Design Patterns

Khoa CNTT – Trường Đại học Công nghệ - ĐHQGHN

Design pattern

- A design pattern is a resuable solution to a common problem in software design
- Design patterns are not specific pieces of code but rather general guidelines or strategies that can be implemented in various programming languages and contexts

Key characteristics

- Resuable: Can be applied to different problems in similar contexts
- Language-agnostic: Not tied to a specific programming language
- Well-documented: Includes the problem it solves, the solution, and consequences of its use
- Tested: Proven to work through prior use and testing

Types of design patterns

- Creational patterns:
  - Deal with object creation mechanisms to ensure objects are created in a way suitable for the situation
  - E.g., singleton, factory method, prototype
- Stuctural patterns:
  - Focus on the composition of classes or objects to form larger structures
  - E.g., Adapter, Composite, Proxy
- Behavioral patterns:
  - Concerned with object interactions and responsibilities
  - E.g., Observer, strategy

Benefits of design patterns

- Improved code readability and maintainability
- Encourages best practices
- Enhances flexibility

Creational patterns

Singleton pattern

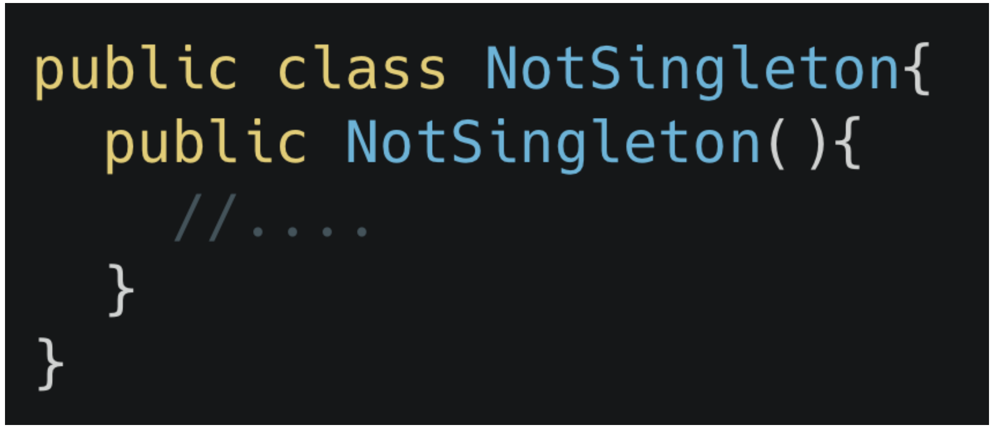
Factory method pattern

Singleton pattern

Singleton pattern

- Describes the way to create an object
- Enforces one and only one object of a Singleton class
- Has the Singleton object globally accessible

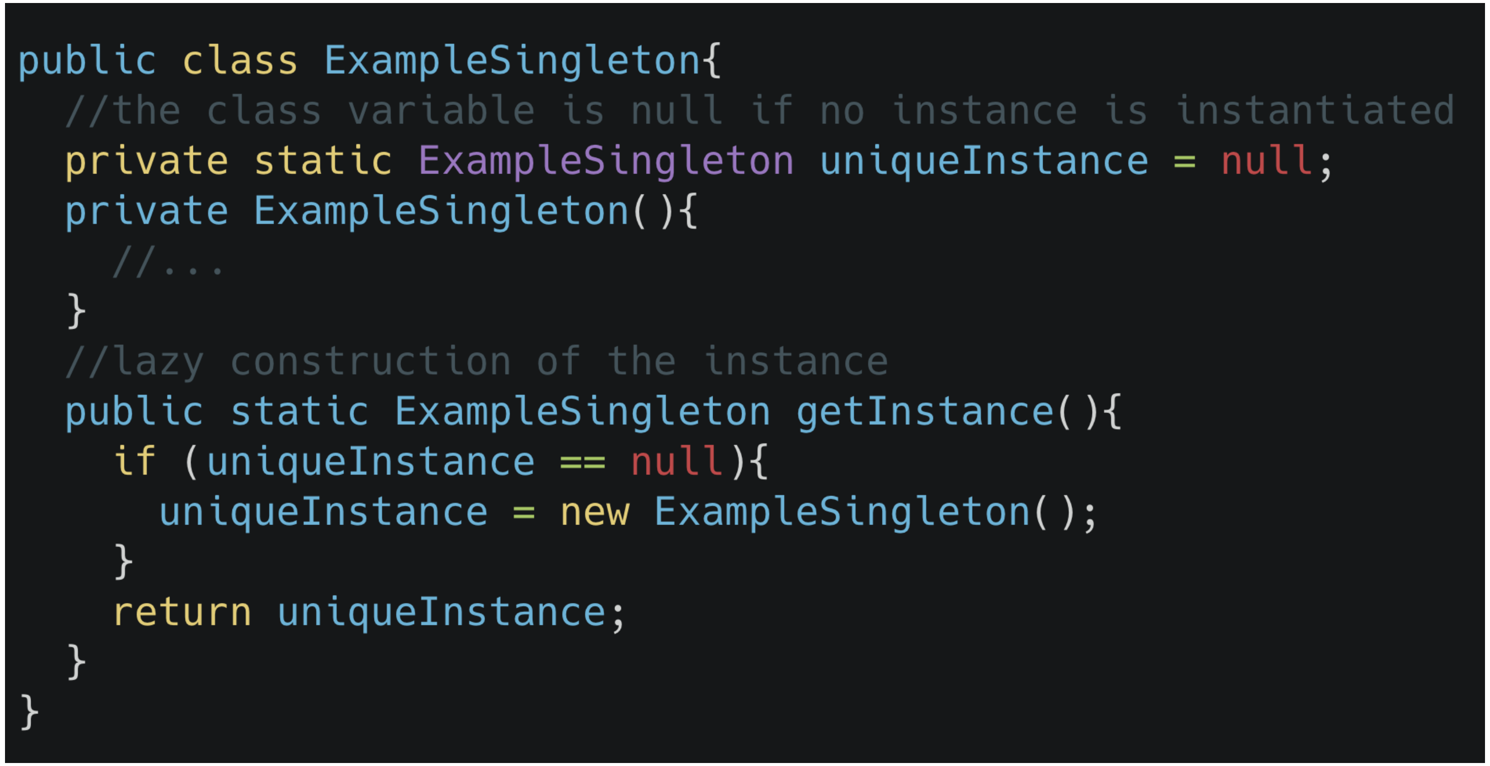
Example



```
public class NotSingleton{
  public NotSingleton( ){
    //....
  }
}
```

- Public constructor
- This constructor may instantiate an object of this class at any time

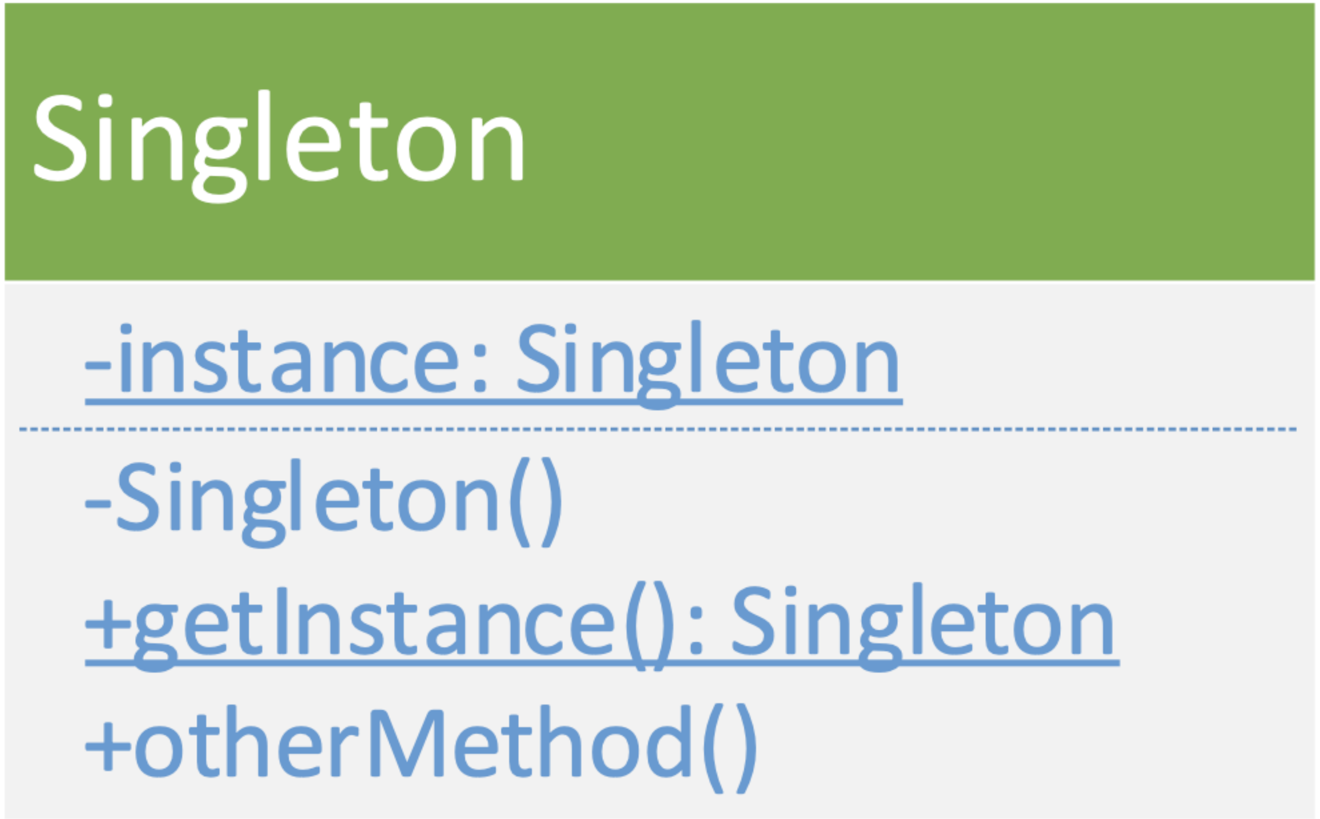
Singleton Pattern - Example



```java
public class ExampleSingleton{
  //the class variable is null if no instance is instantiated
  private static ExampleSingleton uniqueInstance = null;
  private ExampleSingleton(){
    //...
  }
  //lazy construction of the instance
  public static ExampleSingleton getInstance(){
    if (uniqueInstance == null){
      uniqueInstance = new ExampleSingleton();
    }
    return uniqueInstance;
  }
}
```

- Private constructor
- Public method instantiates the class object, if it is not already instantiated

Singleton pattern



Singleton

-instance: Singleton

-Singleton()

+getInstance(): Singleton

+otherMethod()

