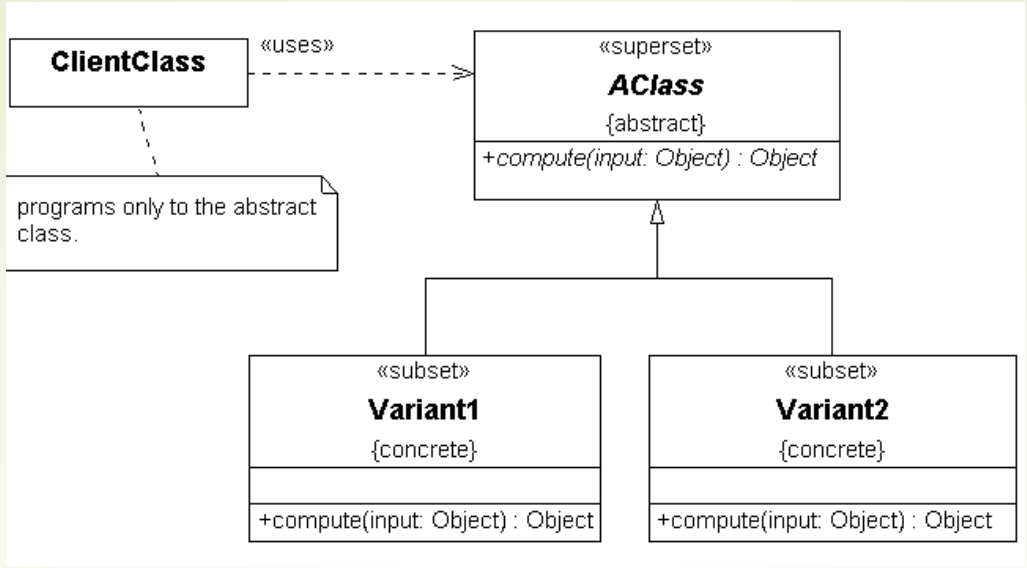


Lecture-13
Object oriented design patterns. Creational Patterns.

