Advanced Programming

→ **Design Patterns:**

Creational Design Patterns

1.
Design Patterns

Design patterns

- Best practices to solve problems.
- Solutions to general problems that software developers faced during software development.
- Not to repeat the same mistakes over and over again.
- Design patterns are programming language independent strategies for solving the common object-oriented design problems.
- Design pattern represents an idea, not a particular implementation.

Design Patterns Categories

Creational

Deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Structural

Deal with how classes and objects are arranged or composed.

Behavioral

Describe how classes and objects interact and communicate with each other.

Concrete class

- Is a class that has an implementation for all of its methods.
- Cannot have any unimplemented methods.
- Can extend an abstract class or implement an interface as long as it implements all their methods.
- It is a complete class and can be instantiated..

→ we can say that any class which is not abstract is a concrete class.

2.
Creational Design Patterns

Creational Design patterns

- Factory
- Prototype
- Singleton
- Builder

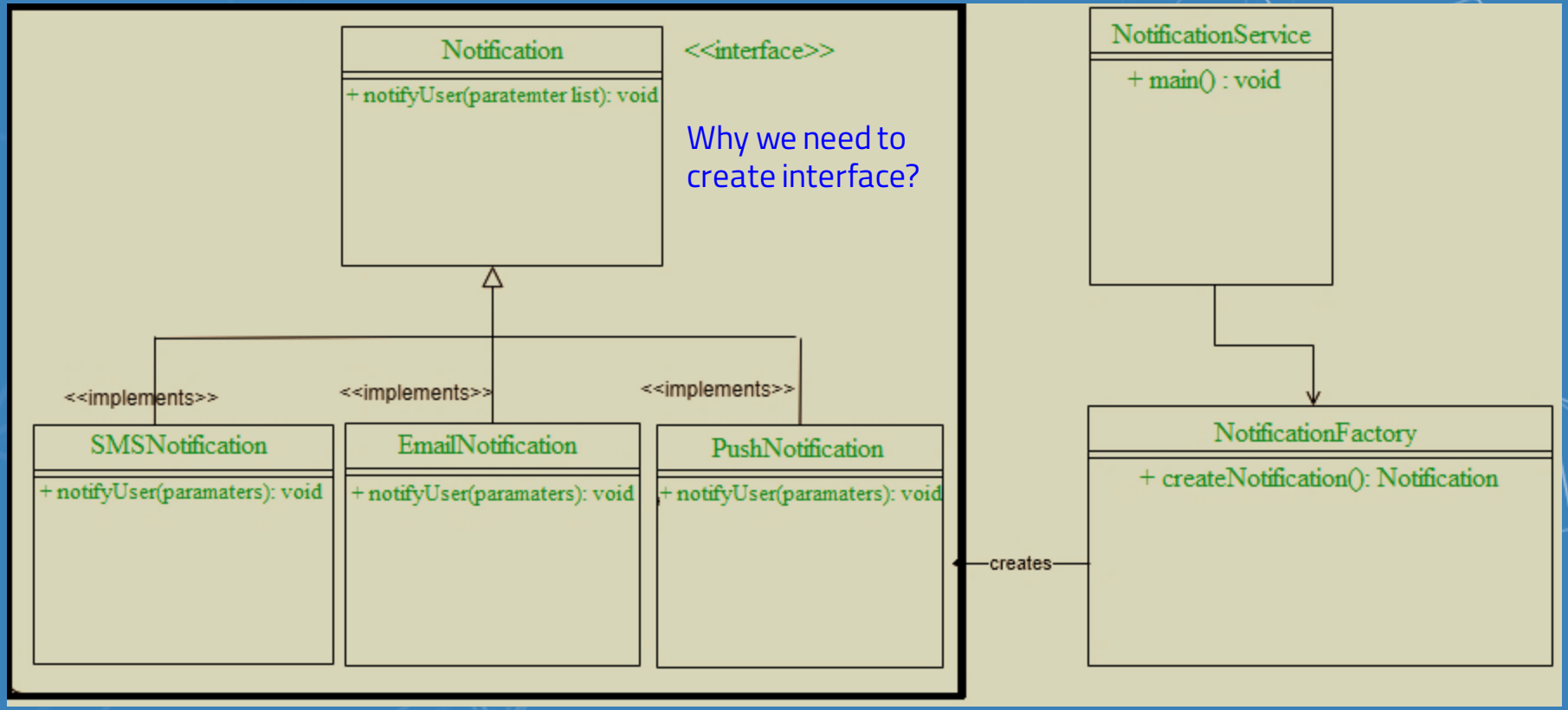
Factory

- Move the object creation logic from client to a separate class.
- Object creation logic is hidden to the client.
- Subclasses decide which object to instantiate by overloading the factory method.
- Also Known As Virtual Constructor.