Factory Method Pattern

Factory Method Pattern

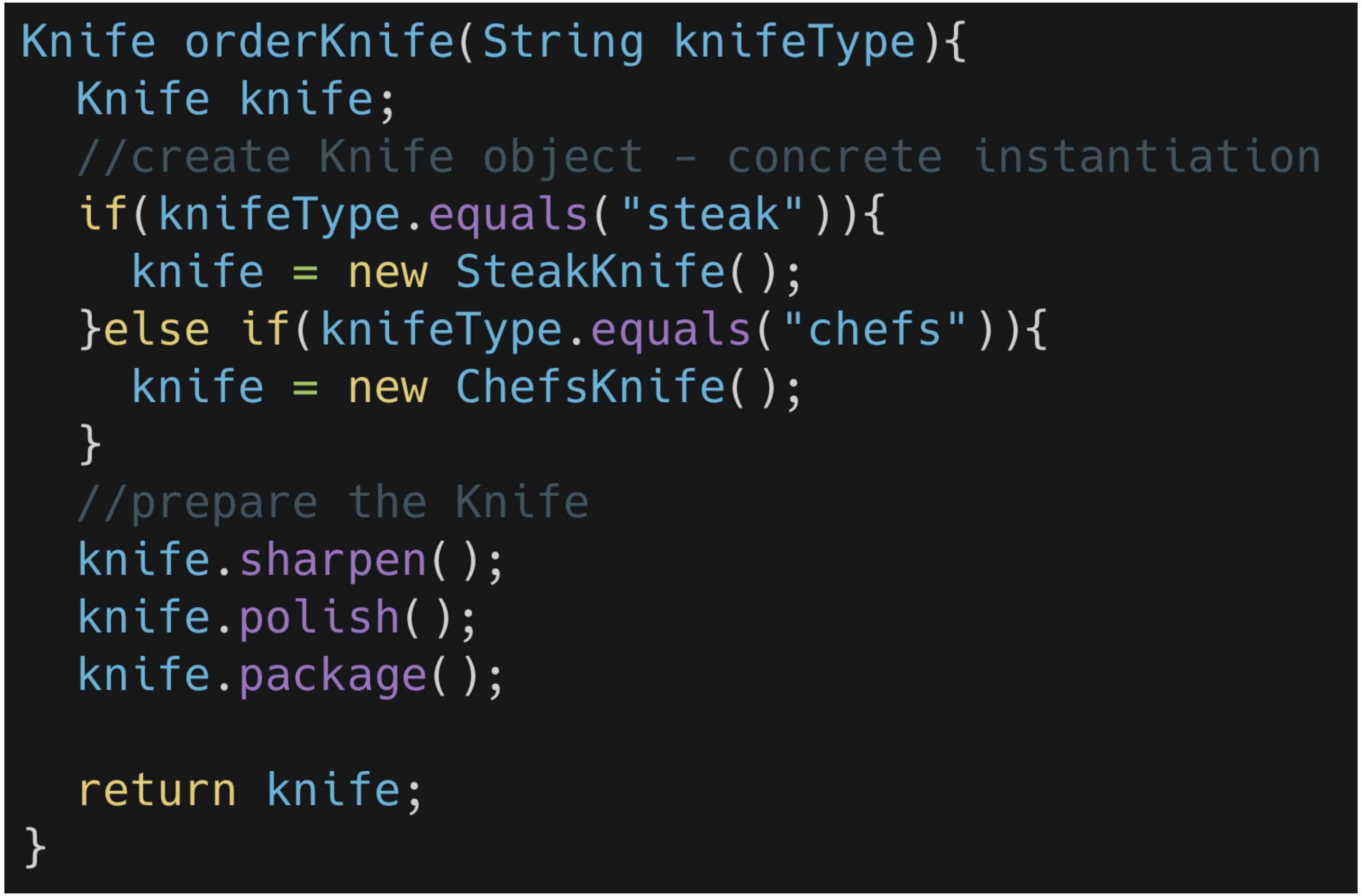
- Purpose of the factory method pattern is creating objects
- Make the software to be easily maintained and changed

Example

- You build an online store to sell knifes
- There multiple type of knifes, for example SteakKnife and ChefsKnife

→ You build a class Knife, and sub-classes SteakKnife and ChefsKnife

Example



```
Knife orderKnife(String knifeType){
  Knife knife;
  //create Knife object - concrete instantiation
  if(knifeType.equals("steak")){
    knife = new SteakKnife();
  }else if(knifeType.equals("chefs")){
    knife = new ChefsKnife();
  }
  //prepare the Knife
  knife.sharpen();
  knife.polish();
  knife.package();

  return knife;
}
```

Example

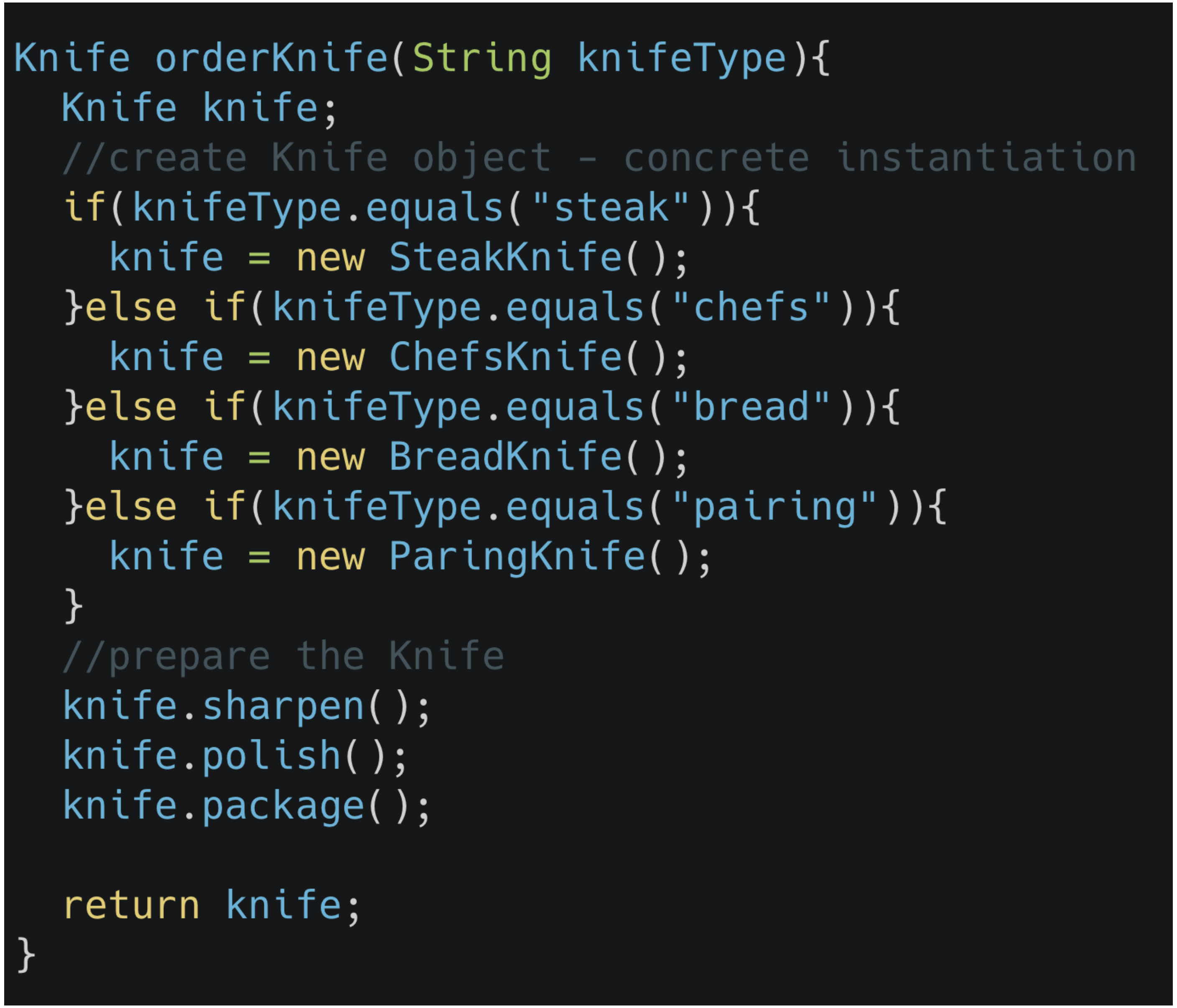
Now your sales improve, you want to add more and more types of knife to your store.

→ New sub-classes are added.

Example

- The list of conditionals grows and grows as new knife types are added.

→ This is getting pretty complicated



```
Knife orderKnife(String knifeType){
  Knife knife;
  //create Knife object - concrete instantiation
  if(knifeType.equals("steak")){
    knife = new SteakKnife();
  }else if(knifeType.equals("chefs")){
    knife = new ChefsKnife();
  }else if(knifeType.equals("bread")){
    knife = new BreadKnife();
  }else if(knifeType.equals("pairing")){
    knife = new ParingKnife();
  }
  //prepare the Knife
  knife.sharpen();
  knife.polish();
  knife.package();

  return knife;
}
```

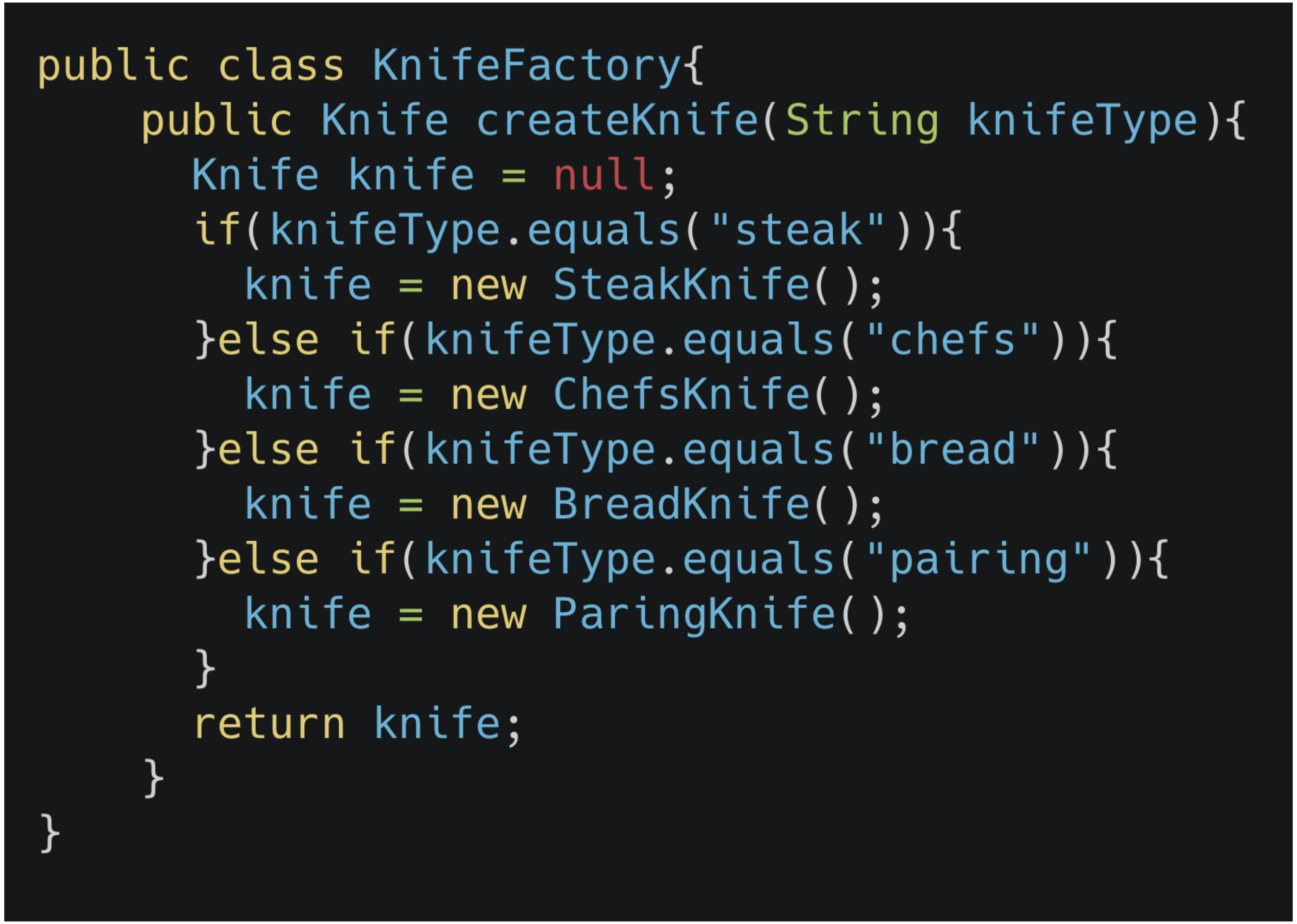
Example

Solution: We create a factory object whose role is to create objects of particular types

