

Java Deep Basics

Java Versions

- Java 11
 - String: `isBlank()` method, `repeat()` method, `lines()` method
 - `IsEmpty()` method on Optional Class:
 - It has a better garbage collection system due to garbage collectors like ZGC and Epsilon.
 - Var keyword is available in Java 11,
 - New methods like `writeString()`, `readString()` to work with files
 - Deprecations of some plugins and Applet api
 - Applications are faster and more secure
- Java 17
 - Sealed classes: This helps in creating a more secure and maintainable codebase by restricting which classes can be subclasses.
`public sealed class Shape permits Circle, Rectangle { // ... }`
 - Records: Records are a special kind of class in Java designed for modeling immutable data objects.
 - Provides a more expressive and flexible syntax for switch statements and expressions
 - Pattern matching for 'instanceof' operator
 - Several security updates

Java Memory Management

- Java memory management is a crucial aspect of the Java runtime environment, ensuring efficient use of memory resources and facilitating automatic garbage collection

Java Memory Areas

- Heap:
 - The largest memory area used for dynamic memory allocation for Java objects.
 - Divided into:
 - ◆ Young Generation: Where new objects are allocated and aged. Further divided into Eden space, Survivor space (S0 and S1).
 - ◆ Old Generation: Where long-lived objects are stored after surviving multiple garbage collection cycles in the Young Generation.
 - ◆ String const pool outside heap but part I think
- Stack:
 - Stores method frames and local variables for each thread.
 - Each thread has its own stack, allowing for method calls and local variable storage.
- Metaspace:

- Replaced PermGen in Java 8.
- Dynamically resizable and stores class metadata.
- Code Cache:
 - Stores compiled bytecode and native code generated by the Just-In-Time (JIT) compiler.

Key Concepts

- Garbage Collection:
 - Automatic process of reclaiming memory by identifying and disposing of objects that are no longer reachable.
 - Types of Garbage Collectors:
 - ◆ Serial Garbage Collector: Uses a single thread for garbage collection, suitable for small applications.
 - ◆ Parallel Garbage Collector: Uses multiple threads for garbage collection, suitable for multi-threaded applications.
 - ◆ CMS (Concurrent Mark-Sweep) Garbage Collector: Focuses on minimizing pause times by doing most of the work concurrently with application threads.
 - ◆ G1 (Garbage-First) Garbage Collector: Designed for large heaps with a focus on predictable pause times.
 - Reference Types:
 - ◆ Strong References: The default reference type. Objects with strong references are not eligible for garbage collection.
 - ◆ Soft References: Used for memory-sensitive caches. Objects are collected only when the JVM is about to run out of memory.
 - ◆ Weak References: Used for canonicalized mappings, such as in weak hash maps. Objects are collected as soon as they are weakly reachable.
 - ◆ Phantom References: Used for scheduling post-mortem cleanup actions.

Memory Management Phases

- Allocation:
 - Memory is allocated on the heap for new objects using the new keyword.
- Promotion:
 - Objects are initially allocated in the Young Generation. If they survive multiple garbage collection cycles, they are promoted to the Old Generation.
- Garbage Collection Cycles:
 - Minor GC: Cleans up the Young Generation. Typically short and frequent.
 - Major GC (Full GC): Cleans up the Old Generation and may also include the Young Generation. More time-consuming than Minor GC.
- Garbage Collection Process
 - Mark: Identifies which objects are still in use.
 - Sweep: Deletes unused objects and reclaims their memory.
 - Compact: Optional step to reduce fragmentation by moving objects and updating references.

- Best Practices:
 - Optimize object creation and destruction to reduce garbage collection overhead.
 - Use appropriate garbage collectors based on application requirements.
 - Monitor memory usage and GC performance using tools like VisualVM, JConsole, or GC logs.
 - Avoid memory leaks by ensuring objects are dereferenced when no longer needed.
 - Use memory-efficient data structures and algorithms.

Java's memory management, primarily through its automatic garbage collection mechanism, relieves developers from manual memory management tasks, reducing the likelihood of memory leaks and enhancing application reliability.

Java Generics

- Generics enable the creation of code that works with different data types while providing compile-time type safety
- Java generics provide a way to define classes, interfaces, and methods with type parameters `<T>`, allowing for type-safe operations and eliminating the need for explicit type casting.
- Type Parameters: Represented by angle brackets `<>` and typically single uppercase letters (e.g., T, E, K, V).
- There can be:
 - Generic classes: `public class Pair<K,V> { .. }`
 - Generic methods: `public <T> void printArray(T[] array) {.. for(T element: array) ..}`
 - Wildcards: `public void processList(List<?> list) {..}`

Advantages of Java Generics

- Type Safety:
 - Compile-time type checking reduces the risk of `ClassCastException`.
- Code Reusability:
 - Generic code can be reused with different types, reducing code duplication.
- Elimination of Explicit Casts:
 - Generics eliminate the need for explicit type casting, making the code cleaner and less error-prone.
- Eg:
 - Collections Framework extensively uses generics to provide type-safe collections.


```
List<String> stringList = new ArrayList<>();
stringList.add("Hello");
String s = stringList.get(0); // No casting needed
```

- Interfaces can also be generic

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Limitations of Java Generics

➤ Type Erasure:

- Java generics use type erasure to maintain backward compatibility with older versions. This means generic type information is erased at runtime, and only the raw type remains.

```
List<String> stringList = new ArrayList<>();
List<Integer> intList = new ArrayList<>();
if (stringList.getClass() == intList.getClass()) {
    System.out.println("Types are erased to the same raw type.");
}
```

➤ Cannot Create Instances of Type Parameters:

- Due to type erasure, you cannot create instances of a generic type parameter.

```
public class GenericClass<T> {
    // T obj = new T(); // Compile-time error
}
```

➤ No Primitive Types:

- Generics work only with reference types, not primitive types. You need to use wrapper classes instead.

```
List<Integer> intList = new ArrayList<>(); // Cannot use List<int>
```

Autoboxing and Unboxing

- *Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, or a double to a Double.

- *Unboxing* is the reverse process where the Java compiler automatically converts the wrapper class to its corresponding primitive type.

```
public class AutoboxingExample {  
    public static void main(String[] args) {  
        // Autoboxing: converting a primitive int to an Integer object  
        Integer integerObject = 10;  
  
        // Unboxing: converting an Integer object to a primitive int  
        int primitiveInt = integerObject;  
  
        // Autoboxing in a collection  
        List<Integer> integerList = new ArrayList<>();  
        integerList.add(5); // Autoboxing: 5 is converted to Integer.valueOf(5)  
  
        // Unboxing during retrieval  
        int value = integerList.get(0); // Unboxing: integerList.get(0) returns an Integer  
  
        // Printing the results  
        System.out.println("Autoboxed Integer: " + integerObject);  
        System.out.println("Unboxed int: " + primitiveInt);  
        System.out.println("Value from List: " + value);  
    }  
}
```

Benefits

- Simplifies Code:
 - Autoboxing and unboxing make the code more readable and easier to write by eliminating the need for explicit conversion between primitive types and their corresponding wrapper objects.
- Integration with Collections:
 - Collections in Java can only hold objects, not primitive types. Autoboxing allows for easy storage of primitive types in collections without manual conversion.
- Cleaner Code:
 - Reduces boilerplate code by handling the conversion automatically.

