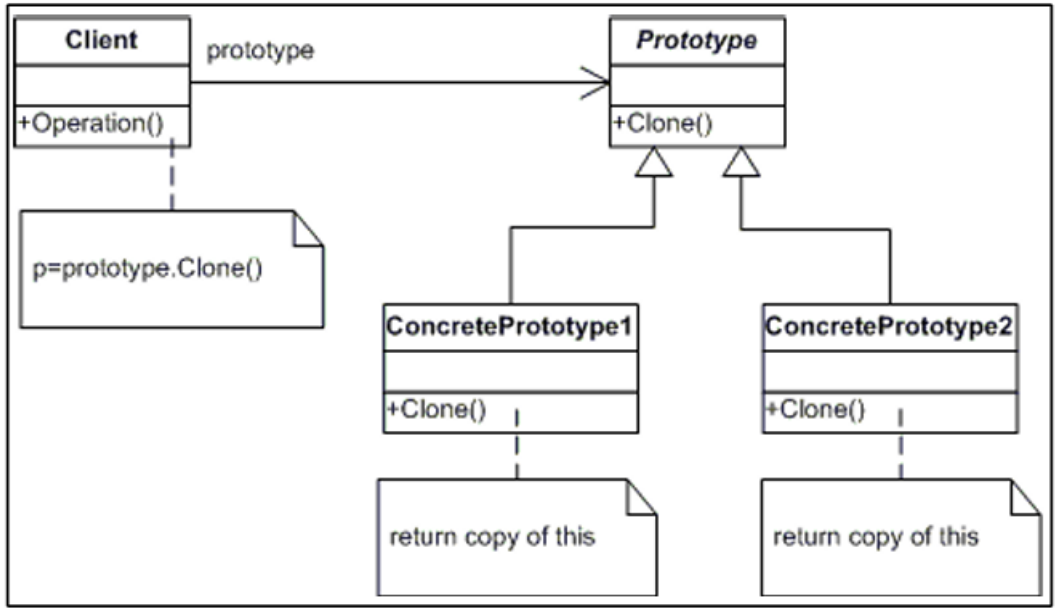
Now let's create concrete classes extending the preceding class: Here's the CurrentAccount.java file:

```
package com.packt.patterninspring.chapter2.prototype.pattern;
public class CurrentAccount extends Account {
  @Override
  public void accountType() {
    System.out.println("CURRENT ACCOUNT");
  }
}
```

Here's how SavingAccount.java should look:

```
package com.packt.patterninspring.chapter2.prototype.pattern;
public class SavingAccount extends Account {
  @Override
  public void accountType() {
    System.out.println("SAVING ACCOUNT");
  }
}
```

Prototype
Design Pattern (Contd.)



Let's create a class to get concrete classes in the AccountCache.java file:

```
package com.packt.patterninspring.chapter2.prototype.pattern;
import java.util.HashMap;
import java.util.Map;
public class AccountCache {
  public static Map<String, Account> accountCacheMap = new HashMap<>();
  static{
    Account currentAccount = new CurrentAccount();
    Account savingAccount = new SavingAccount();
    accountCacheMap.put("SAVING", savingAccount);
    accountCacheMap.put("CURRENT", currentAccount);
  }
}
```

PrototypePatternMain.java is a demo class that we will use to test the design pattern AccountCache to get the Account object by passing a piece of information, such as the type, and then call the clone() method:

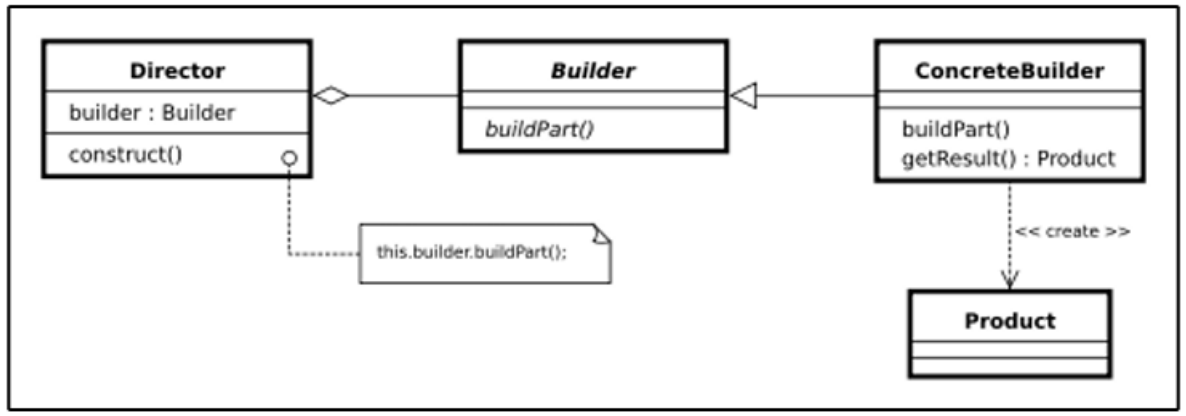
```
package com.packt.patterninspring.chapter2.prototype.pattern;
public class PrototypePatternMain {
  public static void main(String[] args) {
    Account currentAccount = (Account)
    AccountCache.accountCacheMap.get("CURRENT").clone();
    currentAccount.accountType();
    Account savingAccount = (Account)
    AccountCache.accountCacheMap.get("SAVING") .clone();
    savingAccount.accountType();
  }
}
```

Builder
Design Pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

The Builder design pattern is used to construct a complex object step by step, and finally it will return the complete object.

This pattern simplifies the construction of complex objects and it hides the details of the object's construction from the client caller code.



- **Builder (AccountBuilder):** This is an abstract class or interface for creating the details of an Account object.
- **ConcreteBuilder:** This is an implementation to construct and assemble details of the account by implementing the Builder interface.
- **Director:** This constructs an object using the Builder interface.
- **Product (Account):** This represents the complex object under construction. AccountBuilder builds the account's internal representation and defines the process by which it's assembled.

Sample implementation of the Builder design pattern

```
package com.packt.patterninspring.chapter2.builder.pattern;
public class Account {
  private String accountName;
  private Long accountNumber;
  private String accountHolder;
  private double balance; private String type; private double interest;
  private Account(AccountBuilder accountBuilder) {
    super();
    this.accountName = accountBuilder.accountName;
    this.accountNumber = accountBuilder.accountNumber;
    this.accountHolder = accountBuilder.accountHolder;
    this.balance = accountBuilder.balance;
    this.type = accountBuilder.type;
    this.interest = accountBuilder.interest;
  }
  //setters and getters
  public static class AccountBuilder {
    private final String accountName;
    private final Long accountNumber;
    private final String accountHolder;
    private double balance;
    private String type;
    private double interest;
    public AccountBuilder(String accountName, String accountHolder, Long accountNumber) {
      this.accountName = accountName;
      this.accountHolder = accountHolder; this.accountNumber = accountNumber;
    }
    public AccountBuilder balance(double balance) { this.balance = balance; return this; } public AccountBuilder type(String type) { this.type
= type; return this; } public AccountBuilder interest(double interest) { this.interest = interest; return this;
```

Sample implementation of the Builder design pattern (Contd.)

```
//setters and getters
  public static class AccountBuilder {
    private final String accountName;
    private final Long accountNumber;
    private final String accountHolder;
    private double balance;
    private String type;
    private double interest;
    public AccountBuilder(String accountName, String accountHolder, Long accountNumber) {
      this.accountName = accountName;
      this.accountHolder = accountHolder;
      this.accountNumber = accountNumber;
    }
    public AccountBuilder balance(double balance) {
      this.balance = balance;
      return this;
    }
    public AccountBuilder type(String type) {
      this.type = type;
      return this;
    }
    public AccountBuilder interest(double interest) {
      this.interest = interest;
      return this;
    }
```

Sample implementation of the Builder design pattern (Contd.)

```
    public Account build() {
      Account user = new Account(this);
      return user;
    }
  }
  public String toString() {
    return "Account [accountName=" + accountName + ", accountNumber=" + accountNumber + ", accountHolder=" + accountHolder + ",
    balance=" + balance + ", type=" + type + ", interest=" + interest + "]";
  }
}
```

AccountBuilderTest.java is a demo class that we will use to test the design pattern.
Let's look at how to build an Account object by passing the initial information to the object:

```
package com.packt.patterninspring.chapter2.builder.pattern;
public class AccountBuilderTest {
  public static void main(String[] args) {
    Account account = new Account.AccountBuilder("Saving Account", "Dinesh Rajput", 1111l)
.balance(38458.32).interest(4.5) .type("SAVING") .build(); System.out.println(account);
  }
}
```

Thank you

RMN