- Sharpening, polishing, and packaging will stay where it is in the orderKnife method
- The responsibility of product object creation is delegated to another object

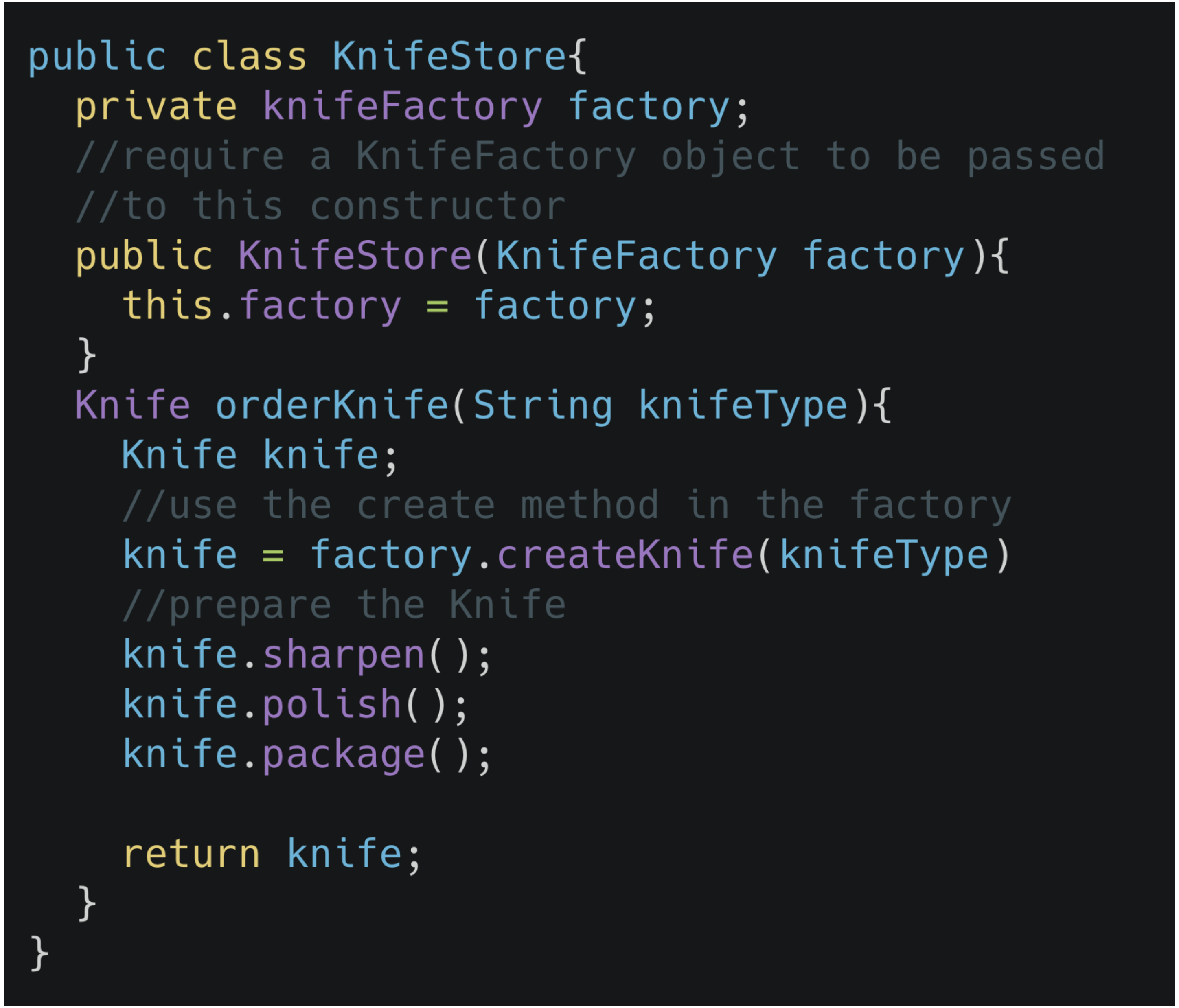
Example

- Factory object is an instance of a Factory class which has a method to create product objects
- The KnifeFactory object can be used in the KnifeStore class



```java
public class KnifeFactory{
    public Knife createKnife(String knifeType){
      Knife knife = null;
      if(knifeType.equals("steak")){
        knife = new SteakKnife();
      }else if(knifeType.equals("chefs")){
        knife = new ChefsKnife();
      }else if(knifeType.equals("bread")){
        knife = new BreadKnife();
      }else if(knifeType.equals("pairing")){
        knife = new ParingKnife();
      }
      return knife;
    }
}
```

Example



```
public class KnifeStore{
  private knifeFactory factory;
  //require a KnifeFactory object to be passed
  //to this constructor
  public KnifeStore(KnifeFactory factory){
    this.factory = factory;
  }
  Knife orderKnife(String knifeType){
    Knife knife;
    //use the create method in the factory
    knife = factory.createKnife(knifeType)
    //prepare the Knife
    knife.sharpen();
    knife.polish();
    knife.package();

    return knife;
  }
}
```

Benefits of factory objects

- It can be used to create products for multiple clients
- If there are multiple clients that want to instantiate the same set of classes, then by using a Factory object, you have cut out redundant code and made the software easier to modify.

Factory method pattern

- Instead of using a separate object to create the products/objects like Factory object, Factory method pattern uses a separate method in the same class to create objects

Example

- Now, you want to create different specific types of knifes
  - Factory object approach would subclass the factory class

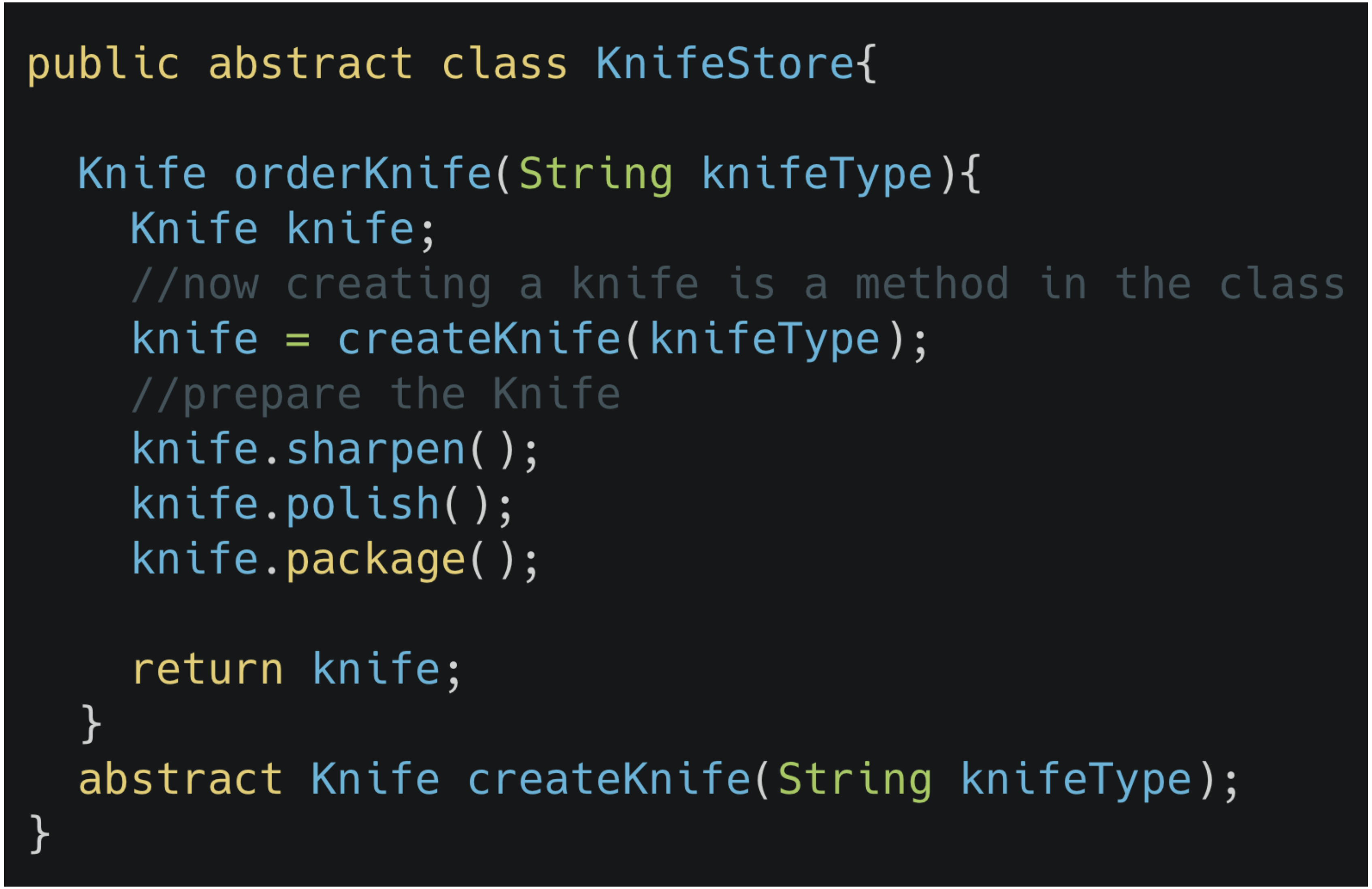
    Ex: **BudgetKnifeFactory** create budget chefs knife and budget steak knife products

    QualityKnifeFactory create quality chefs knife and quality steak knife products

  - Factory method approach has **BugdetKnifeStore** subclass KnifeStore. Also, **BugdetKnifeStore** has a method (factory method) creating budget chefs knife and budget steak knife products