Drawbacks

- Performance Overhead:
 - Frequent autoboxing and unboxing can introduce performance overhead due to the creation of unnecessary wrapper objects and the extra time spent on conversion.
- Null Pointer Exceptions:
 - Unboxing a null reference can lead to a `NullPointerException`.

```
Integer integerObject = null;
int primitiveInt = integerObject; // Throws NullPointerException
```

➤ Unexpected Behavior:

- Equality checks between wrapper objects can lead to unexpected results because == checks reference equality for objects.
- Java Integer Caching: Java caches Integer objects for values between -128 and 127. This means that Integer.valueOf(127) will return the same object from the cache. This can lead to unexpected results when using == for comparison. For values outside this range, new Integer objects are created.

```
Integer x = 127;
Integer y = 127;
System.out.println(x == y); // true, same object from cache

Integer m = 128;
Integer n = 128;
System.out.println(m == n); // false, different objects
```

Logging Mechanisms

- Java logging frameworks are essential for monitoring and debugging applications. They provide a way to record runtime information, errors, and system behavior.
- Popular Java Logging Frameworks
 - Java Util Logging (JUL)
 - ◆ Java Util Logging (JUL) is the logging API provided by the JDK.
 - ◆ Part of the Java Standard Library (no external dependencies).
 - ◆ Use this when simple logging
 - Apache Log4j 2 (configuration below)
 - ◆ Log4j 2 is a popular and powerful logging framework from the Apache Software Foundation.
 - ◆ Asynchronous logging for high performance.
 - ◆ Highly configurable via XML, JSON, YAML, or properties files.
 - ◆ Supports advanced filtering and custom log levels.
 - ◆ Choose this for more complex applications requiring high performance and flexibility
 - SLF4J (Simple Logging Facade for Java)

- ◆ SLF4J is a facade for various logging frameworks, providing a simple and unified logging API.
- ◆ Choose this if you want to decouple your application from the underlying logging framework, use SLF4J with Logback or Log4j 2.
- Logback
 - ◆ Logback is designed by the same creator of Log4j and is intended as a successor, providing a faster and more robust logging framework.
 - ◆ High performance and reliability.
- Best Practices:
 - Use a Facade: Use SLF4J as the logging API to allow flexibility in choosing the underlying logging framework.
 - Log Levels: Appropriately use log levels (TRACE, DEBUG, INFO, WARN, ERROR) to categorize log messages.
 - Configuration Management: Externalize logging configurations to easily modify them without changing the code.
 - Performance: Avoid excessive logging in performance-critical sections and consider asynchronous logging for high throughput applications.
 - Security: Be cautious about logging sensitive information to avoid security risks.

Log4j 2 Logging framework

Steps to implement it in our code:

- Add Log4j 2 Dependencies to pom.xml
 - Add the Log4j 2 dependencies to your Maven pom.xml file.
 - You'll typically need log4j-api and log4j-core as the essential dependencies.
- Create Log4j 2 Configuration File
 - Log4j 2 supports several configuration formats: XML, JSON, YAML, and properties files.
 - Here, using property file

2014.5.18

Properties Configuration (`log4j2.properties`)

If you prefer to use a properties file instead of XML:

properties

Copy code

```
status = warn
name = PropertiesConfig

appender.console.type = Console
appender.console.name = ConsoleAppender
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n

appender.file.type = File
appender.file.name = FileAppender
appender.file.fileName = logs/app.log
appender.file.layout.type = PatternLayout
appender.file.layout.pattern = %d{ISO8601} [%t] %-5p %c{1}:%L - %m%n


logger.example.name = com.example
logger.example.level = debug
logger.example.additivity = false
logger.example.appenderRef.console.ref = ConsoleAppender
logger.example.appenderRef.file.ref = FileAppender

rootLogger.level = info
rootLogger.appenderRef.console.ref = ConsoleAppender
```

- Use Log4j 2 in Your Code
 - With the dependencies and configuration in place, you can now use Log4j 2 in your Java code

2014.5.18

java

 Copy code

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4j2Example {
    // Create a logger instance
    private static final Logger logger = LogManager.getLogger(Log4j2Example.class);

    public static void main(String[] args) {
        logger.trace("This is a TRACE message");
        logger.debug("This is a DEBUG message");
        logger.info("This is an INFO message");
        logger.warn("This is a WARN message");
        logger.error("This is an ERROR message");
        logger.fatal("This is a FATAL message");
    }
}
```

➤ Running the Application

- When you run your Java application, Log4j 2 will read the configuration file (log4j2.properties), and log messages will be output according to the configuration settings.
- In this example, log messages are printed to the console and also written to a file (logs/app.log).

Java Design Principles

1. SOLID Principles

- SOLID is an acronym for five design principles intended to make software designs more understandable, flexible, and maintainable
- Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle

2. DRY (Don't Repeat Yourself)

- Avoid unnecessary repetition in code

3. KISS (Keep It Simple, Stupid)

- Eg: instead of multiple if else, use switch statement

4. YAGNI (You Ain't Gonna Need It)

- If you are adding add method, no need to add subtract, multiply etc

5. Composition Over Inheritance

- Prefer composition over inheritance to achieve code reuse.
- Like instead of extending/creating sub class; just declare the parent class object as a field member of the supposed child class

6. Favor Immutability

- Immutable objects are easier to design, implement, and use. They are

inherently thread-safe and have fewer side effects.

7. Law of Demeter (LoD)

- Each unit/class should only talk to its immediate friends and not to strangers. This principle encourages loose coupling.

8. Encapsulation

- Encapsulation is the bundling of data and methods that operate on the data within one unit, e.g., a class. This hides the internal state of the object from the outside world and only exposes a controlled interface.

SOLID Principles

1. Single Responsibility Principle (SRP)

- A class should have only one reason to change, meaning it should have only one job or responsibility.

```
// Violation of SRP
class User {
    void getUserDetails() { /*...*/ }
    void saveUserToDatabase() { /*...*/ }
}

// Adhering to SRP
class User {
    void getUserDetails() { /*...*/ }
}

class UserRepository {
    void saveUserToDatabase(User user) { /*...*/ }
}
```

2. Open/Closed Principle (OCP)

- Software entities should be open for extension but closed for modification.

```
// Violation of OCP
class Rectangle {
    int width, height;
    int area() { return width * height; }
}

// Adhering to OCP
abstract class Shape {
    abstract int area();
}

class Rectangle extends Shape {
    int width, height;
    int area() { return width * height; }
}

class Circle extends Shape {
    int radius;
    int area() { return (int) (Math.PI * radius * radius); }
}
```

3. Liskov Substitution Principle (LSP)

- A child class should be able to substitute its parent class without affecting the correctness of the program.
- Here, it is violation, as Ostrich can't be a bird fully as it can't fly.