Factory

Allow new classes to be added to system and handle creation without affection client code.

Factory Example [1]



Factory Example

- Create Notification interface

```
public interface Notification {

    public abstract void notifyUser();

}
```

Factory Example

- Create all implementation classes

```
public class SMSNotification implements Notification {

    @Override
    public void notifyUser() {
        System.out.println("Sending SMS notifiaction");
    }

}
```

```
public class EmailNotification implements Notification {

    @Override
    public void notifyUser() {
        System.out.println("Sending email notifiaction");
    }

}
```

Factory Example

- Create a factory class to instantiate concrete class.

```
public class NotificationFactory {

    public Notification createNotification(String channel) {
        if (channel.equals("SMS"))
            return new SMSNotification();

        if (channel.equals("EMAIL"))
            return new EmailNotification();

        return null;
    }
}
```

Factory Example

- Create and get an object of concrete class

```
public class NotificationService {

    public static void main(String[] args) {
        NotificationFactory notificationFactory = new NotificationFactory();

        Notification notification = notificationFactory.createNotification("SMS");
        notification.notifyUser();

        notification = notificationFactory.createNotification("EMAIL");
        notification.notifyUser();
    }
}
```

Factory Example

Add “Push Notification” class to your implementation.

Prototype

- Complex object is costly to create.
- To create more instances we use an existing instance as our prototype.
- Used when creating a new object is expensive and costs resources:
  - performance cost.
  - Some calculation is needed to create object.

Cloneable Interface in Java

- The cloneable interface in java allows the implementing class to clone its objects without needing to use the new operator.
- It offers a method called clone() defined in the Object class that is used for cloning.
- Syntax of the clone() method:

```
protected Object clone() throws CloneNotSupportedException {
```

Clone → Simple Example

```java
public class Employee implements Cloneable {

    private int id;
    private String name;

    public Employee() {

    }

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public Employee clone() throws CloneNotSupportedException {
        return (Employee)super.clone();
    }
```

```java
public class TestCloneable {

    public static void main(String[] args) throws CloneNotSupportedException {

        Employee e1 = new Employee(1, "Hamzeh");
        Employee e2 = e1.clone();

        System.out.println(e1 + " " + e1.getId() + " " + e1.getName());
        System.out.println(e2 + " " + e2.getId() + " " + e2.getName());
    }

}
```

New Operator VS Cloneable Interface

- New operator is a pretty expensive operation especially when you are creating complex objects, it would take more processing time, first for allocating memory and then for copying every field one by one.

- Clone() creates a new instance of the same class and directly copies all the fields to the new instance. it saves the extra processing that would be used in the first allocation and then in copying the values.

Shallow Copy VS Deep Copy

- In a shallow copy, All the fields of the original objects are copied as it is but, if the original object contains any objects as fields then only their references will be copied, not the complete objects.
- Deep copy stores the copy of the original object and recursively copies the objects as well.
- It is possible to create a deep copy of an object but it required to manually modify the clone() method.

Shallow Copy

```java
public class TestCloneable {

    public static void main(String[] args) throws CloneNotSupportedException {

        Address address = new Address("Jordan", "Amman", "079xxxxxxx");

        Employee e1 = new Employee(1, "Hamzeh", address);
        Employee e2 = e1.clone();

        System.out.println(e1);
        System.out.println(e2);

        e2.getAddresse().setCity("Irbid");

        System.out.println(e1);
        System.out.println(e2);
    }

}
```

```java
@Override
public Employee clone() throws CloneNotSupportedException {
    return (Employee)super.clone();
}
```

roblems | Console | Progress

ninated> TestCloneable [Java Application] D:\jsfCourse\java\jdk1.8.0_202\bin\javaw.exe (Dec 3, 2021 9:49:19 PM – 9:49:20 PM)

```
prototype.Employee@15db9742 Employee [id=1, name=Hamzeh Address [country=Jordan, city=Amman, mobile=079xxxxxxx]]
prototype.Employee@6d06d69c Employee [id=1, name=Hamzeh Address [country=Jordan, city=Amman, mobile=079xxxxxxx]]
prototype.Employee@15db9742 Employee [id=1, name=Hamzeh Address [country=Jordan, city=Irbid, mobile=079xxxxxxx]]
prototype.Employee@6d06d69c Employee [id=1, name=Hamzeh Address [country=Jordan, city=Irbid, mobile=079xxxxxxx]]
```