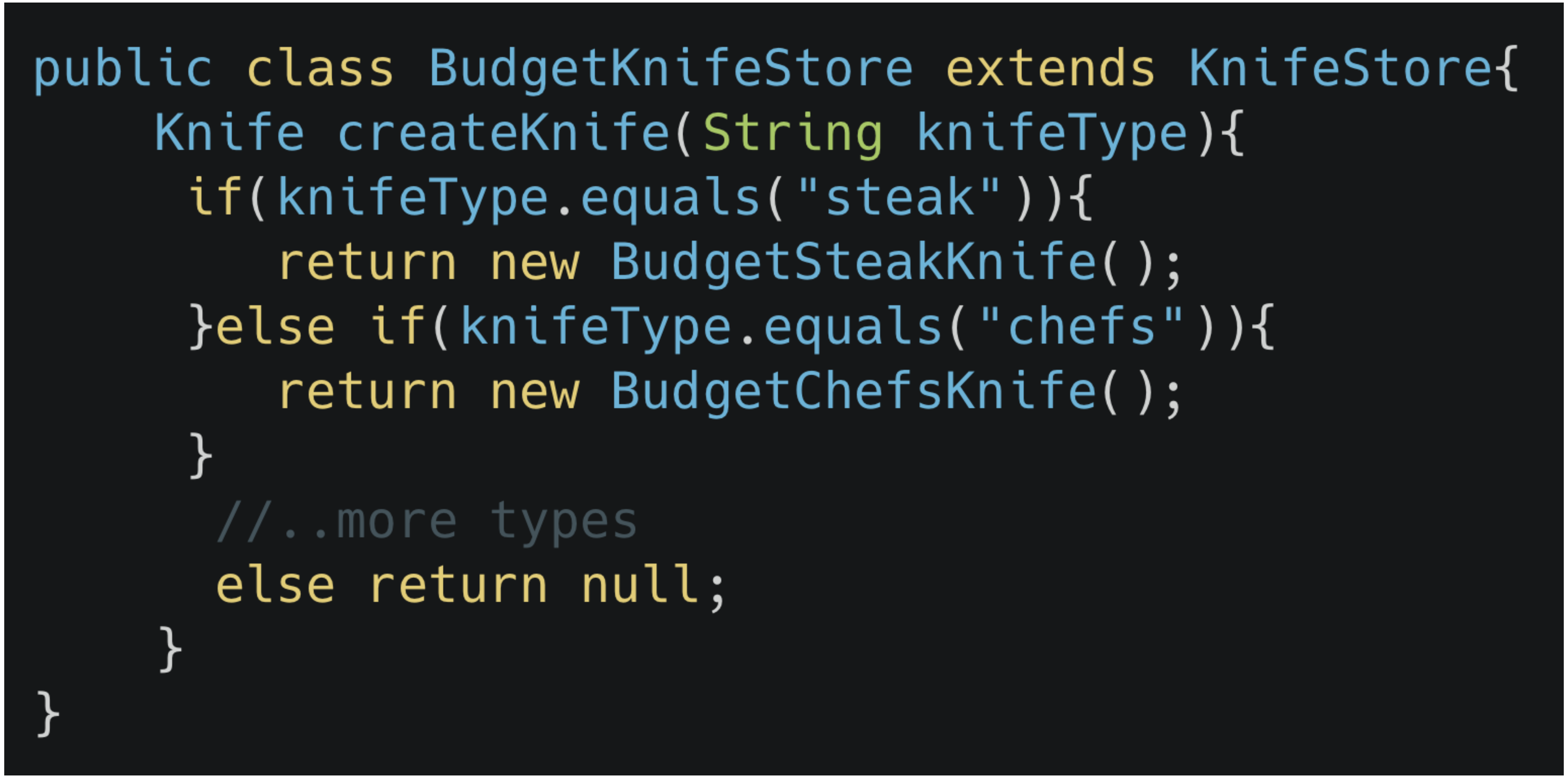
Example

- createKnife() is the factory method
- Factory method is abstract since we want the factory method to be defined by the subclasses



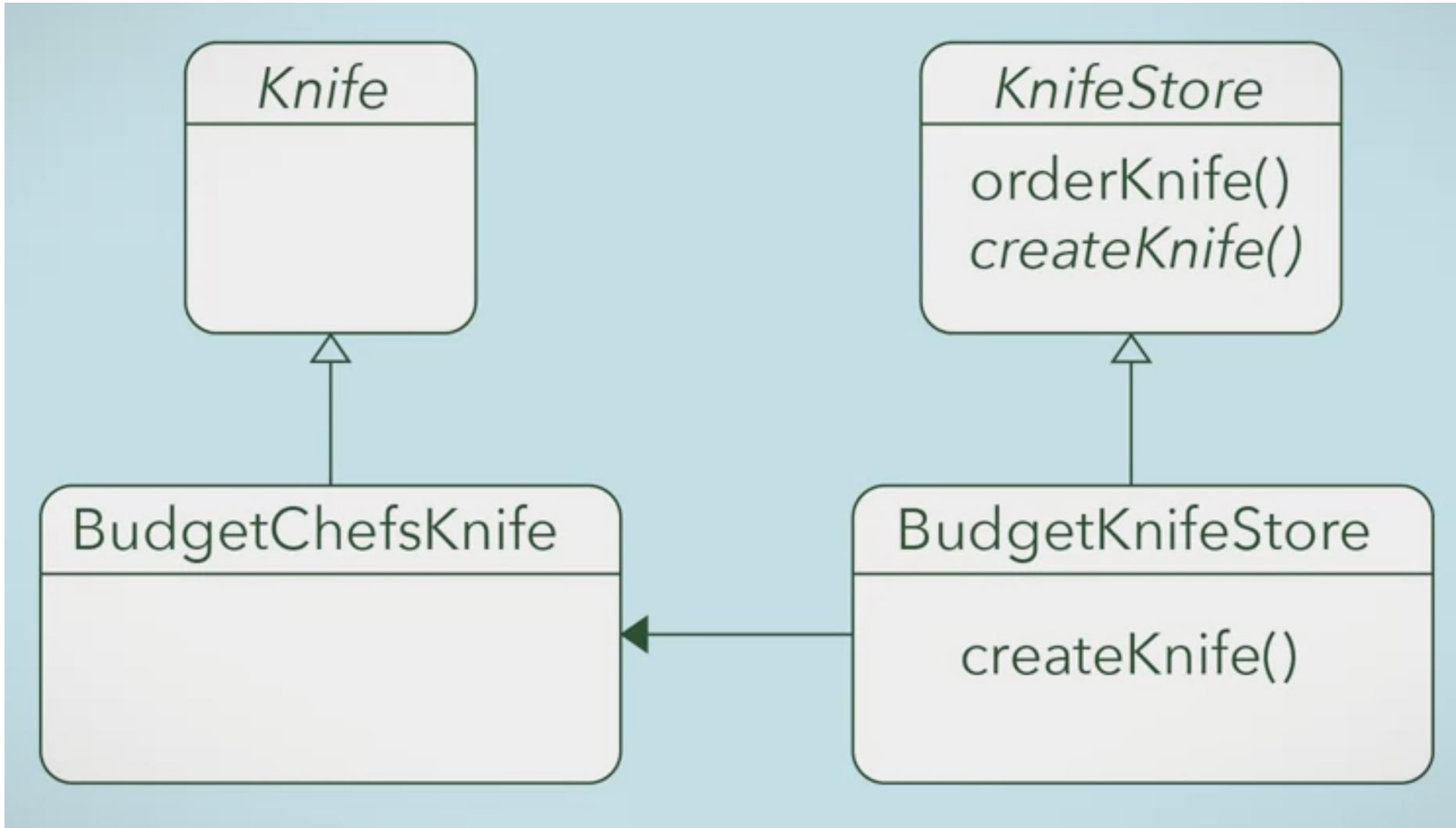
```
public abstract class KnifeStore{

  Knife orderKnife(String knifeType){
    Knife knife;
    //now creating a knife is a method in the class
    knife = createKnife(knifeType);
    //prepare the Knife
    knife.sharpen();
    knife.polish();
    knife.package();

    return knife;
  }
  abstract Knife createKnife(String knifeType);
}
```

Example

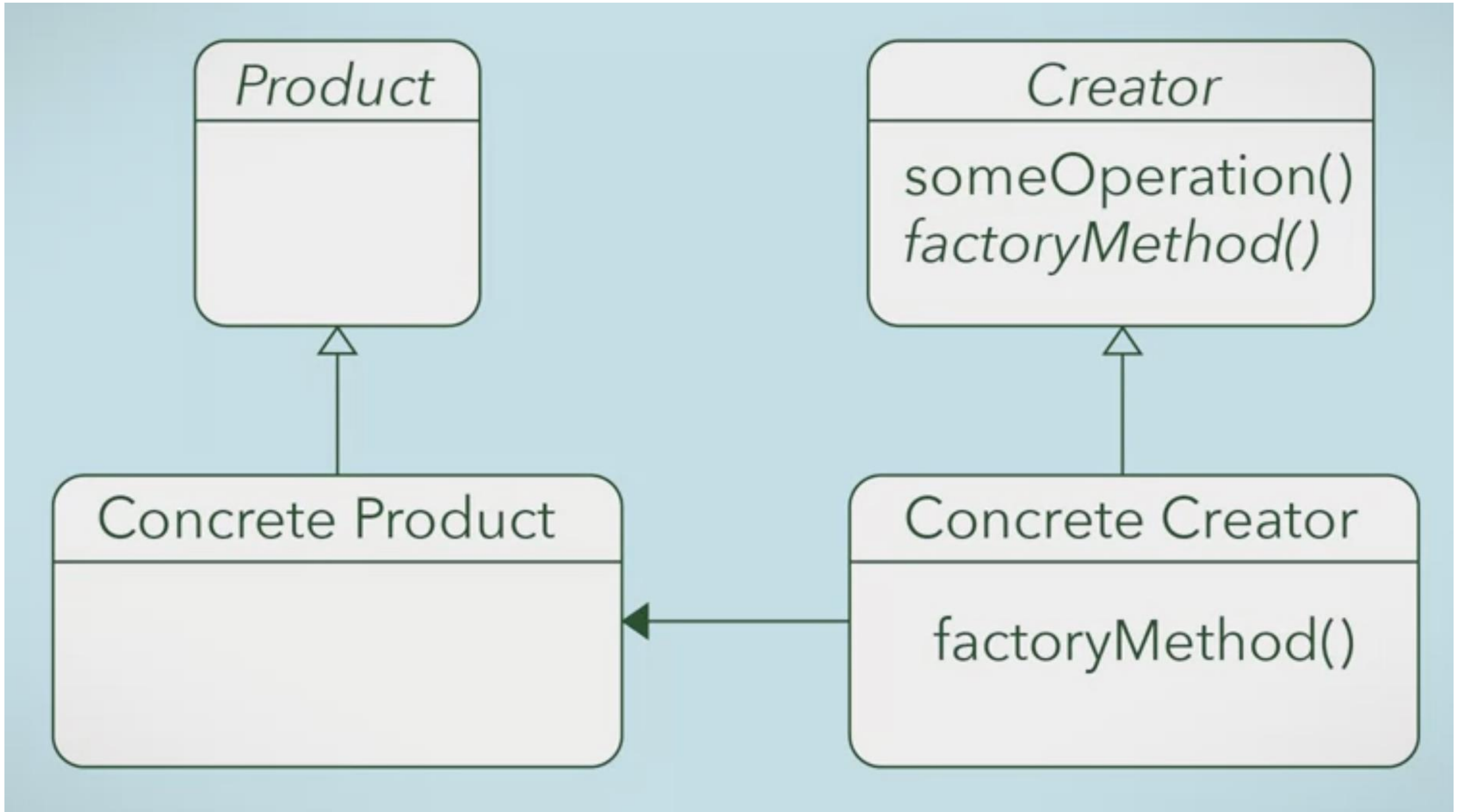


```java
public class BudgetKnifeStore extends KnifeStore{
    Knife createKnife(String knifeType){
     if(knifeType.equals("steak")){
        return new BudgetSteakKnife();
     }else if(knifeType.equals("chefs")){
        return new BudgetChefsKnife();
     }
      //..more types
      else return null;
    }
}
```

Example


Knife
KnifeStore
orderKnife()
createKnife()
BudgetChefsKnife
BudgetKnifeStore
createKnife()