Lecturer: Temurbek A. Kuchkorov

Subject: OOP-II

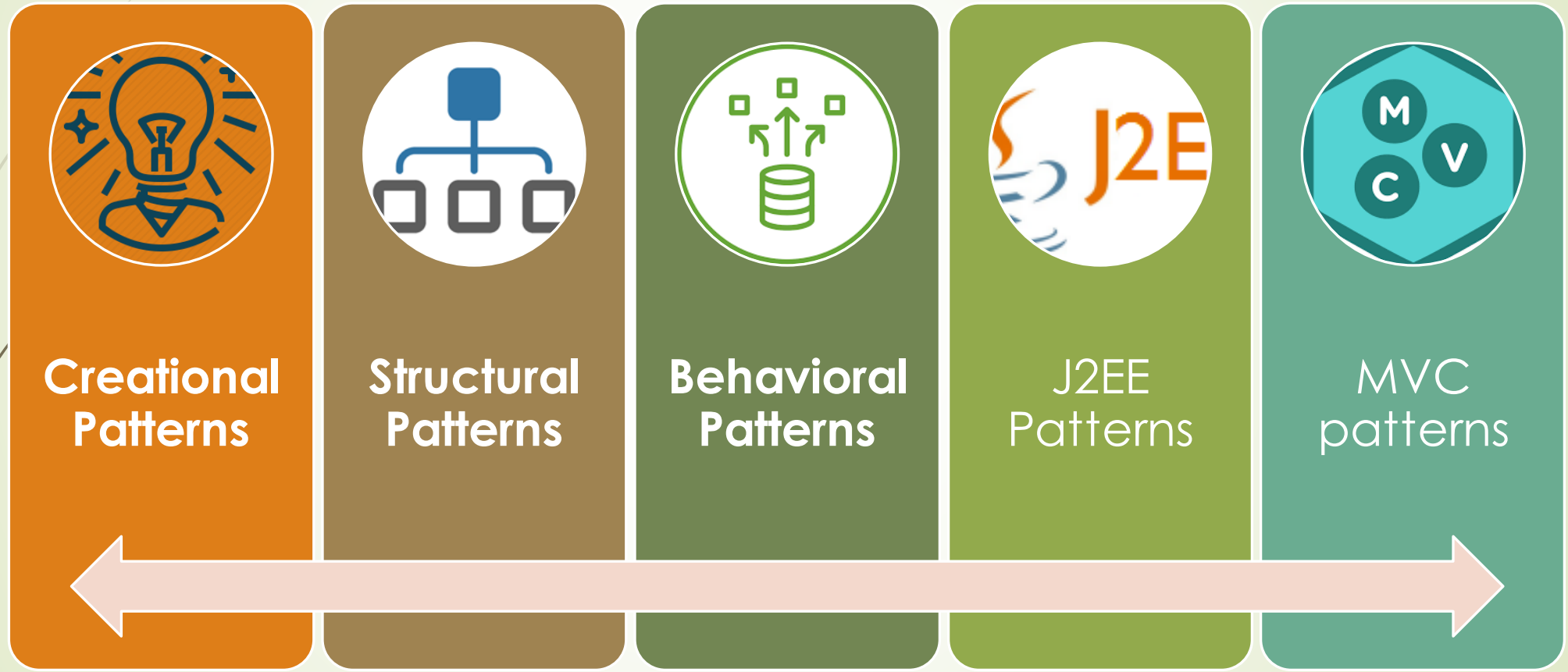
E-mail: timanet4u@gmail.com

Table of contents

1. Design patterns overview

- **Design patterns** represent the best practices used by experienced object-oriented software developers.
- **Design patterns** are solutions to general problems that software developers faced during software development.
- **Gang of Four (GOF)**. According to these authors design patterns are primarily based on the following principles of object orientated design.
  - Program to an interface not an implementation
  - Favor object composition over inheritance

2. Types of design patterns



3. Creational patterns

- Creational design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

1. Hiding the creation logic
2. Different method of instantiating objects
3. Flexibility for creating object

4. Singleton Pattern

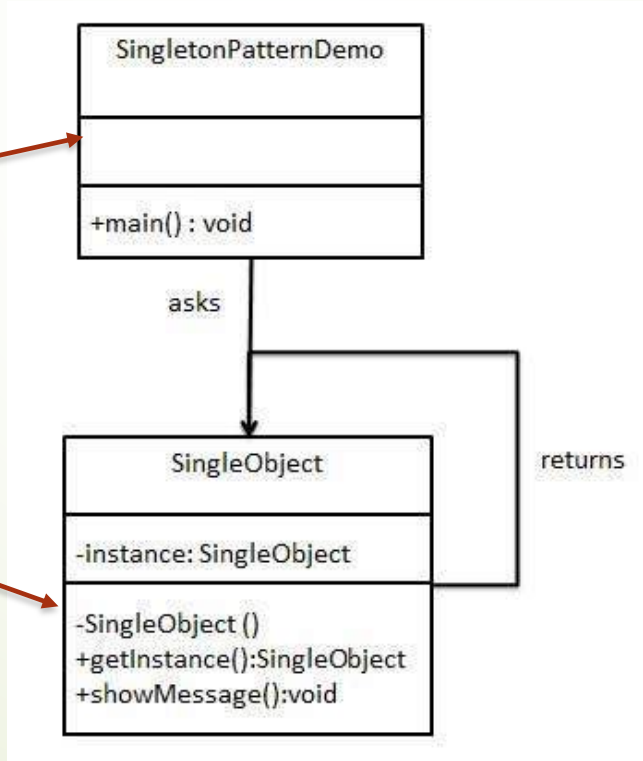
- Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- Main **characteristics of Singleton Pattern**:
  - Type of creational pattern;
  - Create only single object;
  - Create object without need to instantiate the object of the class.

