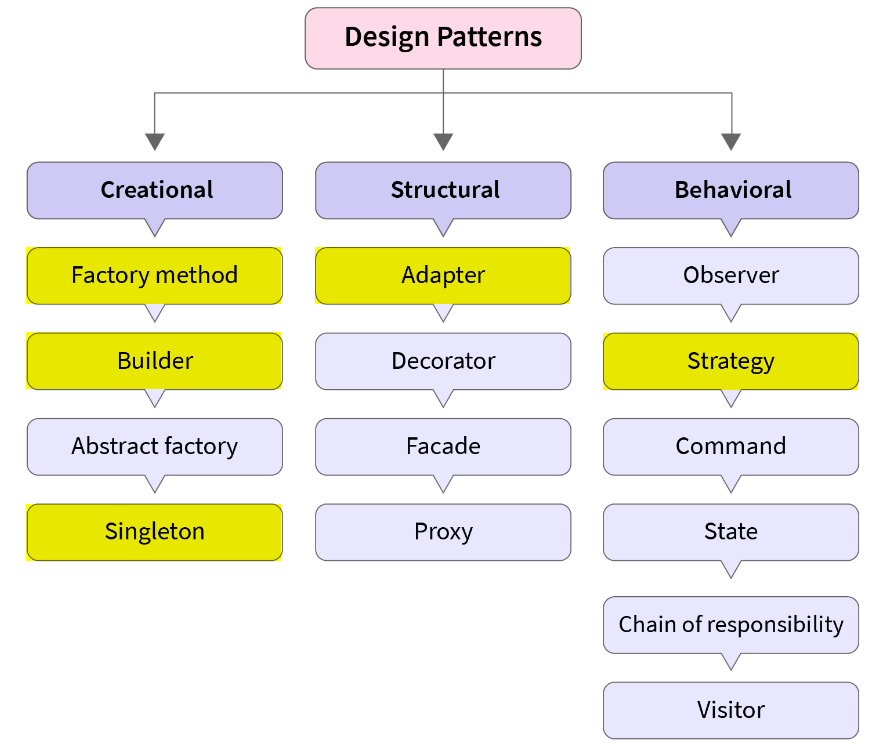
# Design Patterns:

Design patterns are reusable solutions to common software development problems. They provide a template for solving similar problems in different scenarios by providing a set of best practices and guidelines. These patterns help developers to write efficient, maintainable, and scalable code. Design patterns can be classified into three categories: creational patterns, structural patterns, and behavioral patterns. Creational patterns deal with object creation mechanisms, structural patterns deal with object composition, and behavioral patterns deal with object communication and delegation.  
<https://www.scaler.com/topics/design-patterns/types-of-design-pattern/>  
<https://www.scaler.com/topics/design-patterns/>



## Creational Design Pattern:

In software engineering, Creational Design Patterns are a category of design patterns that focus on object creation mechanisms, trying to create objects in a manner suitable to the situation. The main goal of these patterns is to provide more flexibility and reusability to the code.

There are five types of Creational Design Patterns:

1. Singleton Pattern: This pattern ensures that a class has only one instance, and provides a global point of access to it. It is useful in situations where you need to limit the number of instances of a class that can be created.
2. Factory Method Pattern: This pattern provides an interface for creating objects, but allows subclasses to decide which class to instantiate. It is useful when you want to defer the creation of an object to subclasses.
3. Abstract Factory Pattern: This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It is useful when you need to create objects that have dependencies on other objects.
4. Builder Pattern: This pattern separates the construction of a complex object from its representation, allowing you to create different representations of the same object. It is useful when you need to create complex objects that have multiple parts.
5. Prototype Pattern: This pattern creates new objects by cloning existing ones. It is useful when the cost of creating a new object is high, and you want to avoid it by cloning an existing one.

Each of these patterns has its own advantages and disadvantages and can be used in different situations. It is important to choose the right pattern based on your specific needs.

### Singleton Design Pattern:

The Singleton pattern can be useful in a variety of scenarios where we need to ensure that there is only one instance of a particular class throughout the lifetime of an application and that this instance is globally accessible.

#### Example 1:

| public class Singleton {   private static Singleton instance;   private Singleton(){}   public static Singleton getInstance(){  if(instance == null){  instance = new Singleton();  }  return instance;  }   public void showMessage(){  System.out.println("Hello World!");  } } |
| --- |

In this example, we create a class called "Singleton" that has a private constructor, so it cannot be instantiated from outside the class. We also declare a private static variable "instance" of the same class, which is used to store the single instance of the class.

The getInstance() method is used to access the single instance of the class. If the instance has not yet been created, the method creates a new instance and returns it. Otherwise, it simply returns the existing instance.