Factory method pattern



Factory method pattern

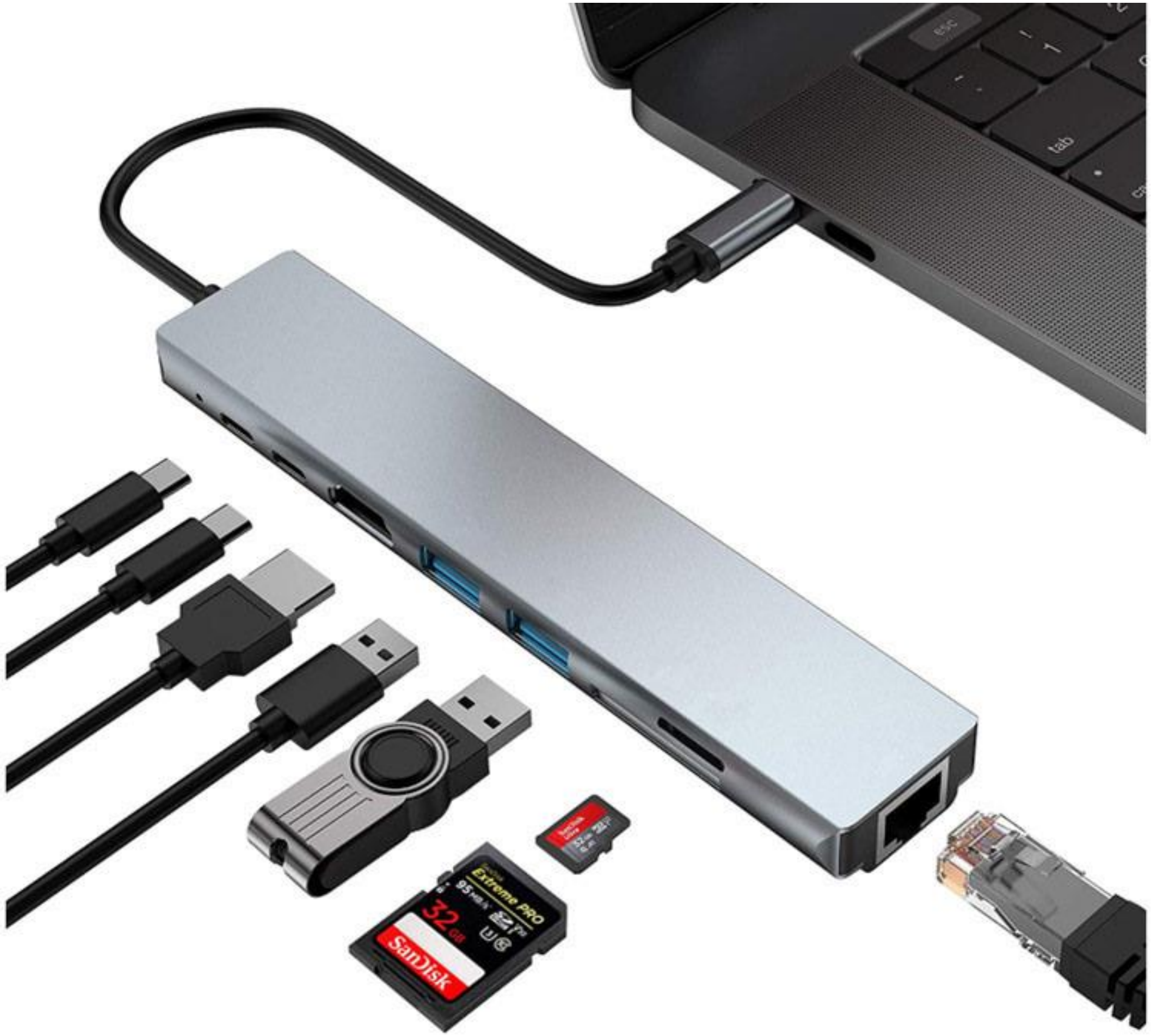
- The factory method design pattern defines an interface for creating objects, but let the subclasses decide which class to instantiate

Structual patterns

Adapter

Adapter pattern

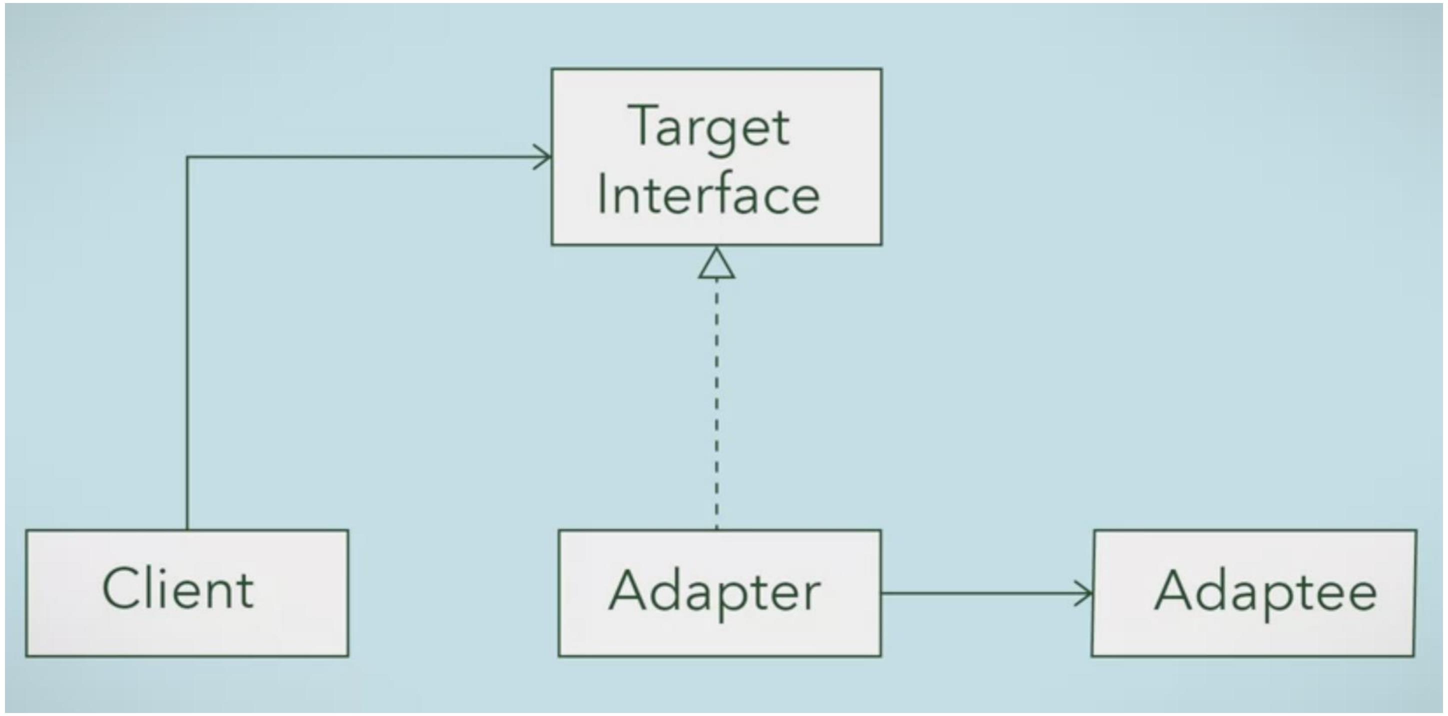
Adapter



Adapter pattern

- Software systems could have incompatible software interfaces
  - Output of one system may not conform with the expected input of another system
- The adapter design pattern will help to faciliate communication between two existing systems by providing a comptible interface

Example



- Client is your system
- Adaptee is a third-party library
- Adapter implement Target Interface which is the interface that the Client use

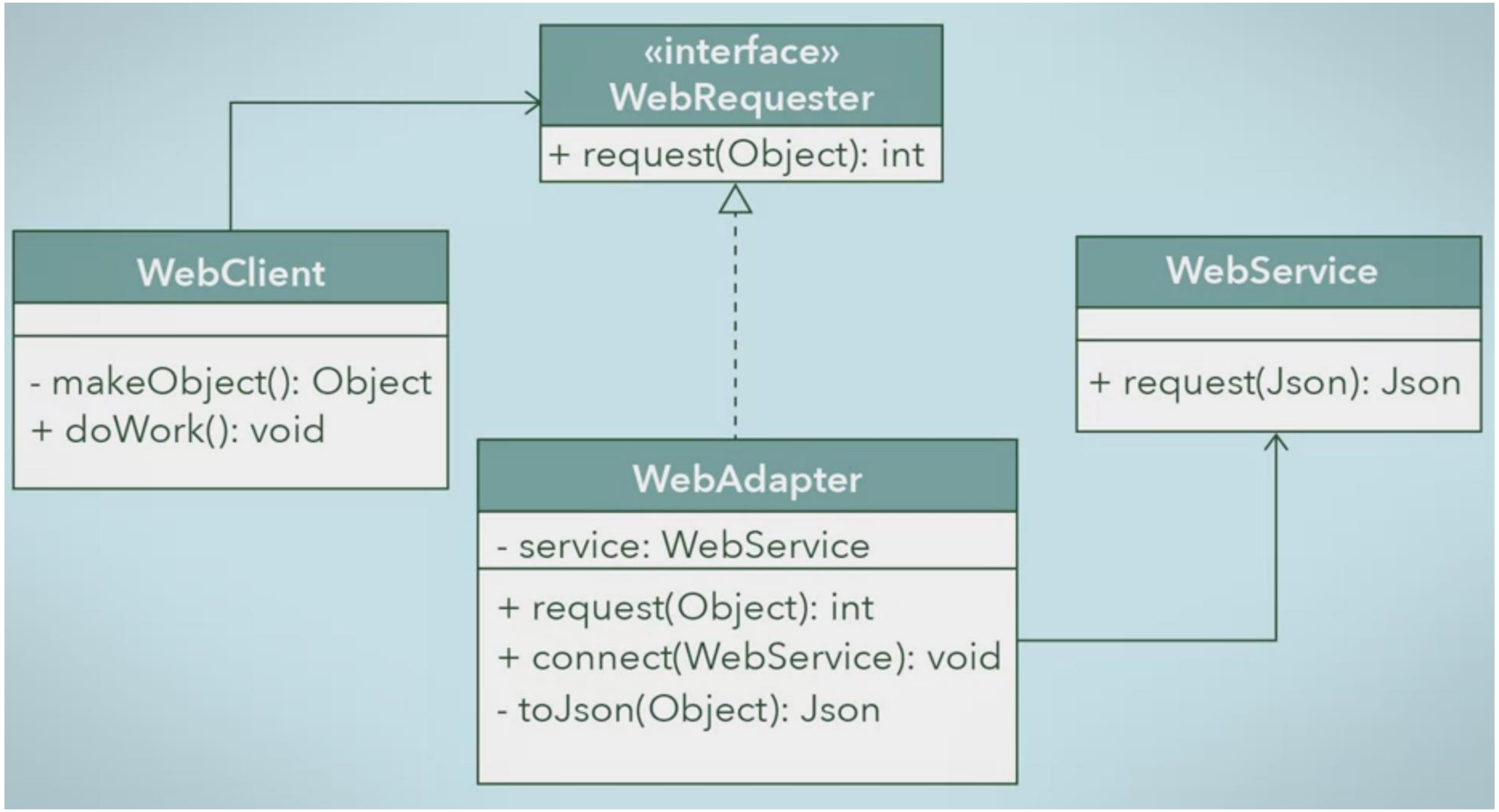
Adapter pattern

- If two interfaces are incompatible, why don't we change one?
  - We may not have direct access to the third-party library or external system.
  - Our changes could break those libraries or systems
  - If we change our system's interface, the changing assumptions could break other parts of our system (e.g., which also use those libaries)

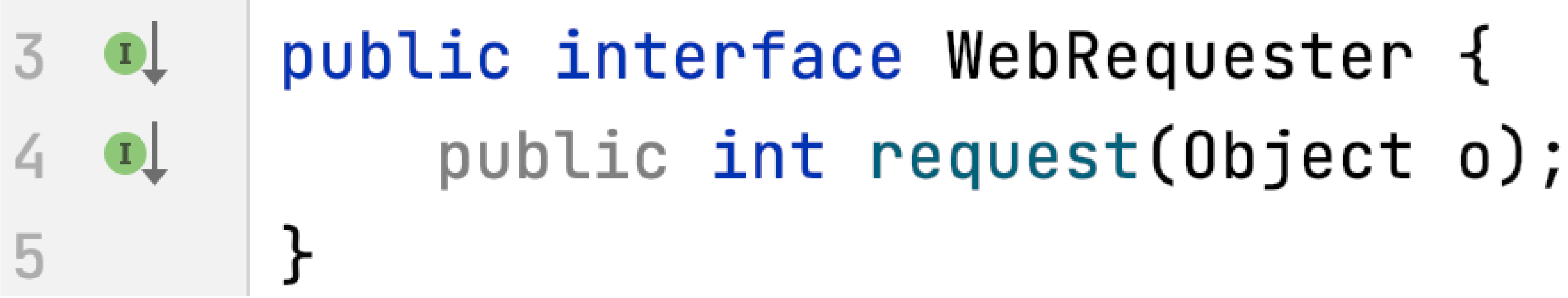
Adapter pattern

- Implementation of an adapter design pattern:
  - Design the target interface
  - Implement the target interface with the adapter class
  - Send the request from the client to the adapter using target interface