4. Singleton Pattern. Implementation

- We're going to create a ***SingleObject* class**. *SingleObject* class have its constructor as private and have a static instance of itself.
- *SingleObject* class provides a static method to get its static instance to outside world. ***SingletonPatternDemo***, our demo class will use *SingleObject* class to get a *SingleObject* object.

Demo class

Singleton class



4. Singleton Pattern. Example

Step1. Create a Singleton Class. (*SingleObject.java*)

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Step2. Get the only object from the singleton class (*SingletonPatternDemo.java*)

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Step 3

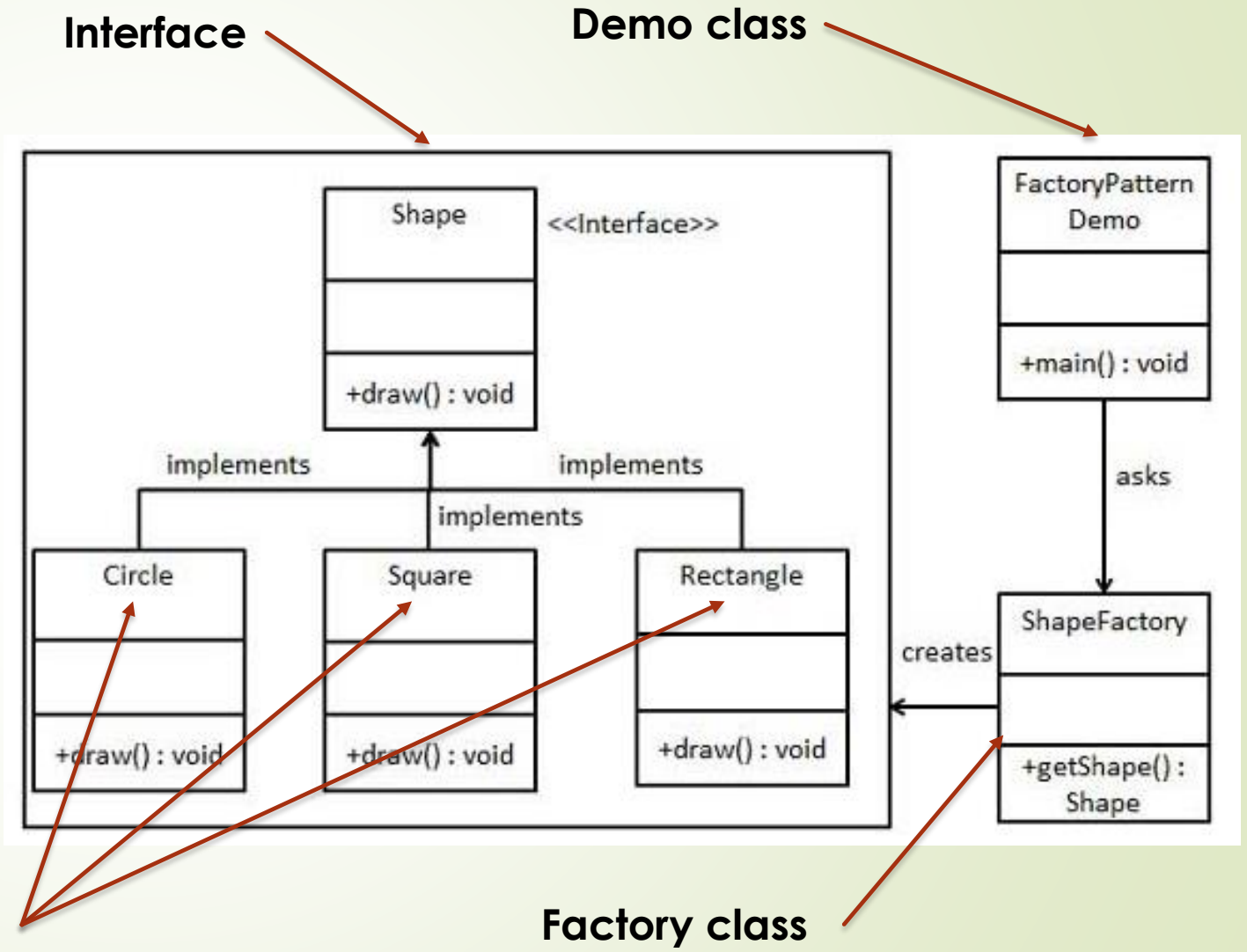
```
Hello World!
```

5. Factory Pattern

- **Factory pattern** is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- Main **characteristics of Factory pattern**:
  - Type of creational pattern;
  - The best way to create an object;
  - Create object without exposing the creation logic;
  - Newly created object using a common interface

5. Factory Pattern. Implementation

- We're going to create a ***Shape* interface** and concrete classes implementing the *Shape* interface. A factory class ***ShapeFactory*** is defined as a next step.
- ***FactoryPatternDemo***, our demo class will use ***ShapeFactory* to get a *Shape*object**. It will pass information ***(CIRCLE / RECTANGLE / SQUARE)*** to *ShapeFactory* to get the type of object it needs.



5. Factory Pattern. Example

Step 1. *Shape.java*

```
public interface Shape {
   void draw();
}
```

Step 2. *Rectangle.java*

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Step 3. Create a Factory to generate object.
ShapeFactory.java

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

5. Factory Pattern. Example [cont.]

Step 4. Use the Factory to get object of concrete class by passing an information such as type. (*FactoryPatternDemo.java)*