The showMessage() method is just an example of a method that can be called on the Singleton instance. In a real-world example, this method would perform some useful operations in the context of the system.

To use the Singleton class, you can simply call the getInstance() method to get the single instance of the class, and then call any methods on that instance. Here's an example of how to use the Singleton class:

| public class Main {  public static void main(String[] args) {   Singleton singleton = Singleton.getInstance();   singleton.showMessage();  } }  In this example, we create a new Singleton instance by calling the getInstance() method, and then we call the showMessage() method on that instance. |
| --- |

#### Example 2:

Here's an example of implementing a singleton class with double-checked locking in Java:

| public class Singleton {    private volatile static Singleton instance;    private Singleton() {}    public static Singleton getInstance() {  if (instance == null) {  synchronized (Singleton.class) {  if (instance == null) {  instance = new Singleton();  }  }  }  return instance;  }    public void showMessage() {  System.out.println("Hello World!");  } } |
| --- |

In this implementation, we have added the volatile keyword to the instance variable, which ensures that any changes made to the variable are immediately visible to all threads. We also use double-checked locking to ensure that only one instance of the Singleton class is created.

The getInstance() method first checks if the instance variable is null without acquiring the lock on the Singleton class. If it is null, the method acquires the lock and then checks again to see if the instance variable is still null. If it is, the method creates a new instance of the Singleton class.

By using double-checked locking and making the instance variable volatile, we can ensure that our Singleton class is thread-safe and can be safely used in a multi-threaded environment.

#### Here are some scenarios where using the Singleton pattern can be helpful:

##### Database Connection:

When an application needs to connect to a database, it's often useful to use a Singleton to ensure that there is only one connection to the database throughout the application's lifetime.

| public class DatabaseConnection {  private static volatile DatabaseConnection instance;  private Connection connection;   private DatabaseConnection() {  // Initialize database connection  }   public static DatabaseConnection getInstance() {  if (instance == null) {  synchronized (DatabaseConnection.class) {  if (instance == null) {  instance = new DatabaseConnection();  }  }  }  return instance;  }   public Connection getConnection() {  return connection;  } } |
| --- |

##### Configuration Management:

In many applications, there is a need for a single source of configuration information that can be accessed from various parts of the application. A Singleton can be used to store this configuration data.

| public class Configuration {  private static volatile Configuration instance;  private Properties properties;   private Configuration() {  // Load configuration properties from file or database  }   public static Configuration getInstance() {  if (instance == null) {  synchronized (Configuration.class) {  if (instance == null) {  instance = new Configuration();  }  }  }  return instance;  }   public String getProperty(String key) {  return properties.getProperty(key);  } } |
| --- |

##### Logging:

##### A Singleton can be used to implement a logging system that writes log messages to a file or database. This ensures that all log messages are written to the same location and that there is only one instance of the logging system.

| public class Logger {  private static volatile Logger instance;  private PrintWriter writer;   private Logger() {  // Initialize log file or database connection  }   public static Logger getInstance() {  if (instance == null) {  synchronized (Logger.class) {  if (instance == null) {  instance = new Logger();  }  }  }  return instance;  }   public void log(String message) {  writer.println(message);  } } |
| --- |

##### Caching:

##### When an application needs to cache data, it's often useful to use a Singleton to store the cached data. This ensures that there is only one instance of the cache and that all parts of the application can access the same cached data.

| public class Cache {  private static volatile Cache instance;  private Map < String, Object > cache;   private Cache() {  // Initialize cache  }   public static Cache getInstance() {  if (instance == null) {  synchronized(Cache.class) {  if (instance == null) {  instance = new Cache();  }  }  }  return instance;  }   public void put(String key, Object value) {  cache.put(key, value);  }   public Object get(String key) {  return cache.get(key);  } } |
| --- |

##### Thread Pools:

##### When an application needs to use a pool of threads to execute tasks, it's often useful to use a Singleton to manage the pool of threads. This ensures that there is only one instance of the thread pool and that all tasks are executed on the same pool of threads.

public class ThreadPool {  
 private static volatile ThreadPool instance;  
 private ExecutorService executor;  
  
 private ThreadPool() {  
 // Initialize thread pool  
 }  
  
 public static ThreadPool getInstance() {  
 if (instance == null) {  
 synchronized (ThreadPool.class) {  
 if (instance == null) {  
 instance = new ThreadPool();  
 }  
 }  
 }  
 return instance;  
 }  
  
 public void execute(Runnable task) {  
 executor.execute(task);  
 }  
}

##### Device Drivers:

##### In many applications, it's necessary to interact with hardware devices in order to perform tasks. Here's an example of how the Singleton pattern can be used to manage a device driver:

| public class DeviceDriver {  private static volatile DeviceDriver instance;  private Device device;   private DeviceDriver() {  // Initialize device  }   public static DeviceDriver getInstance() {  if (instance == null) {  synchronized (DeviceDriver.class) {  if (instance == null) {  instance = new DeviceDriver();  }  }  }  return instance;  }   public void write(byte[] data) {  device.write(data);  }   public byte[] read() {  return device.read();  } } |
| --- |