Example

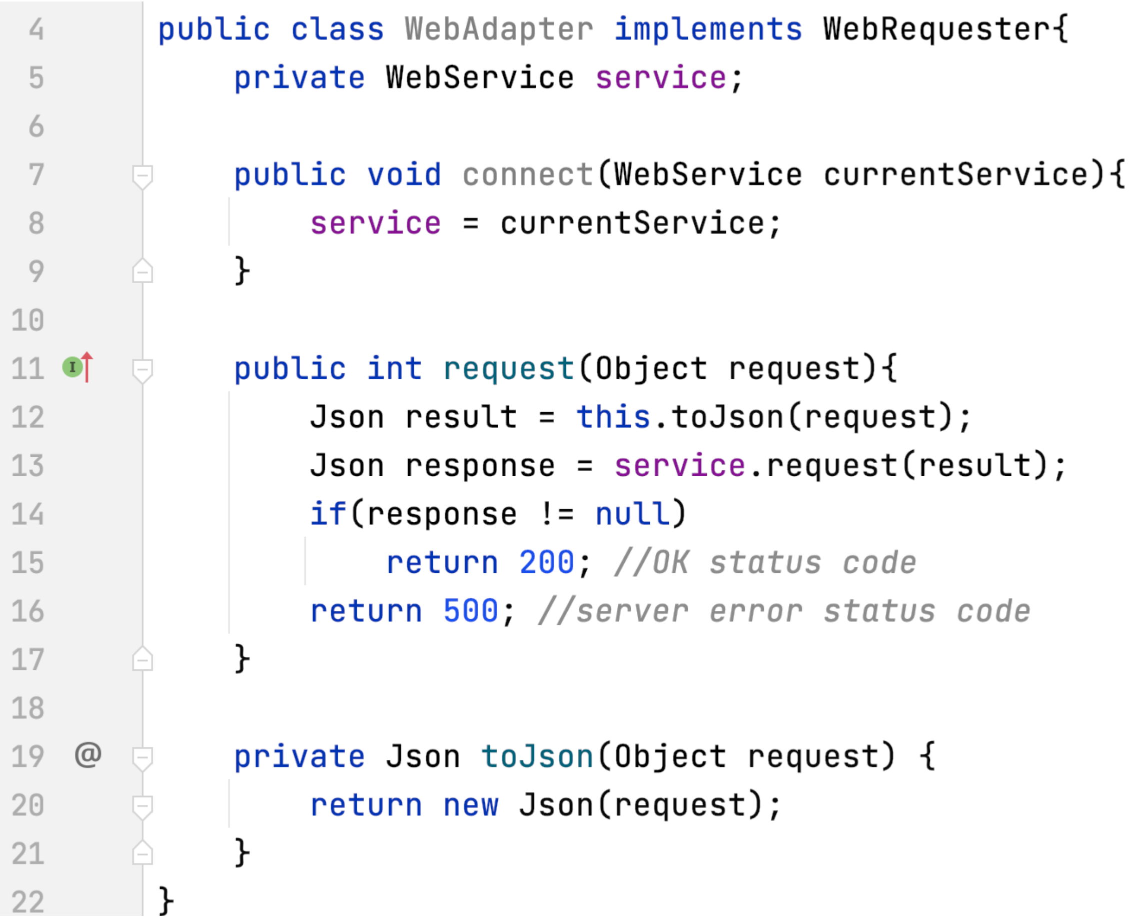

«interface»
WebRequester
+ request(Object): int
WebClient
- makeObject(): Object
+ doWork(): void
WebService
+ request(Json): Json
WebAdapter
- service: WebService
+ request(Object): int
+ connect(WebService): void
- toJson(Object): Json

Step 1: Design the target interface



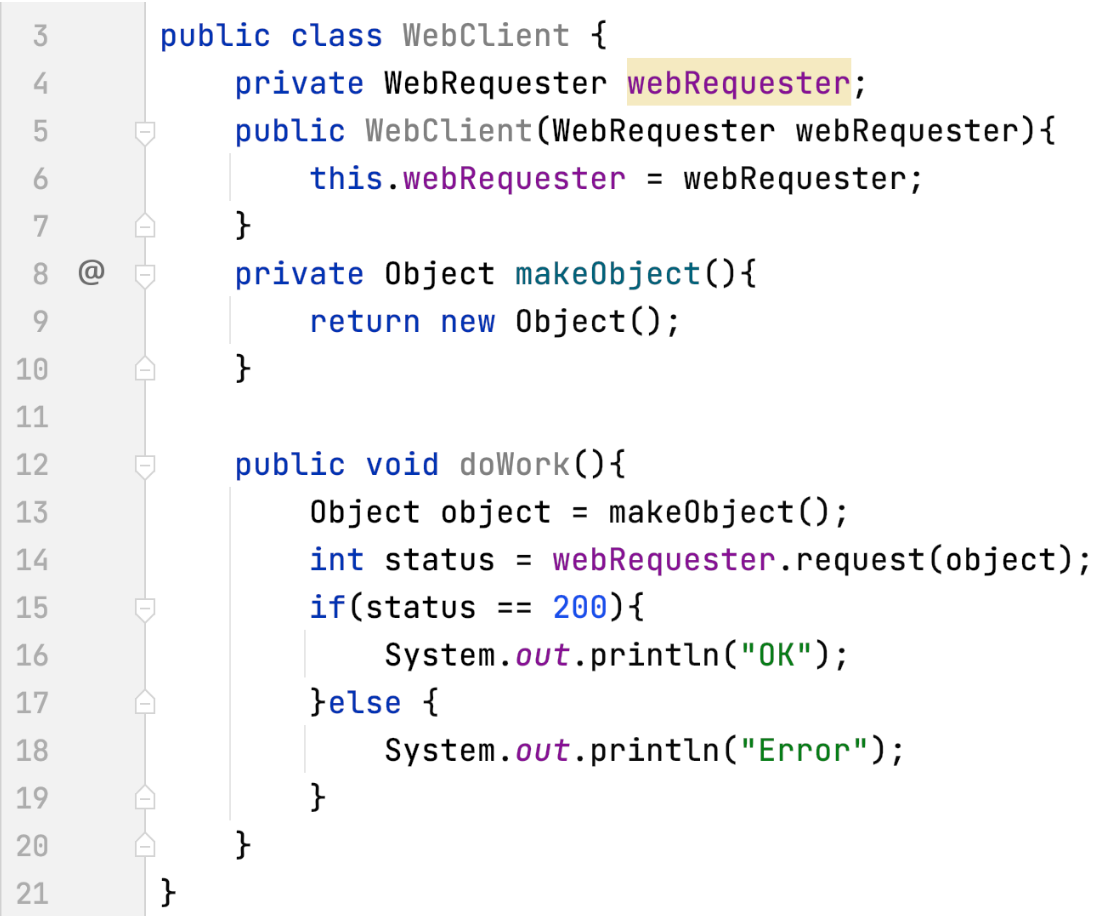
```java
public interface WebRequester {
    public int request(Object o);
}
```

Step 2: Implement the target interface with the adapter class



```java
public class WebAdapter implements WebRequester{
    private WebService service;

    public void connect(WebService currentService){
        service = currentService;
    }

    public int request(Object request){
        Json result = this.toJson(request);
        Json response = service.request(result);
        if(response != null)
            return 200; //OK status code
        return 500; //server error status code
    }

    private Json toJson(Object request) {
        return new Json(request);
    }
}
```

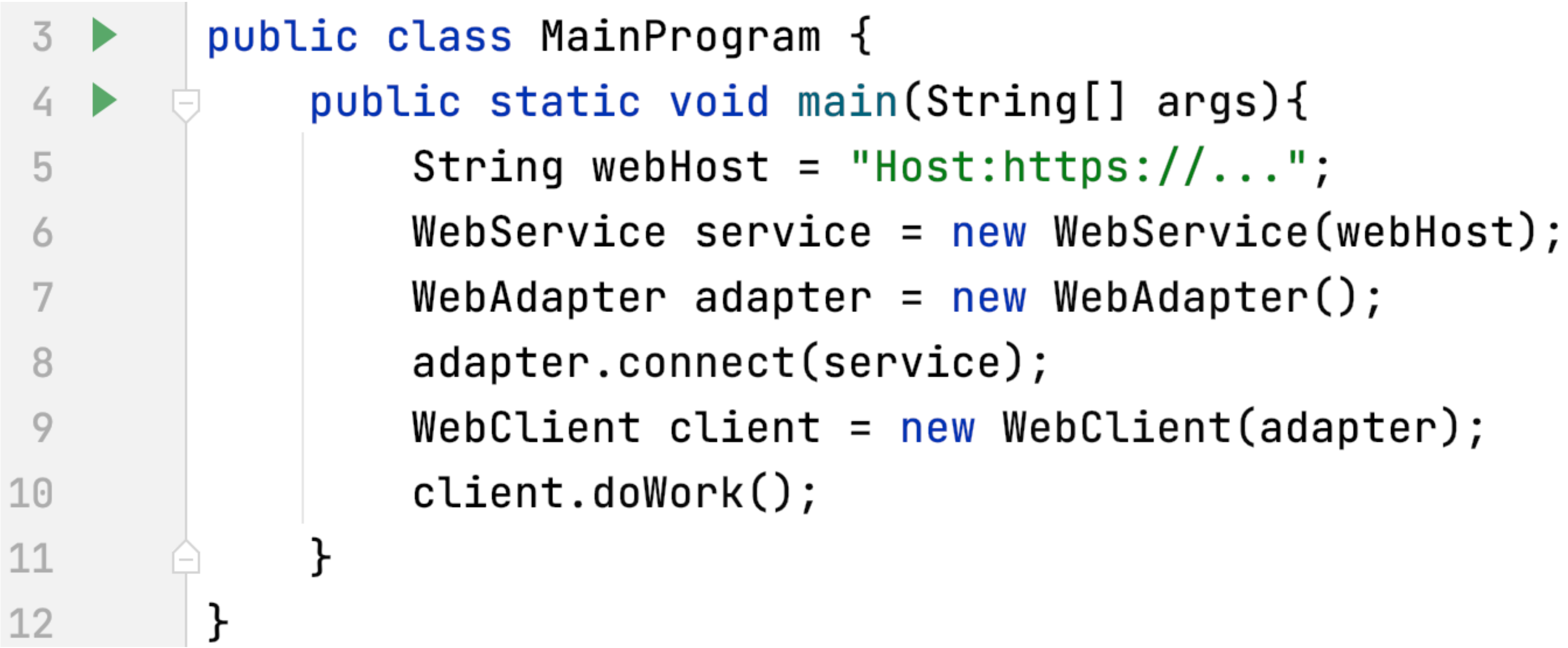
Step 3: Send the request from the client to the adapter using the target interface



```java
public class WebClient {
    private WebRequester webRequester;
    public WebClient(WebRequester webRequester){
        this.webRequester = webRequester;
    }
    private Object makeObject(){
        return new Object();
    }

    public void doWork(){
        Object object = makeObject();
        int status = webRequester.request(object);
        if(status == 200){
            System.out.println("OK");
        }else {
            System.out.println("Error");
        }
    }
}
```