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

Step 5. Check result

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

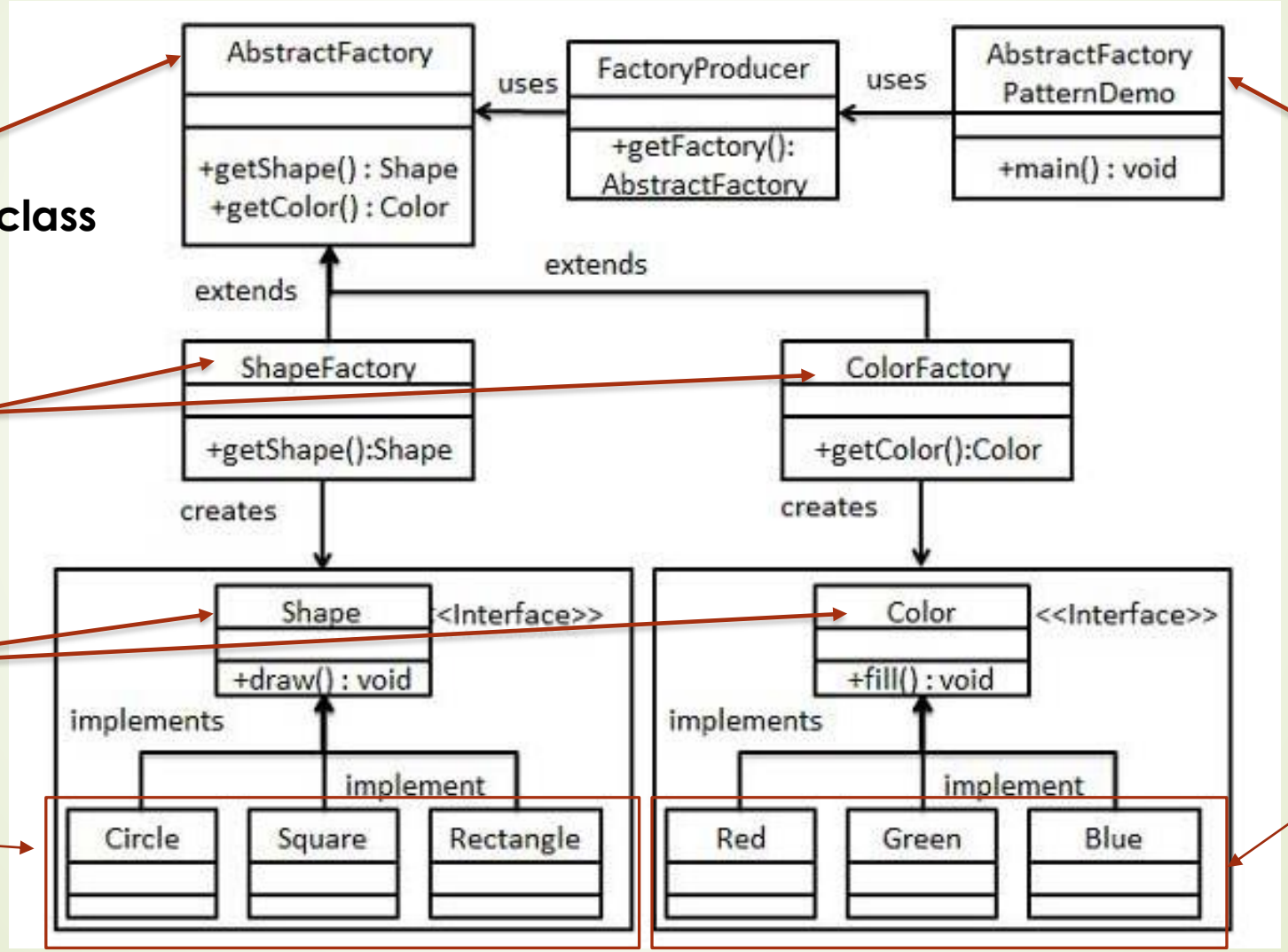
6. Abstract Factory Pattern

- **Abstract Factory patterns** work around a super-factory which creates other factories. This factory is also called as **factory of factories**. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- Main **characteristics of Factory pattern**:
  - Type of creational pattern;
  - The best way to create an object;
  - The interface is responsible for creating a factory of related objects without explicitly specifying their classes;
  - Each generated factory can give the objects.

6. Abstract Factory Pattern. Implementation

- We are going to create a ***Shape*** and ***Color*** **interfaces** and concrete classes implementing these interfaces.
- We create an **abstract factory class** ***AbstractFactory*** as next step.
- Factory classes ***ShapeFactory*** and ***ColorFactoryare*** defined where each factory **extends** ***AbstractFactory***.
- A factory **creator/generator class** ***FactoryProducer*** is created.
- ***AbstractFactoryPatternDemo***, our demo class **uses *FactoryProducer* to get a *AbstractFactory*** object.
- It will pass information **(*CIRCLE* / *RECTANGLE* / *SQUARE* for *Shape*)** to ***AbstractFactory*** to get the type of object it needs.
- It also passes information **(*RED* / *GREEN* / *BLUE* for *Color*)** to ***AbstractFactory*** to get the type of object it needs.

6. Abstract Factory Pattern. Implementation [cont.]



Abstract factory class

Demo class

Factory classes

Interfaces

Classes

Classes

6. Abstract Factory Pattern. Example

Step 1. Create an interface Item representing food item and packing. (*Item.java*)

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

Packing.java

```
public interface Packing {
   public String pack();
}
```

Step 2. Create concrete classes implementing the Packing interface. (*Wrapper.java)*

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

Step 3. Create abstract classes implementing the item interface providing default functionalities.

Burger.java

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {

        @Override
        public Packing packing() {
       return new Bottle();
        }

        @Override
        public abstract float price();
}
```

6. Abstract Factory Pattern. Example [cont.]

Step 4. Create concrete classes extending Burger and ColdDrink classes

VegBurger.java

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Coke.java

```
public class Coke extends ColdDrink {

   @Override
   public float price() {
      return 30.0f;
   }

   @Override
   public String name() {
      return "Coke";
   }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

6. Abstract Factory Pattern. Example [cont.]

Step 5. Create a Meal class having Item objects defined above.