In general, the Singleton pattern is useful whenever there is a need for a single instance of a particular class throughout the lifetime of an application. However, it's important to use the pattern judiciously and not overuse it, as it can lead to tight coupling between different parts of the application and make the code harder to test and maintain.

### 

### Builder Design Pattern:

The Builder pattern is a design pattern that allows you to create complex objects step by step. It provides a way to create different variations of an object while using the same construction code.

In simpler words, imagine that you want to build a house. You can use different builders for different parts of the house, such as a plumber to handle the pipes and a carpenter to build the doors. However, you need someone to coordinate all of the builders to create the final house. The Builder pattern provides this coordination.

In software development, you might want to create an object that has many different attributes. Instead of having one constructor that takes all the attributes as arguments, the Builder pattern allows you to create a separate builder class that sets each attribute separately. This makes the object creation more flexible and easier to understand.

#### Example 1:

| public class Car {  private String make;  private String model;  private int year;  private String color;   private Car(String make, String model, int year, String color) {  this.make = make;  this.model = model;  this.year = year;  this.color = color;  }   @Override  public String toString() {  return "Car{" +  "make='" + make + '\'' +  ", model='" + model + '\'' +  ", year=" + year +  ", color='" + color + '\'' +  '}';  }   public static class CarBuilder {  private String make;  private String model;  private int year;  private String color;   public CarBuilder() {  }   public CarBuilder setMake(String make) {  this.make = make;  return this;  }   public CarBuilder setModel(String model) {  this.model = model;  return this;  }   public CarBuilder setYear(int year) {  this.year = year;  return this;  }   public CarBuilder setColor(String color) {  this.color = color;  return this;  }   public Car build() {  return new Car(make, model, year, color);  }  } } |
| --- |

In this example, the Car class represents the object that we want to create. We have a separate Car.CarBuilder class, which sets each attribute of the Car object separately, and then constructs the final Car object using the Car.CarBuilder.build() method.

Here's an example of how we can use the Builder pattern to create a Car object:

Car car = new Car.CarBuilder()  
 .setMake("Toyota")  
 .setModel("Camry")  
 .setYear(2021)  
 .setColor("Red")  
 .build();  
System.out.println(car);

In this example, we use the Car.CarBuilder class to create a Car object with a make of Toyota, model of Camry, the year of 2021, and color of Red. The Car.CarBuilder.build() method constructs the final Car object, which we then print to the console.

#### Example 2:

Suppose you are developing an e-commerce application that allows users to place orders for products. You want to create an Order object that contains information about the products that were ordered, the billing and shipping addresses, and other optional details such as discounts and taxes.

Using the Builder pattern, you can create a OrderBuilder class that allows you to set each of these parameters separately, and then construct the final Order object using the build() method. Here's an example of how this might look in Java:

| public class Order {  private List<Product> products;  private Address billingAddress;  private Address shippingAddress;  private double discount;  private double tax;   private Order(List<Product> products, Address billingAddress, Address shippingAddress,  double discount, double tax) {  this.products = products;  this.billingAddress = billingAddress;  this.shippingAddress = shippingAddress;  this.discount = discount;  this.tax = tax;  }   // getters and setters   public static class OrderBuilder {  private List<Product> products;  private Address billingAddress;  private Address shippingAddress;  private double discount;  private double tax;   public OrderBuilder() {  products = new ArrayList<>();  discount = 0.0;  tax = 0.0;  }   public OrderBuilder addProduct(Product product) {  products.add(product);  return this;  }   public OrderBuilder setBillingAddress(Address billingAddress) {  this.billingAddress = billingAddress;  return this;  }   public OrderBuilder setShippingAddress(Address shippingAddress) {  this.shippingAddress = shippingAddress;  return this;  }   public OrderBuilder setDiscount(double discount) {  this.discount = discount;  return this;  }   public OrderBuilder setTax(double tax) {  this.tax = tax;  return this;  }   public Order build() {  return new Order(products, billingAddress, shippingAddress, discount, tax);  }  } } |
| --- |

In this example, the Order class represents the object that we want to create. We have a separate Order.OrderBuilder class that sets each attribute of the Order object separately, and then constructs the final Order object using the Order.OrderBuilder.build() method.