```
// Adhering to LSP
class Bird {
    void fly() { /*...*/ }
}

class Sparrow extends Bird { }

class Ostrich extends Bird {
    @Override
    void fly() { throw new UnsupportedOperationException("Ostrich can't fly"); }
}
```

4. Interface Segregation Principle (ISP)

- Clients should not be forced to depend on interfaces they do not use.
- So larger interfaces should be split into smaller ones

```
// Violation of ISP
interface Worker {
    void work();
    void eat();
}

class HumanWorker implements Worker {
    void work() { /*...*/ }
    void eat() { /*...*/ }
}

class RobotWorker implements Worker {
    void work() { /*...*/ }
    void eat() { throw new UnsupportedOperationException(); }
}
```

```
// Adhering to ISP
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class HumanWorker implements Workable, Eatable {
    void work() { /*...*/ }
    void eat() { /*...*/ }
}

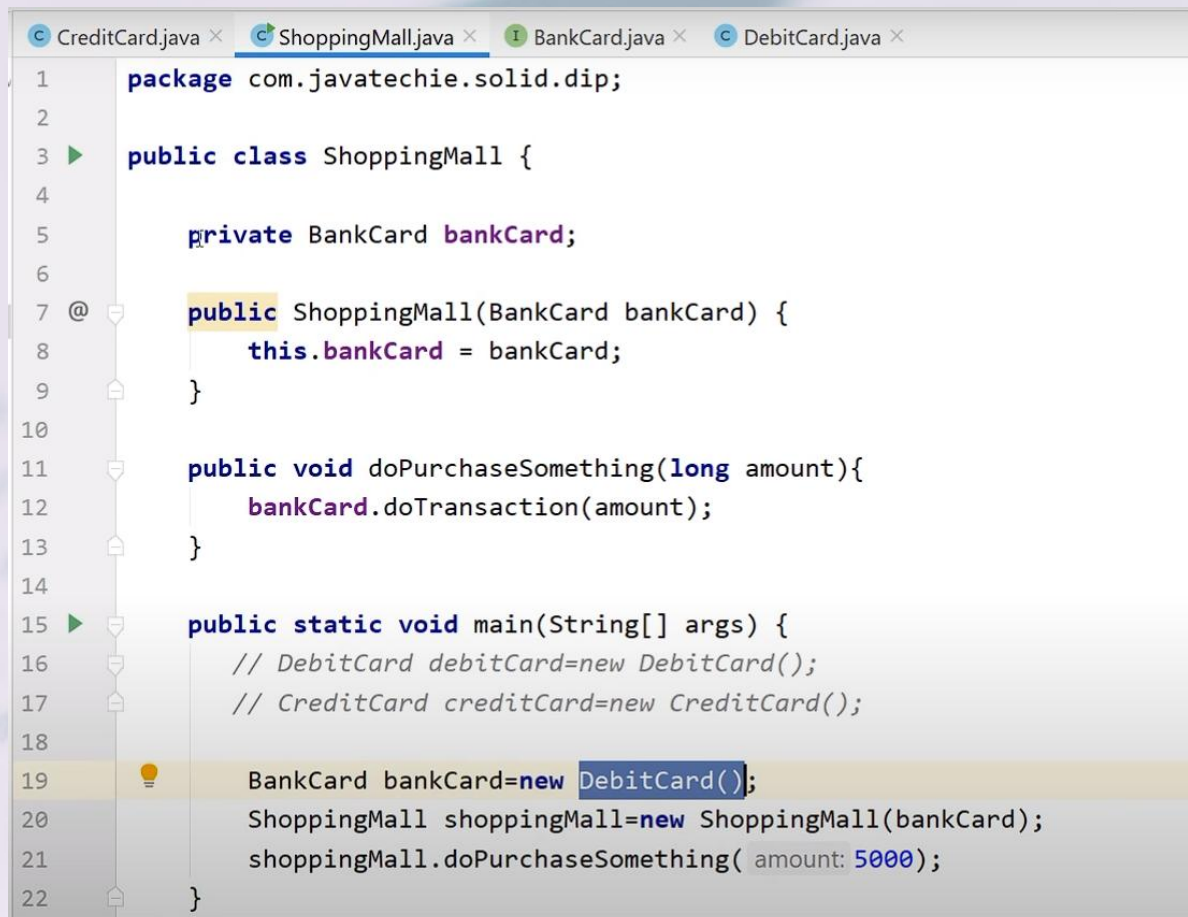
class RobotWorker implements Workable {
    void work() { /*...*/ }
}
```

5. Dependency Inversion Principle (DIP)

- Basically Depend upon abstractions/interfaces and not upon concrete classes
- High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces).
- Abstractions should not depend on details. Details should depend on abstractions.
- Example:
 - Initially our ShoppingMall class had to separately call the classes 'CreditCard' or 'DebitCard' depending on which card we have to use, so it was tightly coupled and we had to make much changes in the ShoppingMall class constructors, variable etc
 - Implementing DIP, we added Bankcard Interface which we can then

implement with the CreditCard and DebitCard classes

- This way, our ShoppingMall class has to be changed just once to include the interface and then won't be changed later, we just have to create object of the respective card in our main program using the interface (line: 19) depending on credit or debit
- Thus, dependency has been moved from Credit/Debit card classes to just the BankCard interface



```
1 package com.javatechie.solid.dip;
2
3 public class ShoppingMall {
4
5     private BankCard bankCard;
6
7     public ShoppingMall(BankCard bankCard) {
8         this.bankCard = bankCard;
9     }
10
11     public void doPurchaseSomething(long amount){
12         bankCard.doTransaction(amount);
13     }
14
15     public static void main(String[] args) {
16         // DebitCard debitCard=new DebitCard();
17         // CreditCard creditCard=new CreditCard();
18
19         BankCard bankCard=new DebitCard();
20         ShoppingMall shoppingMall=new ShoppingMall(bankCard);
21         shoppingMall.doPurchaseSomething( amount: 5000);
22     }
```

2014.5.18

Java Design Patterns

Design patterns are tried and tested solutions to common problems; well documented and understood by all; language neutral; avoids technical risk during software development life cycle

1. Creational Patterns
 - a) Singleton
 - b) Prototype
 - c) Factory Method
 - d) Abstract Factory
 - e) Builder
2. Structural
3. Behavioral
4. Concurrency
5. Architectural
 - a) MVC (Model-View-Controller) Pattern
 - b) MVP (Model-View-Presenter) Pattern
 - c) MVVM (Model-View-ViewModel) Pattern

Creational Pattern

Deals with the way of creating objects. Eg: new keyword in java

1. Singleton Pattern
 - Ensures that a class has only one instance and provides a global point of access to that instance.
2. Factory Pattern
 - Create a factory class and using it in client/main class to create the objects of child classes which are inherited from parent interface
3. Abstract Factory Pattern
 - Similar to above except here it's multi levelled; you have a factory for creating factories which create child class objects.
4. Builder Pattern
 - Focuses on constructing a complex object from simple objects using step by step approach
5. Prototype Pattern:
 - Creates new objects by copying/cloning an existing object, thus avoiding the overhead of creating objects from scratch.
 - Used when creation of new object is needed but without costing

Singleton Design Pattern

- A singleton class is a class that can have only one object/instance at a time
- Object creation only takes place once, so new creation requests refer to the previous object only leading to memory efficiency
- Spring uses singleton beans as default and multi threaded apps use this single object repeatedly for logging, caching, configuration etc
- To create a singleton class
 - Private constructors
 - Private Static variable: Instance (object)
 - Static method that return a reference to the above instance (getInstance)
- Eg:

```
3 //this is the default lazy singleton initialization
4 class SingletonSample {
5     private static SingletonSample instance; //creating private static object variable
6
7     private SingletonSample(){} //private constructor
8
9     public static SingletonSample getInstance() { //public static method to create/get object
10         if(instance == null) {
11             instance = new SingletonSample();
12         }
13
14         return instance;
15     }
16 }
```

In main(), creating object, use the getInstance() method

```
64 public static void main(String[] args) {
65
66     SingletonSample instance1 = SingletonSample.getInstance();
67     System.out.println(instance1.hashCode());
68     SingletonSample instance2 = SingletonSample.getInstance();
69     System.out.println(instance2.hashCode()); //same hashcode
70 }
```

Also, Eager Initialization

```
19 //this is the eager singleton initialization
20 class SingletonEager {
21     private static SingletonEager instance = new SingletonEager(); //no if condition, creates obj everytime
22     private SingletonEager(){}
23
24     public static SingletonEager getInstance() { return instance; }
25 }
26
27
28
29
30 //using synchronized method (just adding synchronized keyword in getInstance method)
31 class SingletonSynchronizedMethod {
32     private static SingletonSynchronizedMethod instance;
33     private SingletonSynchronizedMethod(){}
34
35     public static synchronized SingletonSynchronizedMethod getInstance() {
36         if(instance == null) {
37             instance = new SingletonSynchronizedMethod();
38         }
39         return instance;
40     }
41 }
42 }
```

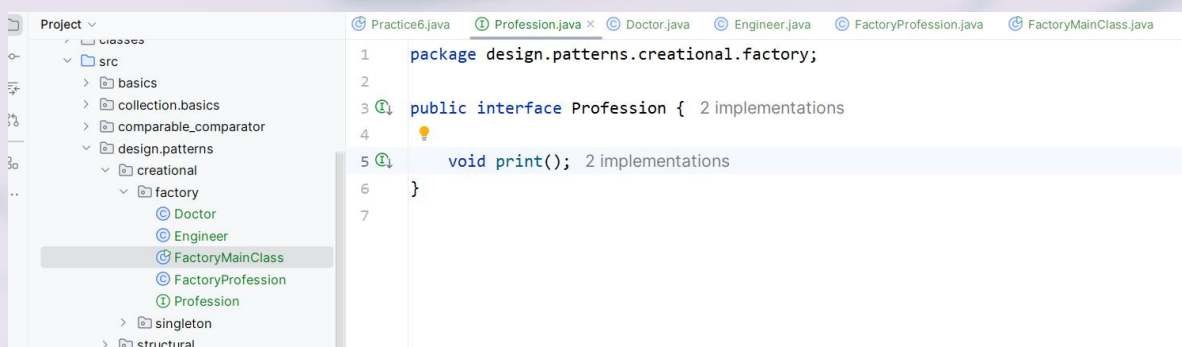

Note:

- Singleton class can be broken through serialization, cloning, reflection
- For those details, refer the Eclipse/IntelliJ code

Factory Design Pattern

- In factory design pattern, we don't expose the creation logic to the client
- We have an interface and concrete classes implementing it
- We create a 'Factory class' which creates and returns the type of object of the concrete classes to the client/main class. (Client is unaware how object is created, whether new keyword or autowire, he doesn't know)

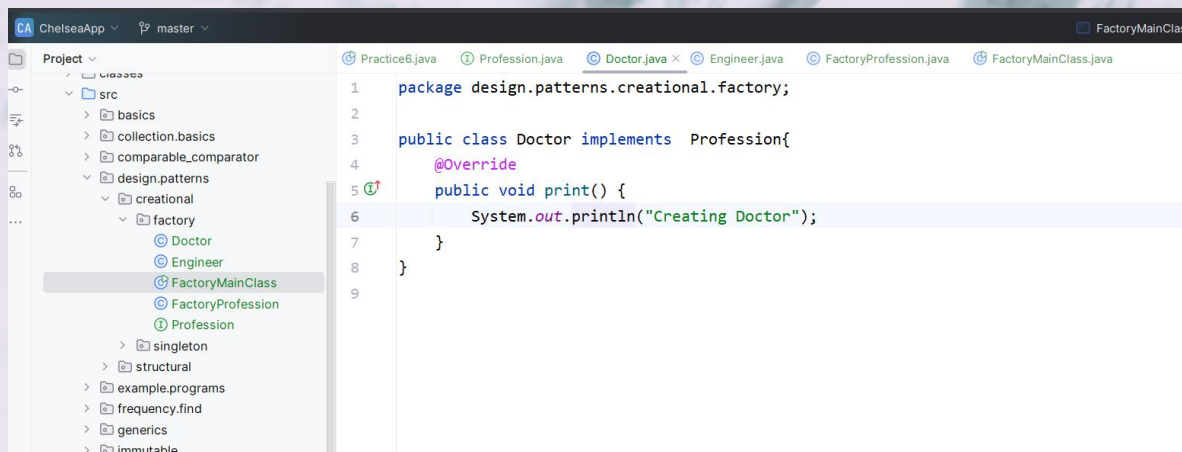
Parent Interface:



The screenshot shows the IntelliJ IDEA interface. On the left, the Project Explorer displays a project structure with a package hierarchy: design.patterns.creational.factory. The main editor shows the code for the Profession interface in Profession.java:

```
1 package design.patterns.creational.factory;
2
3 public interface Profession { 2 implementations
4
5     void print(); 2 implementations
6
7 }
```

Concrete child classes

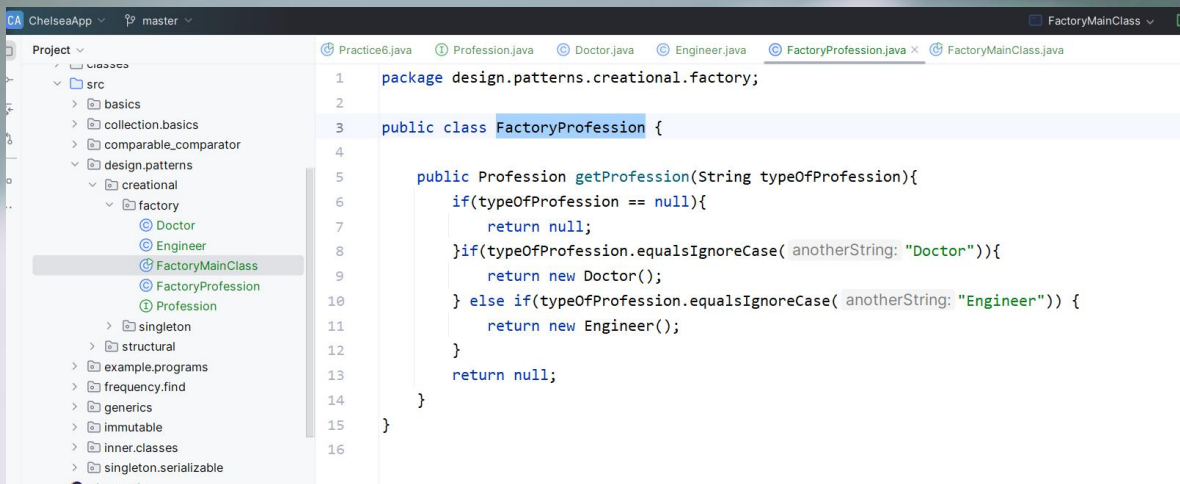


The screenshot shows the IntelliJ IDEA interface. On the left, the Project Explorer displays the same project structure as before. The main editor shows the code for the Doctor class in Doctor.java, which implements the Profession interface:

```
1 package design.patterns.creational.factory;
2
3 public class Doctor implements Profession{
4     @Override
5     public void print() {
6         System.out.println("Creating Doctor");
7     }
8 }
9
```

Our Factory class

2014.5.18



```
1 package design.patterns.creational.factory;
2
3 public class FactoryProfessional {
4
5     public Profession getProfession(String typeOfProfession){
6         if(typeOfProfession == null){
7             return null;
8         }if(typeOfProfession.equalsIgnoreCase( anotherString: "Doctor")){
9             return new Doctor();
10        } else if(typeOfProfession.equalsIgnoreCase( anotherString: "Engineer")) {
11            return new Engineer();
12        }
13        return null;
14    }
15 }
16
```

Using our Factory class in our client/main class



```
1 package design.patterns.creational.factory;
2
3 public class FactoryMainClass {
4     public static void main(String[] args) {
5         FactoryProfessional factoryProfessional = new FactoryProfessional();
6
7         Profession doc = factoryProfessional.getProfession( typeOfProfession: "Doctor");
8         doc.print();
9     }
10 }
11
```

Run FactoryMainClass

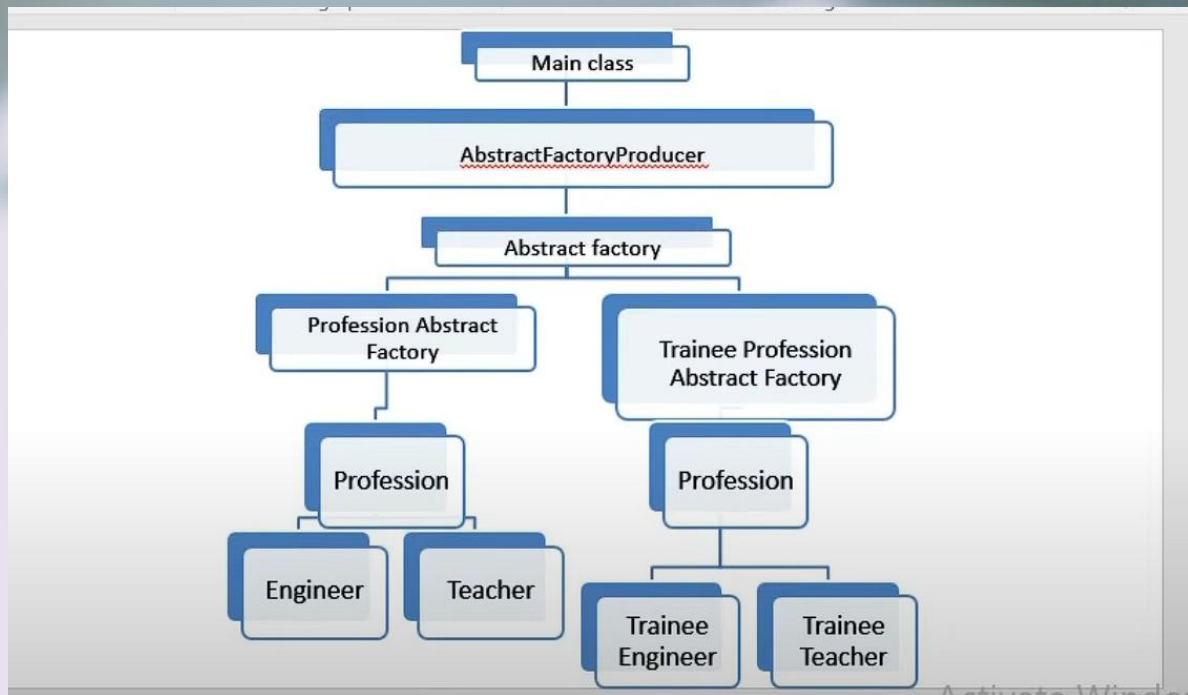
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.1.4\lib\id" "Creating Doctor"

Process finished with exit code 0

Abstract Factory Design Pattern

- Similar to Factory design pattern except here the process is multi levelled; you have a factory for creating factories which create child class objects of the implemented parent interface
- Also called 'Kit'

2014.5.18



- Here we have innermost factory classes for creating trainee or full profession
- Then the inner abstract factory class is abstract and its method is also abstract and is extended in above classes
- Outer producer factory class calls the above inner abstract factory class depending on the input type required

Parent interface and child classes same like before (extra child classes for Engineer and Trainee Engineer)

Now, Innermost class for creating object

```

package design.patterns.creational.abstractFactory;

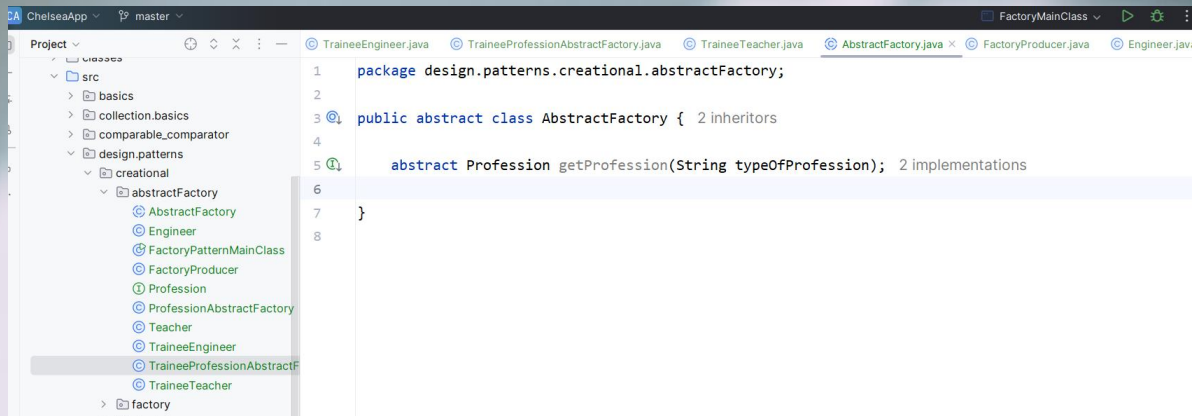
//also create 1 for full profession type extending AbstractFactory
public class TraineeProfessionAbstractFactory extends AbstractFactory{