Meal.java

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }

   public float getCost(){
      float cost = 0.0f;

      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){

      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " + item.packing().pack());
         System.out.println(", Price : " + item.price());
      }
   }
}
```

6. Abstract Factory Pattern. Example [cont.]

Step 6. Create a MealBuilder class, the actual builder class responsible to create Meal objects.

MealBuilder.java

```
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

6. Abstract Factory Pattern. Example [cont.]

Step 7. BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

BuilderPatternDemo.java

```
public class BuilderPatternDemo {
   public static void main(String[] args) {

      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " + vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\n\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " + nonVegMeal.getCost());
   }
}
```

6. Abstract Factory Pattern. Example [cont.]

Step 8. Check the result

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0



Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

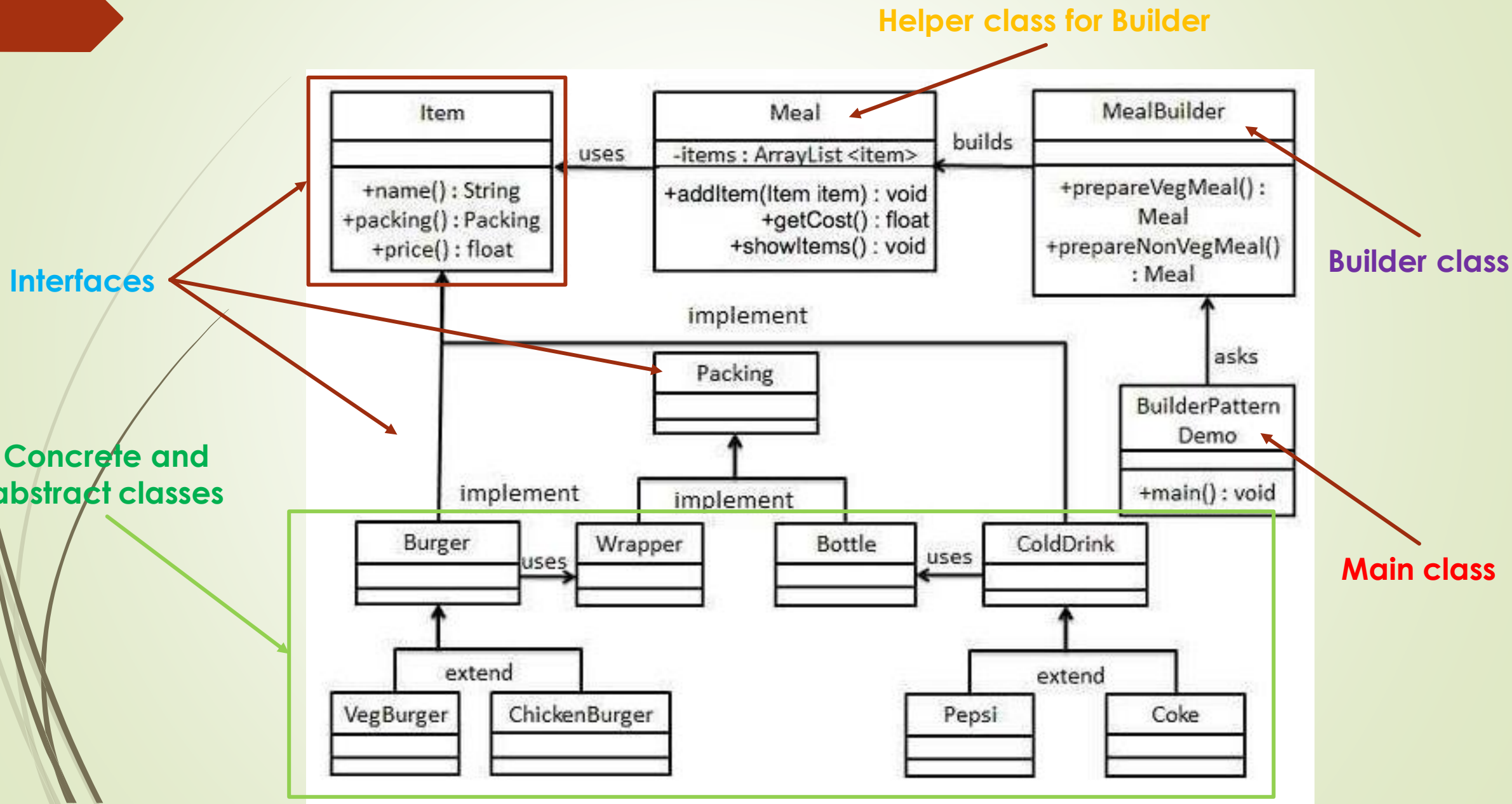
7. Builder Pattern

- Builder pattern builds a complex object using simple objects and using a step by step approach.
- Main **characteristics of Factory pattern**:
  - Type of creational design pattern
  - This pattern provides one of the best ways to create an object
  - A Builder class builds the final object step by step.
  - This builder is independent of other objects.

7. Builder Pattern. Implementation

- We have considered a business case of **fast-food restaurant** where a typical meal could be a burger and a cold drink.
- We are going to create an ***Item*** interface representing food items such as burgers and cold drinks, ***Packing*** interface representing packaging of food items and concrete classes (***VegBurger***, ***ChickenBurger***, ***Pepsi***, ***Coke***)
- We then create a ***Meal*** class having ***ArrayList*** of *Item* and a ***MealBuilder*** to build different types of *Meal* objects by combining *Item*.
- ***BuilderPatternDemo***, our demo class will use *MealBuilder* to build a *Meal*.

7. Builder Pattern. Implementation [cont.]



7. Builder Pattern. Implementation [cont.]

Step 1. Create an interface Item representing food item and packing.

Item.java

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

Packing.java

```
public interface Packing {
   public String pack();
}
```

Step 2. Create concrete classes implementing the Packing interface.

Wrapper.java

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

Step 3. Create abstract classes implementing the item interface.