Here's an example of how we can use the Builder pattern to create an Order object:

| Order order = new Order.OrderBuilder()  .addProduct(new Product("iPhone", 999.99))  .addProduct(new Product("MacBook", 1999.99))  .setBillingAddress(new Address("John Doe", "123 Main St", "Anytown", "CA", "12345"))  .setShippingAddress(new Address("Jane Doe", "456 Elm St", "Anytown", "CA", "12345"))  .setDiscount(100.0)  .setTax(89.0)  .build();  In this example, we create an `OrderBuilder` object and use the `addProduct()`, `setBillingAddress()`, `setShippingAddress()`, `setDiscount()` and `setTax()` methods to set each attribute of the `Order` object separately. Finally, we call the `build()` method to create the final `Order` object.  Using the Builder pattern in this way makes it easy to create complex objects with many optional parameters. It also makes the code more readable and easier to understand, because each parameter is set separately using a descriptive method name. |
| --- |

#### Commonly used in scenarios:

The Builder design pattern is commonly used in scenarios where the creation of an object involves a large number of optional parameters, or when you want to create objects in a step-by-step fashion. Here are some scenarios where you might want to use the Builder pattern:

##### Creating immutable objects:

In some cases, it's desirable to create objects that can't be modified once they're created. The Builder pattern is well-suited to this, because it allows you to construct an object piece by piece, and then create an immutable version of that object.

| public class Employee {  private final String firstName;  private final String lastName;  private final int age;  private final String department;   public static class Builder {  private final String firstName;  private final String lastName;  private int age;  private String department;   public Builder(String firstName, String lastName) {  this.firstName = firstName;  this.lastName = lastName;  }   public Builder age(int age) {  this.age = age;  return this;  }   public Builder department(String department) {  this.department = department;  return this;  }   public Employee build() {  return new Employee(this);  }  }   private Employee(Builder builder) {  firstName = builder.firstName;  lastName = builder.lastName;  age = builder.age;  department = builder.department;  }   public String getFirstName() {  return firstName;  }   public String getLastName() {  return lastName;  }   public int getAge() {  return age;  }   public String getDepartment() {  return department;  } } |
| --- |

In this example, we define an Employee class with four final fields: firstName, lastName, age, and department. We also define a Builder class with the same four fields. The Builder class has methods for setting the age and department fields, and a build() method that creates a new Employee object.

Here's how you can create a new Employee object using the Builder:

| Employee employee = new Employee.Builder("John", "Doe")  .age(30)  .department("IT")  .build(); |
| --- |

This creates a new Employee object with the firstName "John", lastName "Doe", age 30, and department "IT". Once the Employee object is created, it can't be modified, because all of the fields are final.

##### Creating complex objects:

If an object has many attributes, creating it using a constructor with many parameters can be confusing and error-prone. The Builder pattern can simplify this process by providing a fluent interface that guides the developer through the creation process. Refer to examples 1 & 2.

##### Providing default values:

If an object has many optional attributes, the Builder pattern can be used to provide default values for those attributes. This can simplify the creation process and make it less error-prone.

Suppose we have a Person class with multiple optional attributes like age, height, weight, and address. We can use the Builder pattern to provide default values for these optional attributes:

| public class Person {  private final String name;  private final int age;  private final double height;  private final double weight;  private final String address;   private Person(Builder builder) {  this.name = builder.name;  this.age = builder.age;  this.height = builder.height != null ? builder.height : 0;  this.weight = builder.weight != null ? builder.weight : 0;  this.address = builder.address != null ? builder.address : "";  }   // getters and builder class  // ... }  public static void main(String[] args) {  Person person1 = new Person.Builder("John")  .age(30)  .address("123 Main St")  .build();  Person person2 = new Person.Builder("Mary")  .weight(60)  .build(); } |
| --- |

In the above example, we have provided default values of 0 for height and weight and an empty string for address. When we create an object using the builder, we only need to provide the required attributes, and the builder will use the default values for the optional attributes that are not specified.