    @Override
    public Profession getProfession(String typeOfProfession){

        if(typeOfProfession == null){
            return null;
        }
        else if(typeOfProfession.equalsIgnoreCase( anotherString: "Engineer")){
            return new TraineeEngineer();
        }
        else if(typeOfProfession.equalsIgnoreCase( anotherString: "Teacher")){
            return new TraineeTeacher();
        }

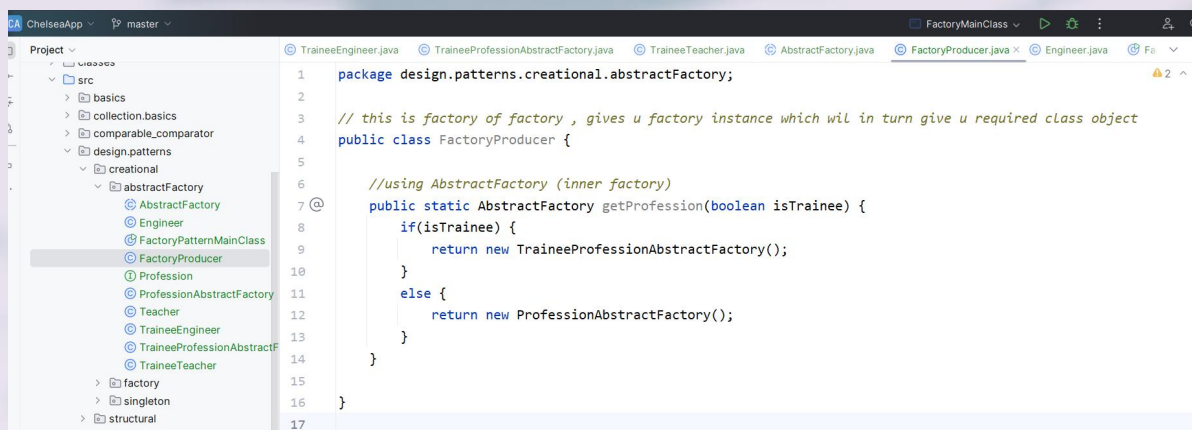
        return null;
    }
}
  
```


Inner 'Abstract' class



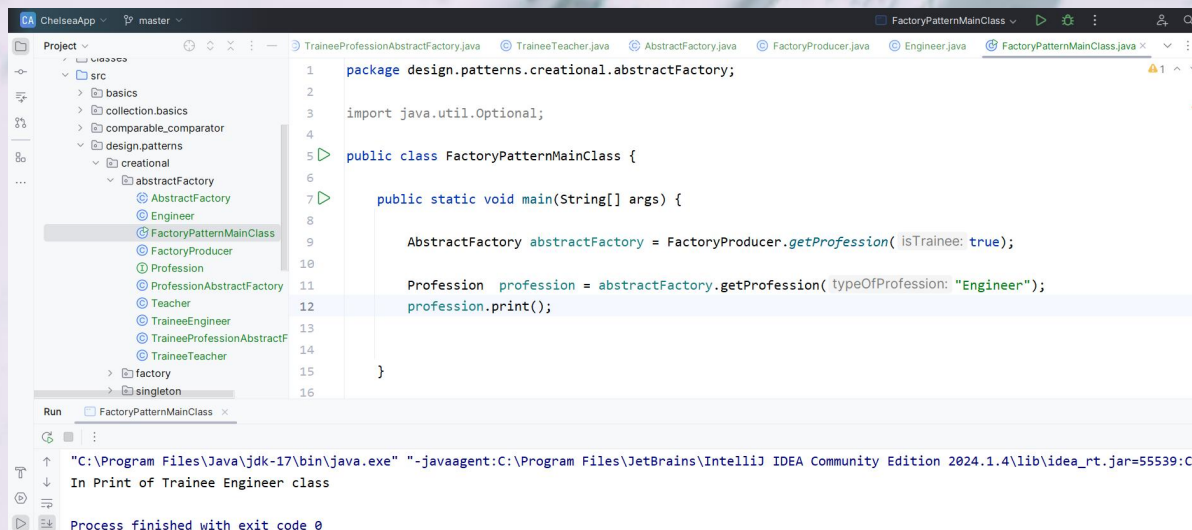
```
1 package design.patterns.creational.abstractFactory;
2
3 public abstract class AbstractFactory { 2 inheritors
4
5     abstract Profession getProfession(String typeOfProfession); 2 implementations
6
7 }
8
```

Outer Factory Producer class



```
1 package design.patterns.creational.abstractFactory;
2
3 // this is factory of factory , gives u factory instance which wil in turn give u required class object
4 public class FactoryProducer {
5
6     //using AbstractFactory (inner factory)
7     public static AbstractFactory getProfession(boolean isTrainee) {
8         if(isTrainee) {
9             return new TraineeProfessionAbstractFactory();
10        }
11        else {
12            return new ProfessionAbstractFactory();
13        }
14    }
15 }
16
17
```

Main/client class