Burger.java

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {

        @Override
        public Packing packing() {
       return new Bottle();
        }

        @Override
        public abstract float price();
}
```

7. Builder Pattern. Implementation [cont.]

Step 4. Create concrete classes extending Burger and ColdDrink classes.

VegBurger.java

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Coke.java

```
public class Coke extends ColdDrink {

   @Override
   public float price() {
      return 30.0f;
   }

   @Override
   public String name() {
      return "Coke";
   }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

7. Builder Pattern. Implementation [cont.]

Step 5. Create a Meal class having Item objects defined above.

Meal.java

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }

   public float getCost(){
      float cost = 0.0f;

      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){

      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " + item.packing().pack());
         System.out.println(", Price : " + item.price());
      }
   }
}
```

Step 6. Create a MealBuilder class, the actual builder class responsible to create Meal objects.

ChickenBurger.java

```
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

7. Builder Pattern. Implementation [cont.]

Step 7. BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

BuilderPatternDemo.java

```
public class BuilderPatternDemo {
   public static void main(String[] args) {

      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " + vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\n\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " + nonVegMeal.getCost());
   }
}
```

Step 8. Verify the output.

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0



Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

Conclusion

- **Design patterns** help to developers writing flexible code and that patterns represent the best practices used by experienced object-oriented approaches.
- **Design patterns** are solutions to general problems that software developers faced during software development.
- **Creational design patterns** provide a way to create objects while hiding the creation logic.

References

- https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
- https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm
- https://www.tutorialspoint.com/design_pattern/factory_pattern.htm
- https://www.tutorialspoint.com/design_pattern/prototype_pattern.htm
- https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
- https://www.tutorialspoint.com/design_pattern/builder_pattern.htm
- http://www.oodesign.com/singleton-pattern.html
- http://www.oodesign.com/factory-pattern.html