The default values could be set in the builder class itself.

| public class Person {  private final String name;  private final int age;  private final int height;  private final int weight;  private final String address;    private Person(Builder builder) {  this.name = builder.name;  this.age = builder.age;  this.height = builder.height;  this.weight = builder.weight;  this.address = builder.address;  }    public static class Builder {  // Required parameters  private final String name;    // Optional parameters - initialized to default values  private int age = 0;  private int height = 0;  private int weight = 0;  private String address = "Not available";    public Builder(String name) {  this.name = name;  }    public Builder age(int age) {  this.age = age;  return this;  }    public Builder height(int height) {  this.height = height;  return this;  }    public Builder weight(int weight) {  this.weight = weight;  return this;  }    public Builder address(String address) {  this.address = address;  return this;  }    public Person build() {  return new Person(this);  }  }    public static void main(String[] args) {  Person person = new Person.Builder("John")  .age(25)  .height(180)  .build();  } } |
| --- |

##### Supporting different object representations:

Sometimes, you need to create objects that can be represented in different ways. For example, you might need to create an object that can be serialized to JSON or XML. The Builder pattern can be used to create a single object that can be represented in different ways, depending on the needs of the application.

Suppose you have a class Person that represents a person with some attributes like name, age, and email. Now you want to support creating Person objects that can be represented in different formats such as JSON and XML.

| public class Person {  private String name;  private int age;  private String email;   public static class Builder {  private String name;  private int age;  private String email;   public Builder name(String name) {  this.name = name;  return this;  }   public Builder age(int age) {  this.age = age;  return this;  }   public Builder email(String email) {  this.email = email;  return this;  }   public Person build() {  return new Person(this);  }   public String toJson() {  // return the JSON representation of the Person object  }   public String toXml() {  // return the XML representation of the Person object  }  }   private Person(Builder builder) {  name = builder.name;  age = builder.age;  email = builder.email;  }   // getters for name, age, and email } |
| --- |

In this example, the Person class has a Builder class nested inside it. The Builder has methods for setting the attributes of the Person object and a build() method for creating a new Person object.

Additionally, the Builder class has two methods for representing the Person object in different formats: toJson() and toXml(). These methods use the data in the Builder instance to create a JSON or XML string representation of the Person object.

Here's an example of how you could use the Builder to create a Person object and represent it as JSON:

| Person person = new Person.Builder()  .name("John Doe")  .age(30)  .email("john.doe@example.com")  .build(); String json = person.toJson(); |
| --- |

And here's an example of how you could use the Builder to create the same Person object and represent it as XML:

Person person = new Person.Builder()  
 .name("John Doe")  
 .age(30)  
 .email("john.doe@example.com")  
 .build();  
String xml = person.toXml();

In this way, the Builder pattern allows you to create objects that can be represented in different ways, depending on the needs of your application.

##### When creating objects with circular dependencies:

Circular dependencies occur when two or more objects depend on each other in a way that creates a loop. The Builder pattern can be used to break these circular dependencies and make it easier to create the objects.

Suppose we have two classes Person and Address, where a person has an address, and an address has a reference to the person it belongs to. This creates a circular dependency between the two classes, making it difficult to instantiate them. We can use the builder pattern to create instances of these classes without the circular dependency.

| // Person.java public class Person {  private String name;  private int age;  private Address address;    public Person(String name, int age, Address address) {  this.name = name;  this.age = age;  this.address = address;  }   public void setAddress(Address address) {  this.address = address;  }   public String getName() {  return name;  }   public int getAge() {  return age;  }   public Address getAddress() {  return address;  } }  // Address.java public class Address {  private String street;  private String city;  private Person person;   public Address(String street, String city) {  this.street = street;  this.city = city;  }   public void setPerson(Person person) {  this.person = person;  }   public String getStreet() {  return street;  }   public String getCity() {  return city;  }   public Person getPerson() {  return person;  } }  // PersonBuilder.java public class PersonBuilder {  private String name;  private int age;  private Address address;   public PersonBuilder setName(String name) {  this.name = name;  return this;  }   public PersonBuilder setAge(int age) {  this.age = age;  return this;  }   public PersonBuilder setAddress(Address address) {  this.address = address;  return this;  }   public Person build() {  Person person = new Person(name, age, address);  address.setPerson(person);  return person;  } }  // AddressBuilder.java public class AddressBuilder {  private String street;  private String city;   public AddressBuilder setStreet(String street) {  this.street = street;  return this;  }   public AddressBuilder setCity(String city) {  this.city = city;  return this;  }   public Address build() {  return new Address(street, city);  } }  // Main.java public class Main {  public static void main(String[] args) {  Address address = new AddressBuilder()  .setStreet("123 Main St")  .setCity("Anytown")  .build();   Person person = new PersonBuilder()  .setName("John Doe")  .setAge(30)  .setAddress(address)  .build();   System.out.println(person.getName() + " is " + person.getAge()   + " years old and lives at " + person.getAddress().getStreet()  + " in " + person.getAddress().getCity() + ".");  } } |
| --- |

In this example, we have created separate builders for Person and Address. The PersonBuilder takes an Address as a parameter, and the AddressBuilder doesn't take any parameters. When a Person object is created using the PersonBuilder, it sets the Address object on the Person, and then sets the Person object on the Address. This way, both the Person and Address objects are created without any circular dependencies.