```
1 package design.patterns.creational.abstractFactory;
2
3 import java.util.Optional;
4
5 public class FactoryPatternMainClass {
6
7     public static void main(String[] args) {
8
9         AbstractFactory abstractFactory = FactoryProducer.getProfession( isTrainee: true);
10
11         Profession profession = abstractFactory.getProfession( typeOfProfession: "Engineer");
12         profession.print();
13     }
14 }
15
16
```

Run FactoryPatternMainClass

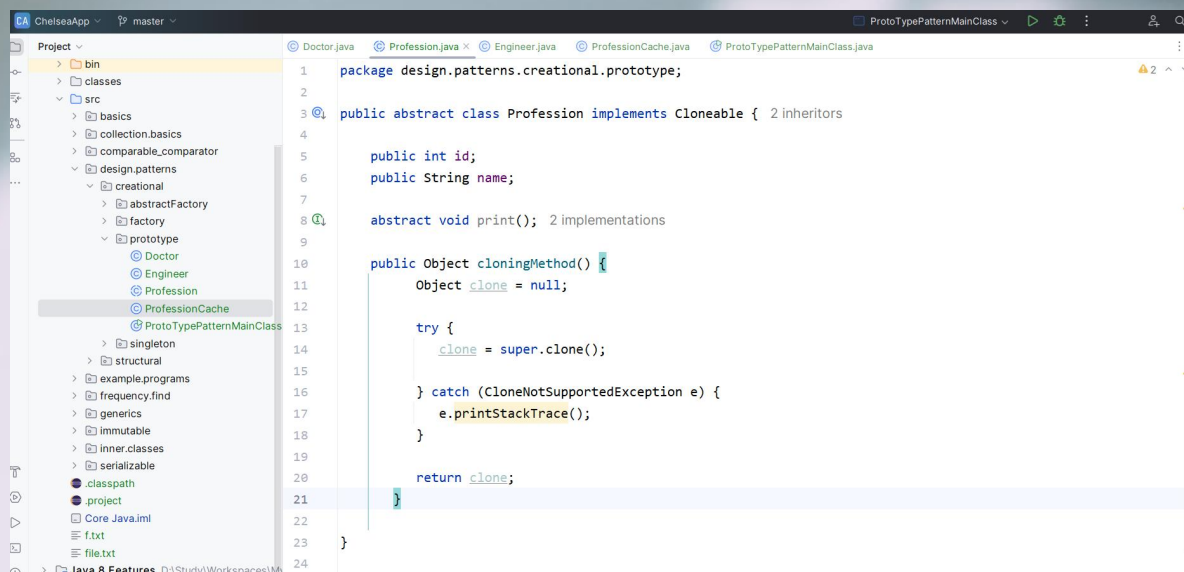
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.1.4\lib\idea_rt.jar=55539:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.1.4\bin" -Didea.config.path=C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.1.4\config In Print of Trainee Engineer class

Process finished with exit code 0

Prototype Design Pattern

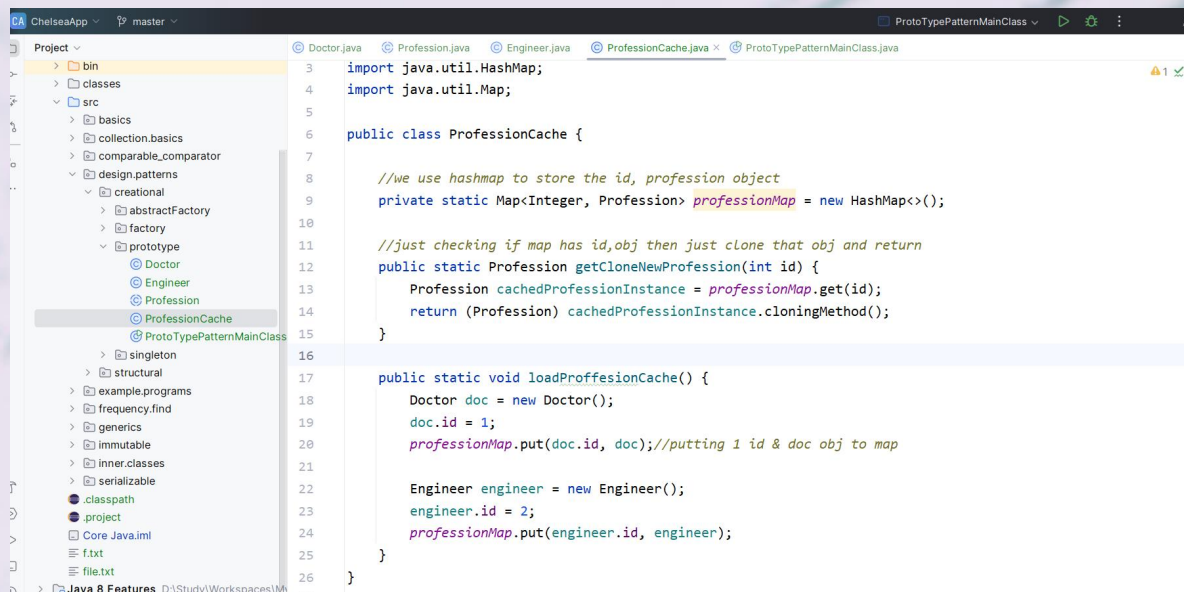
- Creates new objects by copying/cloning an existing object, thus avoiding the overhead of creating objects from scratch.
- Used when creation of new object is needed but without costing
- Here, instead of parent interface, we have abstract class which implements Cloneable interface to use the super.clone() method to copy object

Abstract parent class overriding cloning method of Cloneable interface



```
1 package design.patterns.creational.prototype;
2
3 @
4
5 public abstract class Profession implements Cloneable { 2 inheritors
6
7     public int id;
8     public String name;
9
10    abstract void print(); 2 implementations
11
12    public Object cloningMethod() {
13        Object clone = null;
14
15        try {
16            clone = super.clone();
17        } catch (CloneNotSupportedException e) {
18            e.printStackTrace();
19        }
20
21        return clone;
22    }
23
24 }
```

Using hashmap to act as cache and then using parent abstract class's overridden cloningMethod() to duplicate the obj whenever required



```
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class ProfessionCache {
7
8     //we use hashmap to store the id, profession object
9     private static Map<Integer, Profession> professionMap = new HashMap<>();
10
11     //just checking if map has id,obj then just clone that obj and return
12     public static Profession getCloneNewProfession(int id) {
13         Profession cachedProfessionInstance = professionMap.get(id);
14         return (Profession) cachedProfessionInstance.cloningMethod();
15     }
16
17     public static void loadProfessionCache() {
18         Doctor doc = new Doctor();
19         doc.id = 1;
20         professionMap.put(doc.id, doc); //putting 1 id & doc obj to map
21
22         Engineer engineer = new Engineer();
23         engineer.id = 2;
24         professionMap.put(engineer.id, engineer);
25     }
26 }
```

Main/client class

2014.5.18

```
package design.patterns.creational.prototype;

public class ProtoTypePatternMainClass {

    public static void main(String[] args) {
        ProfessionCache.loadProffesionCache();

        Profession docProfession = ProfessionCache.getCloneNewProfession( id: 1);
        System.out.println(docProfession);

        Profession engProfession = ProfessionCache.getCloneNewProfession( id: 2);
        System.out.println(engProfession);