Deep Copy

```java
public class TestCloneable {

    public static void main(String[] args) throws CloneNotSupportedException {

        Address address = new Address("Jordan", "Amman", "079xxxxxxx");

        Employee e1 = new Employee(1, "Hamzeh", address);
        Employee e2 = e1.clone();

        System.out.println(e1);
        System.out.println(e2);

        e2.getAddresse().setCity("Irbid");

        System.out.println(e1);
        System.out.println(e2);
    }

}
```

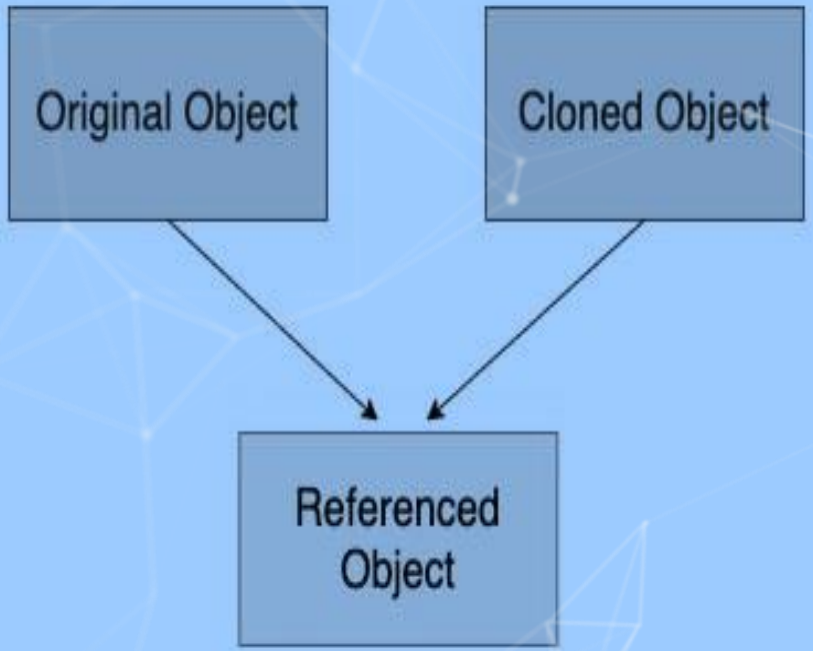
```java
@Override
public Employee clone() throws CloneNotSupportedException {

    Employee emp = (Employee)super.clone();
    emp.addresse = new Address(this.addresse.getCountry(),
                               this.addresse.getCity(),
                               this.addresse.getMobile());
    return emp;

}
```

roblems | Console | Progress

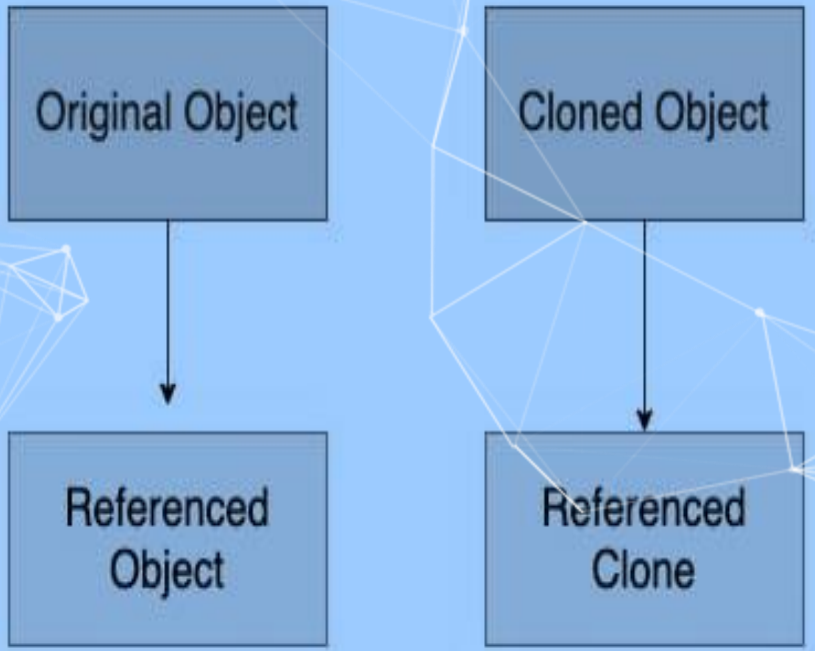
inated> TestCloneable [Java Application] D:\jsfCourse\java\jdk1.8.0_202\bin\javaw.exe (Dec 3, 2021 9:54:47 PM – 9:54:48 PM)

```
prototype.Employee@15db9742 Employee [id=1, name=Hamzeh Address [country=Jordan, city=Amman, mobile=079xxxxxxx]]
prototype.Employee@6d06d69c Employee [id=1, name=Hamzeh Address [country=Jordan, city=Amman, mobile=079xxxxxxx]]
prototype.Employee@15db9742 Employee [id=1, name=Hamzeh Address [country=Jordan, city=Amman, mobile=079xxxxxxx]]
prototype.Employee@6d06d69c Employee [id=1, name=Hamzeh Address [country=Jordan, city=Irbid, mobile=079xxxxxxx]]
```