##### Validate an object of a class before creation:

The builder pattern allows you to create objects that are validated before they are created, ensuring that they are in a valid state from the start.

Suppose you have a Person class with some attributes like name, age, and email. You want to ensure that the Person object is valid, which means that it has a non-empty name, a valid age, and a valid email address. You can use the Builder pattern to validate the Person object before creating it.

| public class Person {  private String name;  private int age;  private String email;   public static class Builder {  private String name;  private int age;  private String email;   public Builder name(String name) {  this.name = name;  return this;  }   public Builder age(int age) {  if (age < 0 || age > 120) {  throw new IllegalArgumentException("Invalid age: " + age);  }  this.age = age;  return this;  }   public Builder email(String email) {  if (!isValidEmail(email)) {  throw new IllegalArgumentException("Invalid email: " + email);  }  this.email = email;  return this;  }   private boolean isValidEmail(String email) {  // implement email validation logic here  return true;  }   public Person build() {  if (name == null || name.isEmpty()) {  throw new IllegalStateException("Name is required");  }  if (age == 0) {  throw new IllegalStateException("Age is required");  }  if (email == null || email.isEmpty()) {  throw new IllegalStateException("Email is required");  }  return new Person(this);  }  }   private Person(Builder builder) {  name = builder.name;  age = builder.age;  email = builder.email;  }   // getters for name, age, and email } |
| --- |

In this example, the Builder class has validation logic in the age() and email() methods. If the age is less than 0 or greater than 120, or if the email is not valid, an IllegalArgumentException is thrown.

Additionally, the Builder has a private isValidEmail() method that implements email validation logic. This method can be reused across different methods that need to validate email addresses.

The build() method of the Builder class checks that the required fields (name, age, and email) are not null or empty, and throws an IllegalStateException if any of them are missing.

Here's an example of how you could use the Builder to create a Person object with validation:

| Person person = new Person.Builder()  .name("John Doe")  .age(30)  .email("john.doe@example.com")  .build(); |
| --- |

##### Creating test data:

If you need to create test data for your application, the Builder pattern can be used to generate objects with specific attribute values for testing purposes.

##### Decoupling the object creation process:

If the creation of an object is complex and involves multiple steps, the Builder pattern can be used to decouple the creation process from the rest of the application. This can make the code more modular and easier to maintain.

Suppose we have a Car class that requires a complex object creation process with multiple steps like building the engine, chassis, and wheels. We can use the Builder pattern to decouple the creation process from the rest of the application:

| public class Car {  private final Engine engine;  private final Chassis chassis;  private final List<Wheel> wheels;   private Car(Builder builder) {  this.engine = builder.engine;  this.chassis = builder.chassis;  this.wheels = builder.wheels;  }   public Engine getEngine() {  return engine;  }   public Chassis getChassis() {  return chassis;  }   public List<Wheel> getWheels() {  return wheels;  }   public static class Builder {  private Engine engine;  private Chassis chassis;  private List<Wheel> wheels;   public Builder withEngine(Engine engine) {  this.engine = engine;  return this;  }   public Builder withChassis(Chassis chassis) {  this.chassis = chassis;  return this;  }   public Builder withWheels(Wheel wheel) {  if (this.wheels == null) {  this.wheels = new ArrayList<>();  }  this.wheels.add(wheel);  return this;  }   public Car build() {  return new Car(this);  }  } }  public class Engine {  private final String type;   private Engine(Builder builder) {  this.type = builder.type;  }   public String getType() {  return type;  }   public static class Builder {  private String type;   public Builder withType(String type) {  this.type = type;  return this;  }   public Engine build() {  return new Engine(this);  }  } }  public class Chassis {  private final String color;   private Chassis(Builder builder) {  this.color = builder.color;  }   public String getColor() {  return color;  }   public static class Builder {  private String color;   public Builder withColor(String color) {  this.color = color;  return this;  }   public Chassis build() {  return new Chassis(this);  }  } }  public class Wheel {  private final int size;   private Wheel(Builder builder) {  this.size = builder.size;  }   public int getSize() {  return size;  }   public static class Builder {  private int size;   public Builder withSize(int size) {  this.size = size;  return this;  }   public Wheel build() {  return new Wheel(this);  }  } }  public static void main(String[] args) {  Car car = new Car.Builder()  .withEngine(new Engine.Builder().withType("V8").build())  .withChassis(new Chassis.Builder().withColor("red").build())  .withWheels(new Wheel.Builder().withSize(20).build())  .withWheels(new Wheel.Builder().withSize(20).build())  .withWheels(new Wheel.Builder().withSize(20).build())  .withWheels(new Wheel.Builder().withSize(20).build())  .build();   System.out.println("Engine type: " + car.getEngine().getType());  System.out.println("Chassis color: " + car.getChassis().getColor());  System.out.println("Number of wheels: " + car.getWheels().size()); } |
| --- |

In this example, we have a Car class with a builder and three component classes (Engine, Chassis, and Wheel) with their respective builders. The Car builder allows us to build a car by specifying its components using their respective builders.