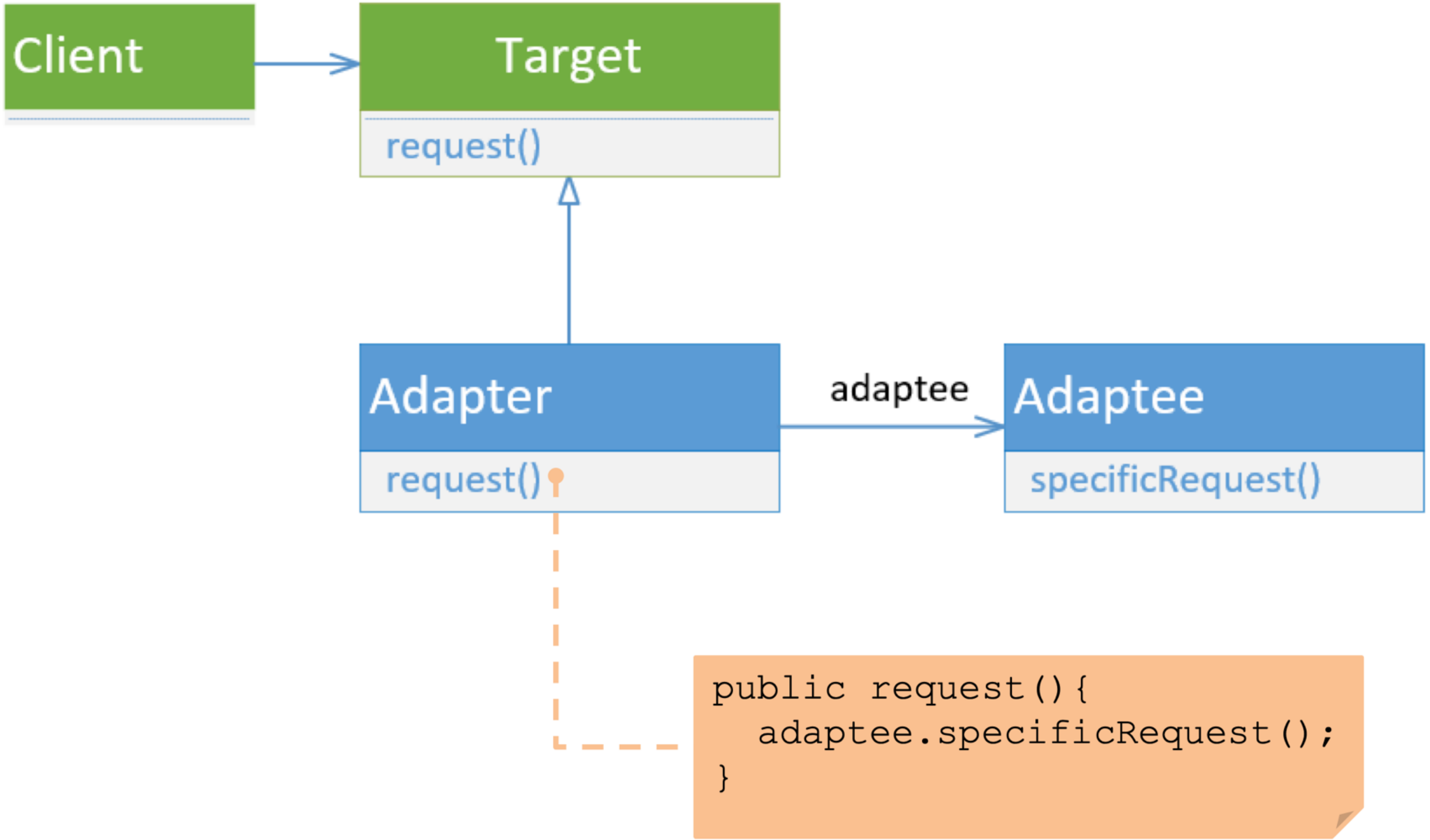
Main program

In the main program: **WebAdapter**, **WebService**, and **WebClient** need to be instantiated



```java
public class MainProgram {
    public static void main(String[] args){
        String webHost = "Host:https://...";
        WebService service = new WebService(webHost);
        WebAdapter adapter = new WebAdapter();
        adapter.connect(service);
        WebClient client = new WebClient(adapter);
        client.doWork();
    }
}
```

Adapter



```
public request(){
  adaptee.specificRequest();
}
```

Façade pattern

Façade pattern

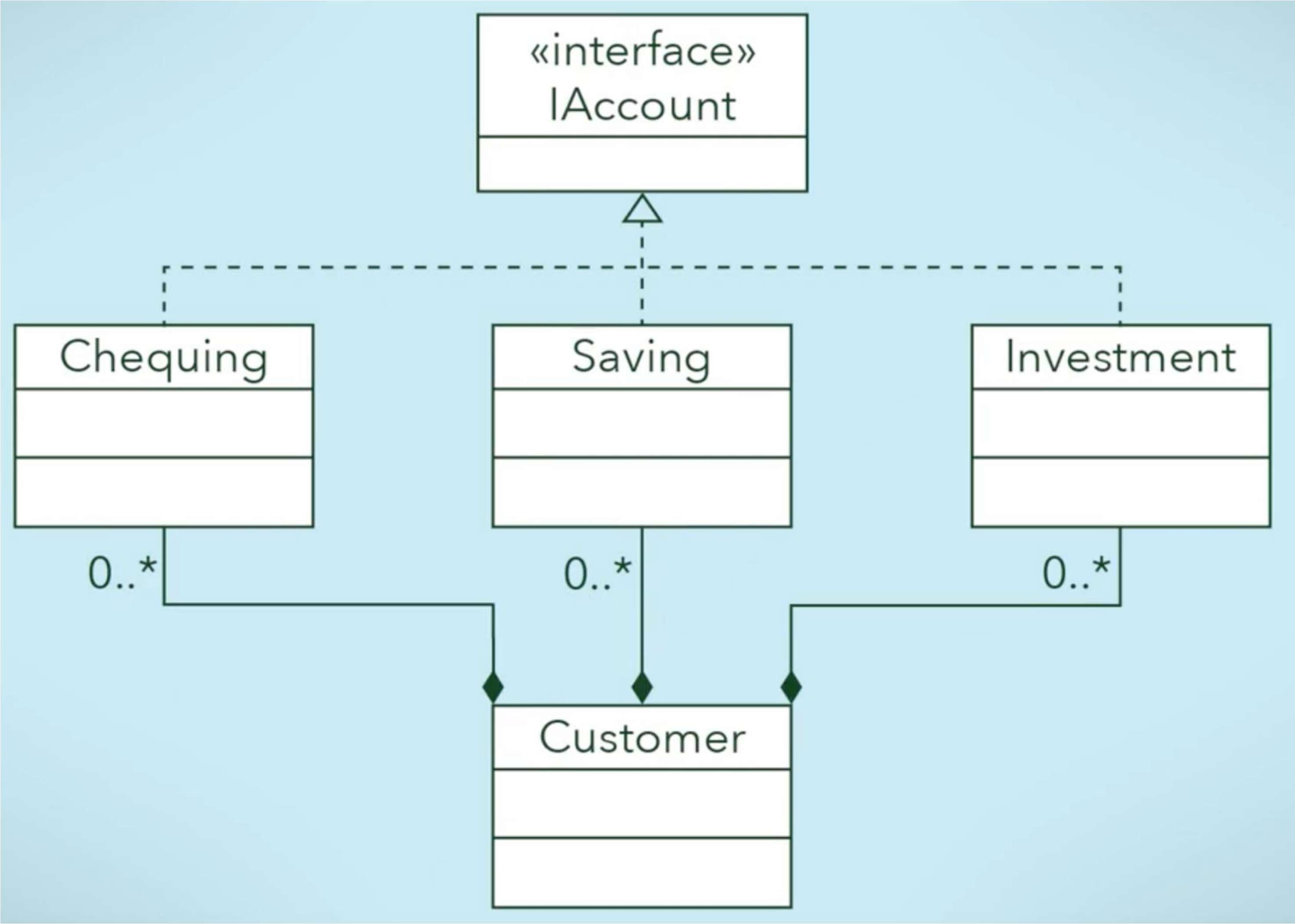
- Façade pattern provides a single, simplified interface for client classes to interact with a subsystem.
- A façade is a wrapper class that encapsulates a subsystem in order to hide the subsystem’s complexity.

Façade pattern

- Façade design pattern can be explained through a number of steps:
  - Design the interface
  - Implement the interface with one or more classes
  - Create the façade class and wrap the classes that implement the interface
  - Use the façade class to access the subsystem

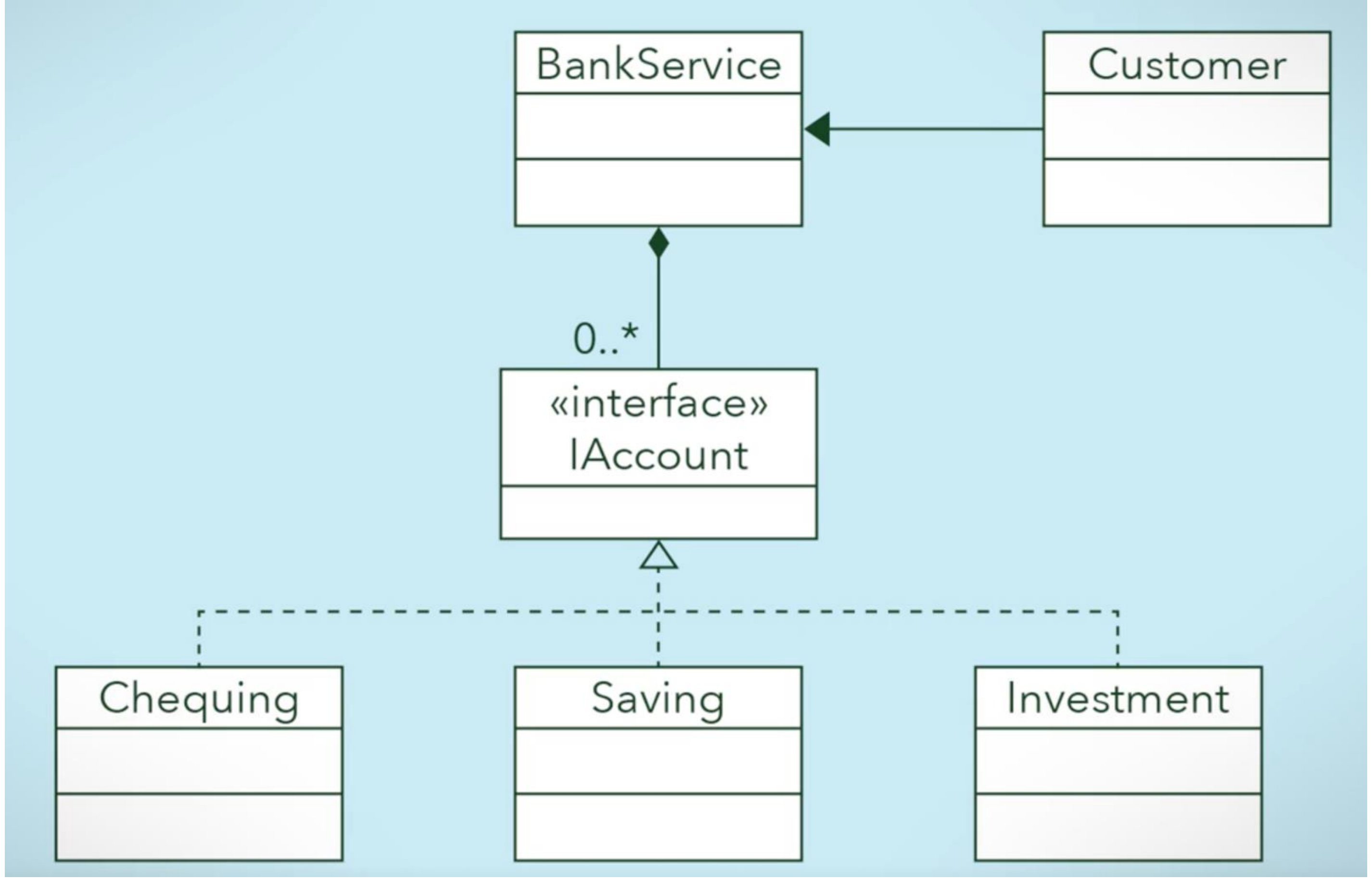
Example

Without façade class, the **Customer** class is responsible for properly creating instances of each class **Chequing**, **Saving**, **Investment**.