        Profession docProfession2 = ProfessionCache.getCloneNewProfession( id: 1);
        System.out.println(docProfession2); //hashcode will be different
    }
}
```

"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.1.4\lib\design.patterns.creational.prototype.Doctor@4c873330
design.patterns.creational.prototype.Engineer@119d7047
design.patterns.creational.prototype.Doctor@776ec8df
Process finished with exit code 0

Builder Design Pattern

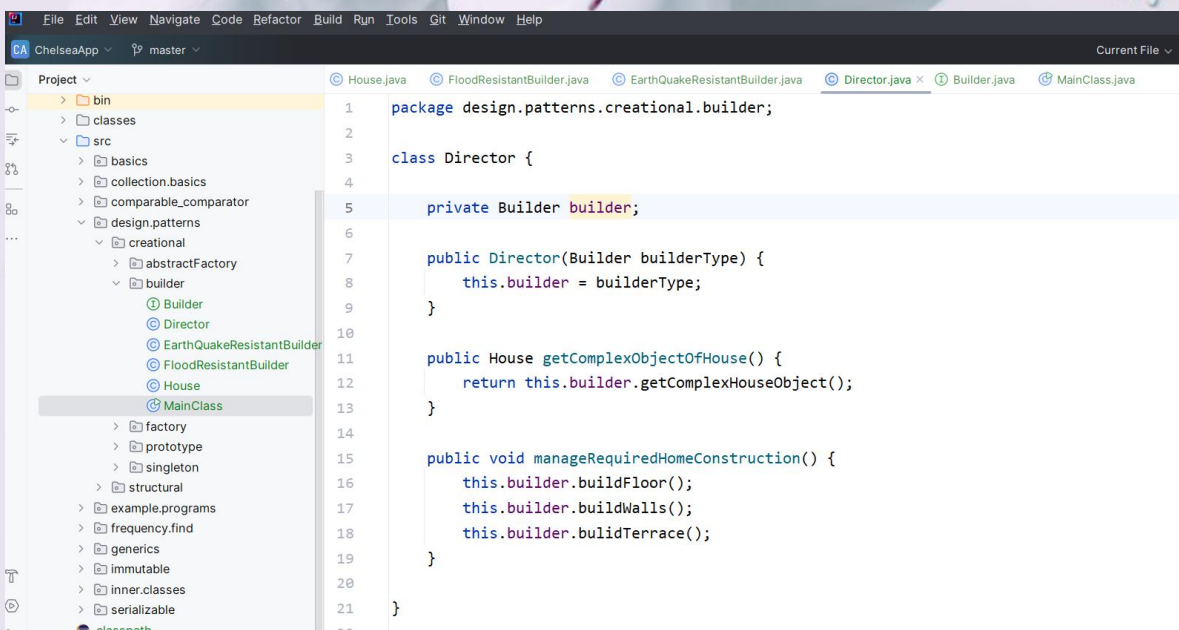
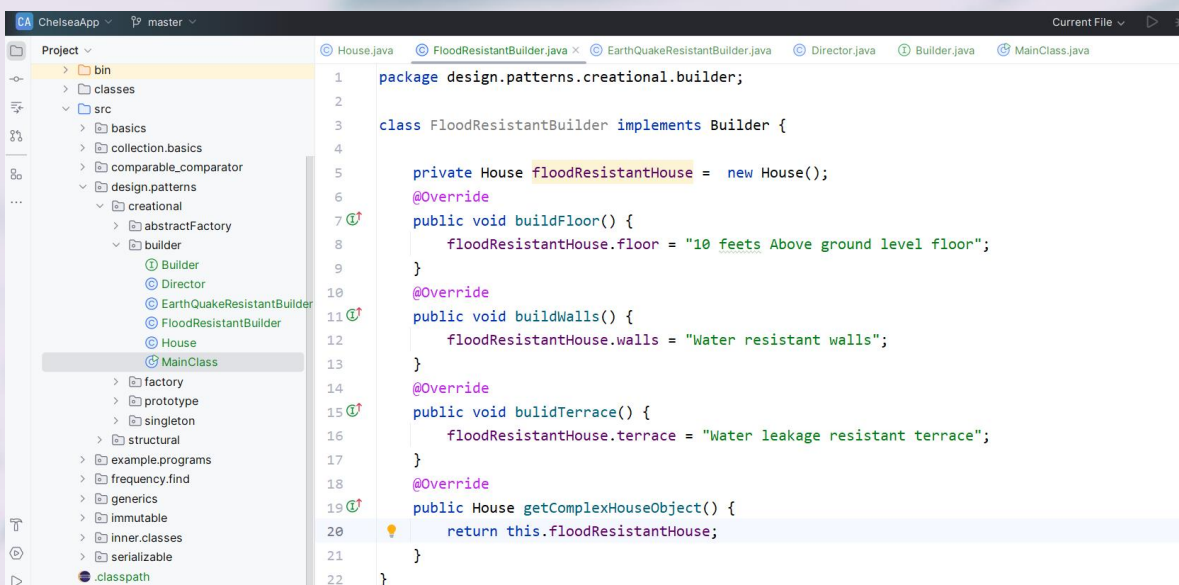
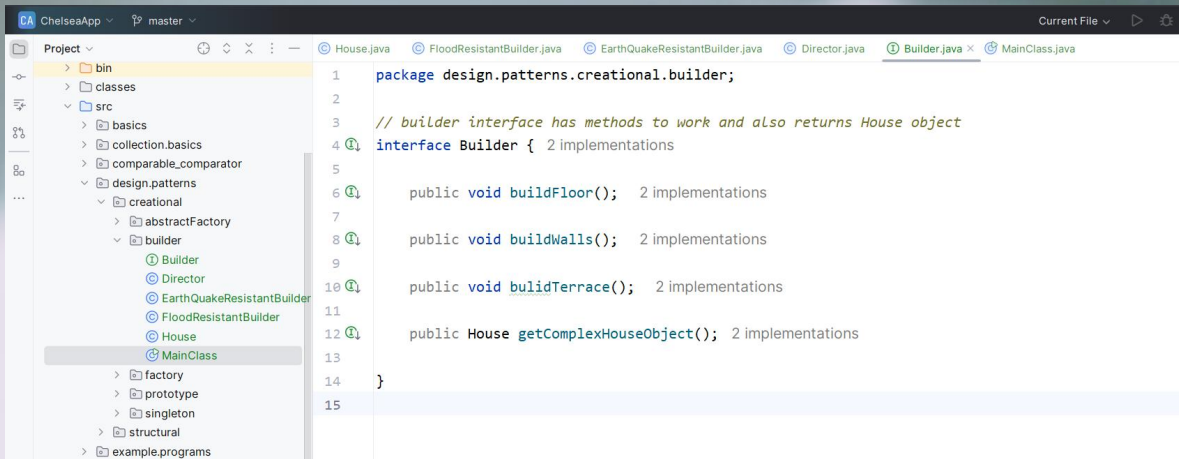
- Focuses on constructing a complex object from simple objects using step by step approach
- Major roles are:
 - Complex object - Eg: House object which we will generate
 - Builder interface - It contains all methods to build our complex object house (floor,walls,terrace) and also return the final house complex object
 - Concrete Builder classes - It is the implementation of the Builder interface and has classes to give different complex objects. Eg: Flood resistant builder, Earthquake resistant builder
 - Director class - It controls all the object creation
 - ◆ It calls the exact concrete builder classes to create our complex object
 - ◆ It returns the final complex house object created

```
package design.patterns.creational.builder;

//complex house object
public class House {

    public String floor;
    public String walls;
    public String terrace;
}
```

design.patterns.creational.builder.House@119d7047
design.patterns.creational.builder.House@119d7047
design.patterns.creational.builder.House@119d7047
Process finished with exit code 0



```
public class MainClass {  
    public static void main(String[] args) {  
        //create object of required house builder  
        EarthQuakeResistantBuilder earthQuakeResistantBuilder = new EarthQuakeResistantBuilder();  
        //create object of director that will keep an eye on your builder  
        Director director = new Director(earthQuakeResistantBuilder);  
        director.manageRequiredHomeConstruction();  
        House houseConstructedAtTheEnd = director.getComplexObjectOfHouse();  
        System.out.println(houseConstructedAtTheEnd);  
        System.out.println(houseConstructedAtTheEnd.floor);  
    }  
}
```

Run MainClass x

"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.1.4\lib\idea_rt.jar=54305:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.1.4\bin" -Dfile.encoding=UTF-8
design.patterns.builder.House@4c873330
Wooden floor
Process finished with exit code 0



记得记住。很多人
样的人，要相信这
最终会成为你面对

2014.5.18