### Factory Design Pattern:

The Factory Design Pattern is a creational pattern that provides an interface to create objects in a superclass but allows subclasses to alter the type of objects that will be created. The pattern separates the responsibility of creating an object from the client that requests the object.

There are three subtypes of the Factory Design Pattern:

1. Simple Factory: This subtype of the Factory Design Pattern is the most basic one. In this pattern, a factory class is responsible for creating objects based on the input provided. The client provides the necessary input to the factory, which in turn creates the required object.
2. Factory Method: In the Factory Method Design Pattern, the creation of objects is delegated to the subclasses. This means that the factory method in the superclass is abstract, and each subclass provides its own implementation of the factory method to create the object.
3. Abstract Factory: The Abstract Factory Design Pattern is used to create families of related objects. In this pattern, an abstract factory is defined that provides an interface to create a set of related objects. Each subclass of the abstract factory provides its own implementation of the factory methods to create the objects.
4. Practical Factory Pattern: It is same as simple factory pattern just a different name.

In summary, the Factory Design Pattern provides a way to create objects without exposing the creation logic to the client. It promotes loose coupling by decoupling the client from the concrete classes, making the code more flexible and easier to maintain.

#### Simple Factory Design Pattern:

let's take the example of a pizza ordering system to explain the Simple Factory design pattern.

In a pizza ordering system, we have different types of pizzas such as cheese pizza, pepperoni pizza, and veggie pizza. The Simple Factory pattern can be used to create instances of these pizzas.

##### Example 1:

| // Product: Pizza interface public interface Pizza {  public void prepare();  public void bake();  public void cut();  public void box(); }  // Concrete Product: CheesePizza class public class CheesePizza implements Pizza {  @Override  public void prepare() {  System.out.println("Preparing cheese pizza");  }   @Override  public void bake() {  System.out.println("Baking cheese pizza");  }   @Override  public void cut() {  System.out.println("Cutting cheese pizza");  }   @Override  public void box() {  System.out.println("Boxing cheese pizza");  } }  // Concrete Product: PepperoniPizza class public class PepperoniPizza implements Pizza {  @Override  public void prepare() {  System.out.println("Preparing pepperoni pizza");  }   @Override  public void bake() {  System.out.println("Baking pepperoni pizza");  }   @Override  public void cut() {  System.out.println("Cutting pepperoni pizza");  }   @Override  public void box() {  System.out.println("Boxing pepperoni pizza");  } }  // Concrete Product: VeggiePizza class public class VeggiePizza implements Pizza {  @Override  public void prepare() {  System.out.println("Preparing veggie pizza");  }   @Override  public void bake() {  System.out.println("Baking veggie pizza");  }   @Override  public void cut() {  System.out.println("Cutting veggie pizza");  }   @Override  public void box() {  System.out.println("Boxing veggie pizza");  } }  // Simple Factory: PizzaFactory class public class PizzaFactory {  public Pizza createPizza(String type) {  Pizza pizza = null;  if (type.equals("cheese")) {  pizza = new CheesePizza();  } else if (type.equals("pepperoni")) {  pizza = new PepperoniPizza();  } else if (type.equals("veggie")) {  pizza = new VeggiePizza();  }  return pizza;  } }  // Client: PizzaStore class public class PizzaStore {  PizzaFactory pizzaFactory;   public PizzaStore(PizzaFactory pizzaFactory) {  this.pizzaFactory = pizzaFactory;  }   public Pizza orderPizza(String type) {  Pizza pizza = pizzaFactory.createPizza(type);  pizza.prepare();  pizza.bake();  pizza.cut();  pizza.box();  return pizza;  }   public static void main(String[] args) {  PizzaFactory pizzaFactory = new PizzaFactory();  PizzaStore pizzaStore = new PizzaStore(pizzaFactory);   pizzaStore.orderPizza("cheese");  pizzaStore.orderPizza("pepperoni");  pizzaStore.orderPizza("veggie");  } } |
| --- |

In this code implementation, we have a Pizza interface that defines the methods that every pizza should have. We also have concrete implementations of Pizza interface such as CheesePizza, PepperoniPizza, and VeggiePizza.