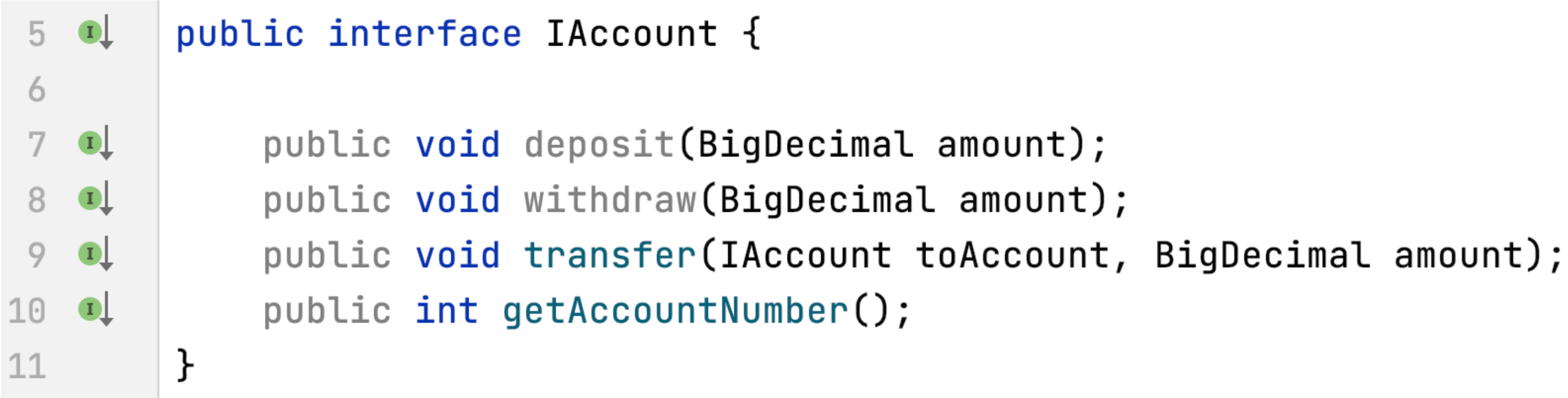


Example

BankService is a façade class, Presents a simpler front to the subsystem for the customer client class to use.

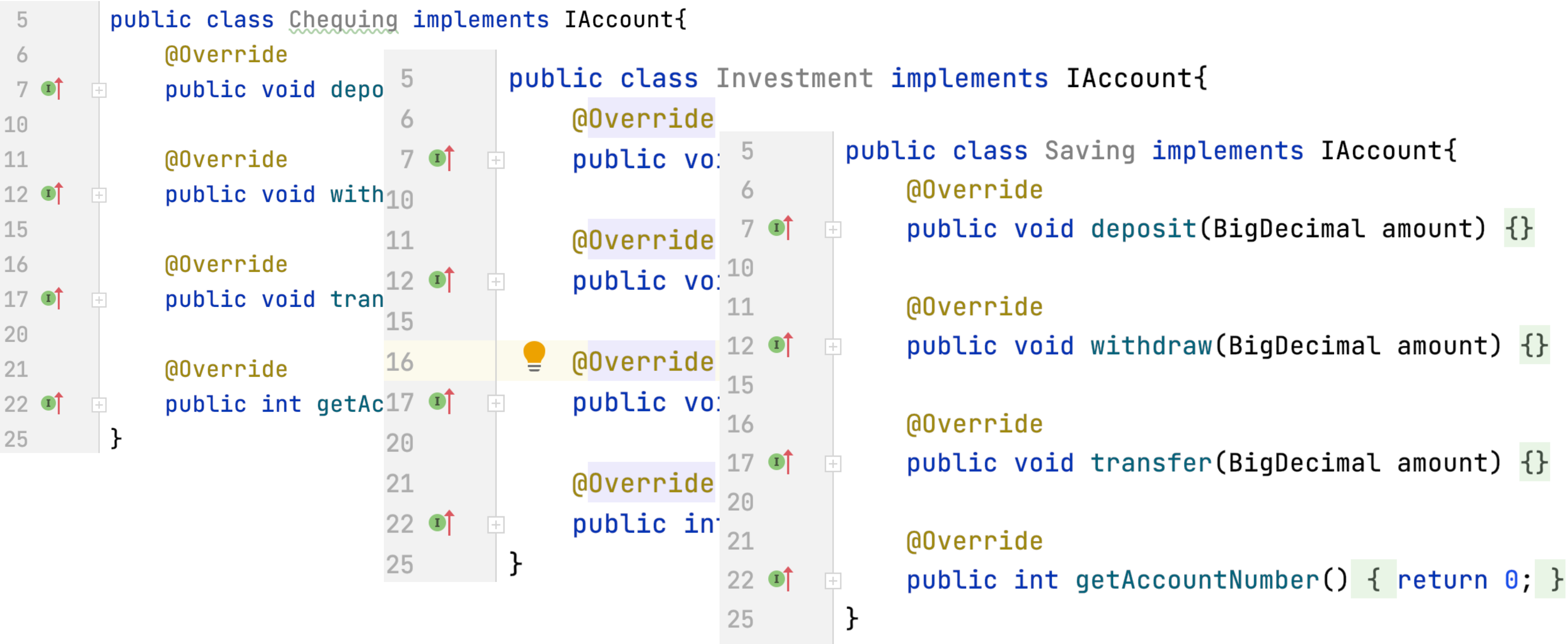


Step 1: Design the interface



```java
public interface IAccount {

    public void deposit(BigDecimal amount);
    public void withdraw(BigDecimal amount);
    public void transfer(IAccount toAccount, BigDecimal amount);
    public int getAccountNumber();
}
```

Step 2: Implement the interface with one or more classes

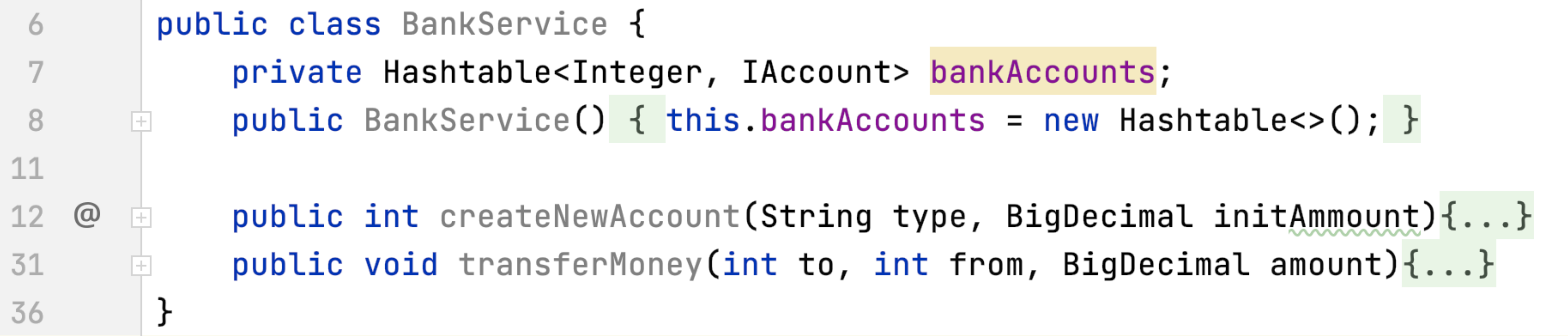


```
public class Chequing implements IAccount{
    @Override
    public void depo

    @Override
    public void with

    @Override
    public void tran

    @Override
    public int getAc
}
```

```
public class Investment implements IAccount{
    @Override
    public voi

    @Override
    public voi

    @Override
    public voi

    @Override
    public in
}
```

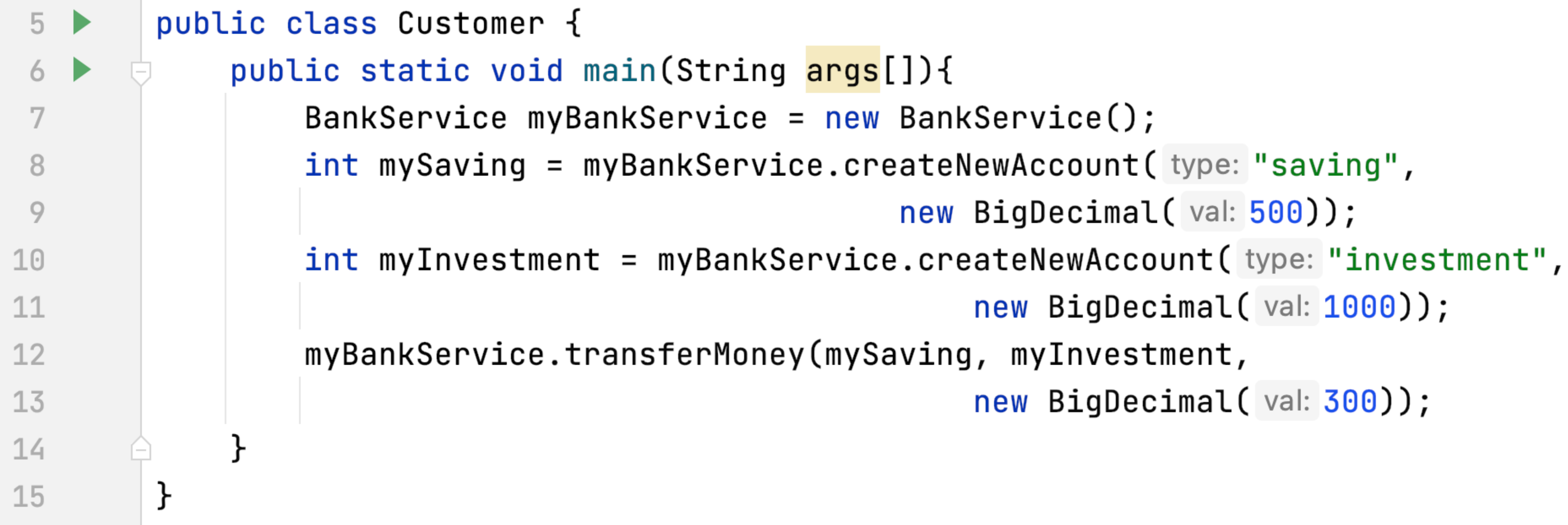
```
public class Saving implements IAccount{
    @Override
    public void deposit(BigDecimal amount) {}

    @Override
    public void withdraw(BigDecimal amount) {}

    @Override
    public void transfer(BigDecimal amount) {}

    @Override
    public int getAccountNumber() { return 0; }
}
```

Step 3: Create the façade class



```java
public class BankService {
    private Hashtable<Integer, IAccount> bankAccounts;
    public BankService() { this.bankAccounts = new Hashtable<>(); }

    public int createNewAccount(String type, BigDecimal initAmmount){...}
    public void transferMoney(int to, int from, BigDecimal amount){...}
}
```

Step 4: Use the façade class to access the subsystem



```java
public class Customer {
    public static void main(String args[]){
        BankService myBankService = new BankService();
        int mySaving = myBankService.createNewAccount( type: "saving",
                                                new BigDecimal( val: 500));
        int myInvestment = myBankService.createNewAccount( type: "investment",
                                                    new BigDecimal( val: 1000));
        myBankService.transferMoney(mySaving, myInvestment,
                                                    new BigDecimal( val: 300));
    }
}
```