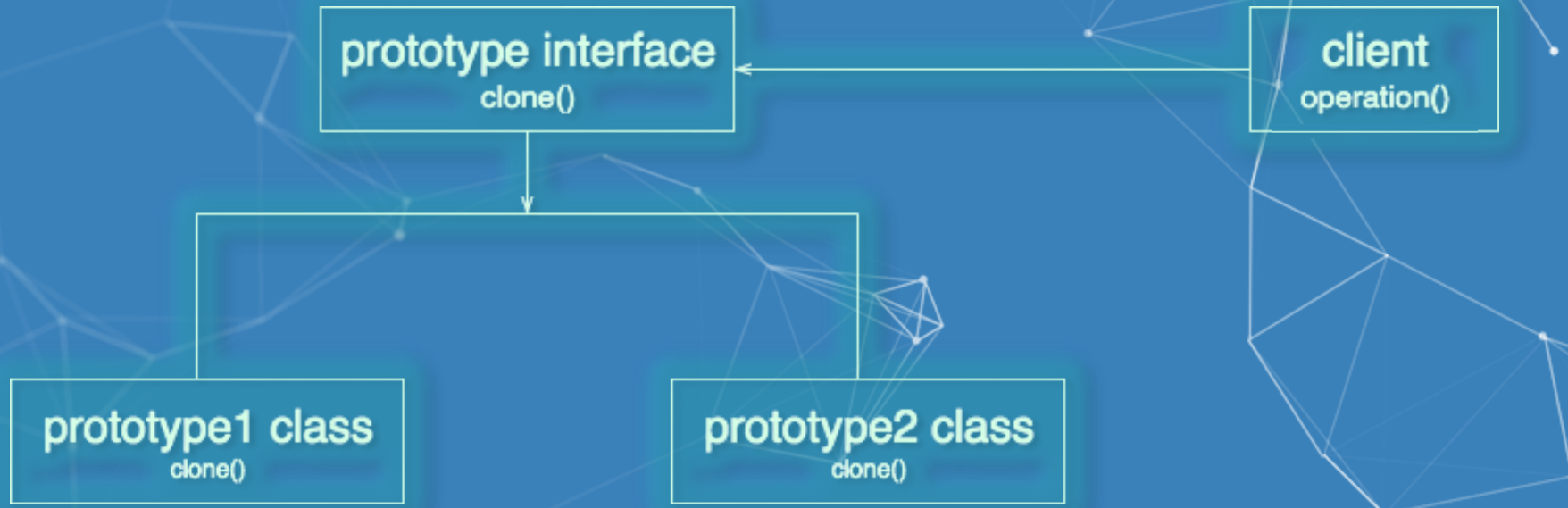
Shallow Clone



Deep Clone



Prototype


prototype interface
clone()
client
operation()
prototype1 class
clone()
prototype2 class
clone()

Why we using CloneNotSupportedException?

Singleton

- The singleton pattern is the most famous among all the design patterns.
- Ensure a class only has one instance, and provide a global point of access to it.
- Make constructor as private.
- Write a static method that has return type object of this singleton class.
- Use getInstance() method to create object instead of constructor.

Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

→ Singleton design pattern is the best solution.

Singleton Example

```
public class Singleton {

    public static Singleton instance = null;

    private Singleton() {

    }

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }

        return instance;
    }

}
```

Singleton Example

```
public class TestSinglton {

    public static void main(String[] args) {

        Singleton x = Singleton.getInstance();
        Singleton y = Singleton.getInstance();
        Singleton z = Singleton.getInstance();

        System.out.println("x = " + x);
        System.out.println("y = " + y);
        System.out.println("z = " + z);

        System.out.println("-------------------------------------------------");
        if (x == y && y == z) {
            System.out.println("Three objects point to the same memory location");
        }
    }
}
```

Singleton Example

- Application configuration values can be tracked in singleton:

  → Theses values are read from file at start and then referred to by other parts of application.

HashMap

- It provides the basic implementation of the Map interface of Java.
- It stores the data in (Key, Value) pairs.
- One object is used as a key (index) to another object (value).
- It can store different types: String keys and Integer values, or the same type, like: String keys and String values.

HashMap Example [2]

```
public class TestHashMap {

    public static void main(String[] args) {

        HashMap<String, String> capitalCities = new HashMap<String, String>();

        // Add keys and values (Country, City)
        capitalCities.put("Jordan", "Amman");
        capitalCities.put("Palestine", "Al Quds");
        capitalCities.put("Syria", "Damascus");
        capitalCities.put("Lebanon ", "Beirut");

        System.out.println(capitalCities);

        System.out.println("Access a value in the HashMap --> " + capitalCities.get("Jordan"));

    }

}
```

Singleton Example 2

```java
public class ConfigManager {

    public static ConfigManager instance;
    private HashMap<String, String> configMap;

    private ConfigManager() {
        configMap = new HashMap<String, String>();
        configMap.put("host", "htu-db");
        configMap.put("port", "1521");
        configMap.put("user", "system");
        configMap.put("password", "system");
    }

    public String update(String key, String value) {
        return configMap.put(key, value);
    }

    public String get(String key) {
        return configMap.get(key);
    }

    public static ConfigManager getInstance() {
        if (instance == null) {
            instance = new ConfigManager();
        }
        return instance;
    }
}
```

```java
public class TestSinglton {

    public static void main(String[] args) {

        ConfigManager configManager = ConfigManager.getInstance();
        ConfigManager configManager2 = ConfigManager.getInstance();

        System.out.println(configManager.get("host"));
        System.out.println(configManager.get("password"));

        configManager.update("password", "Aa.2021");
        System.out.println(configManager.get("password"));
        System.out.println(configManager2.get("password"));

    }

}
```

Builder

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

- It is used to construct a complex object step by step and the final step will return the object.

Builder Example

```
public class User {

    private String firstName;
    private String lastName;
    private int age;
    private String phone;

    public User(UserBuilder userBuilder) {
        this.firstName = userBuilder.getFirstName();
        this.lastName = userBuilder.getLastName();
        this.age = userBuilder.getAge();
        this.phone = userBuilder.getPhone();
    }
    // public User(String firstName, String lastName, int age, String phone) {

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
```

Builder Example

```java
public class UserBuilder {

    private String firstName;
    private String lastName;
    private int age;
    private String phone;

    public UserBuilder withFirstName(String firstName) {
        this.firstName = firstName;
        return this;
    }

    public UserBuilder withLastName(String lastName) {
        this.lastName = lastName;
        return this;
    }

    public UserBuilder withAge(int age) {
        this.age = age;
        return this;
    }

    public UserBuilder withPhone(String phone) {
        this.phone = phone;
        return this;
    }

    public User build() {
        // return new User(firstName, lastName, age, phone);
        return new User(this);
    }

    public String getFirstName() {
        return firstName;
    }
```

```java
public class TestUserBuilder {

    public static void main(String[] args) {

        User user1 = new UserBuilder()
                        .withFirstName("Hamzeh")
                        .withLastName("Asefan")
                        .withAge(39)
                        .withPhone("079xxxxxxx")
                        .build();
        System.out.println(user1);

        // No age
        // No Phone
        User user2 = new UserBuilder()
                        .withFirstName("Hamzeh")
                        .withLastName("Asefan")
                        .build();
        System.out.println(user2);

    }

}
```

Builder → Existing Implementations in JDK

- java.lang.StringBuilder → append()

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

    StringBuilder builder = new StringBuilder("Test String Builder: ");
    String data = builder
                    .append(100)
                    .append(" Hamzeh ")
                    .append("Asefan ")
                    .append(1000.00)
                    .append(" ")
                    .append(true)
                    .toString();

    System.out.println(data);

}
```


THANKS!
Any questions?

References

[1] https://www.geeksforgeeks.org/factory-method-design-pattern-in-java/

[2] https://www.w3schools.com/java/java_hashmap.asp