The PizzaFactory class provides a simple factory method createPizza(String type) that takes a type parameter and returns an instance of Pizza based on the input provided by the client.

One of the disadvantages of the Simple Factory pattern is that it violates the Open-Closed Principle. If we need to add a new type of product, we need to modify the factory code, which means we need to recompile and redeploy the code.

##### Example 2:

Suppose we have an interface called Vehicle:

| public interface Vehicle {  void drive(); } |
| --- |

We have three classes that implement the Vehicle interface: Car, Truck, and Motorcycle.

| public class Car implements Vehicle {  @Override  public void drive() {  System.out.println("Driving a car.");  } }  public class Truck implements Vehicle {  @Override  public void drive() {  System.out.println("Driving a truck.");  } }  public class Motorcycle implements Vehicle {  @Override  public void drive() {  System.out.println("Driving a motorcycle.");  } } |
| --- |

Finally, we have a VehicleFactory class that creates the different types of vehicles:

| public class VehicleFactory {  public static Vehicle createVehicle(String type) {  if (type.equalsIgnoreCase("car")) {  return new Car();  } else if (type.equalsIgnoreCase("truck")) {  return new Truck();  } else if (type.equalsIgnoreCase("motorcycle")) {  return new Motorcycle();  } else {  throw new IllegalArgumentException("Invalid vehicle type: " + type);  }  } } |
| --- |

We can use the VehicleFactory to create different types of vehicles:

| Vehicle car = VehicleFactory.createVehicle("car"); Vehicle truck = VehicleFactory.createVehicle("truck"); Vehicle motorcycle = VehicleFactory.createVehicle("motorcycle");  car.drive(); truck.drive(); motorcycle.drive(); Example 3: Suppose you are working on an e-commerce website and you need to generate invoices for different types of orders. You have three types of orders - "PhysicalOrder", "DigitalOrder", and "ServiceOrder". Each of these types of orders has its own invoice format.  To implement this, you can use the Factory pattern. You can define an abstract class called "Order" which will have a method to generate the invoice. Then, you can create three concrete classes, "PhysicalOrder", "DigitalOrder", and "ServiceOrder", which will implement the "Order" interface and provide their own implementation of the generateInvoice() method.  Next, you can create a "OrderFactory" class that will create objects of the "Order" interface based on the type of order. The "OrderFactory" class will have a method called "createOrder" which will take the type of order as an input and return an object of the corresponding type.  // Abstract Order class public abstract class Order {  public abstract void generateInvoice(); }  // Concrete PhysicalOrder class public class PhysicalOrder extends Order {  public void generateInvoice() {  // Implementation for generating invoice for physical order  } }  // Concrete DigitalOrder class public class DigitalOrder extends Order {  public void generateInvoice() {  // Implementation for generating invoice for digital order  } }  // Concrete ServiceOrder class public class ServiceOrder extends Order {  public void generateInvoice() {  // Implementation for generating invoice for service order  } }  // OrderFactory class public class OrderFactory {  public static Order createOrder(String type) {  switch(type) {  case "Physical":  return new PhysicalOrder();  case "Digital":  return new DigitalOrder();  case "Service":  return new ServiceOrder();  default:  throw new IllegalArgumentException("Invalid order type: " + type);  }  } }  // Client code that uses the factory to create orders and generate invoices public class Client {  public static void main(String[] args) {  Order physicalOrder = OrderFactory.createOrder("Physical");  physicalOrder.generateInvoice();    Order digitalOrder = OrderFactory.createOrder("Digital");  digitalOrder.generateInvoice();    Order serviceOrder = OrderFactory.createOrder("Service");  serviceOrder.generateInvoice();  } } |
| --- |

#### Factory Method Design Pattern:

Suppose we have a game where players can choose different types of characters to play with, such as a warrior, a mage, or a rogue. Each character has its own set of attributes and abilities. We want to create these characters dynamically at runtime using a factory method pattern.

##### Example 1:

First, we define an abstract Character class that has some common methods for all types of characters:

| public abstract class Character {  protected String name;  protected int health;  protected int strength;  protected int agility;  protected int intelligence;   public Character(String name, int health, int strength, int agility, int intelligence) {  this.name = name;  this.health = health;  this.strength = strength;  this.agility = agility;  this.intelligence = intelligence;  }   public abstract void display();  public abstract void attack(); } |
| --- |

We also define some concrete Character subclasses that represent the different types of characters:

| public class Warrior extends Character {  public Warrior(String name, int health, int strength, int agility, int intelligence) {  super(name, health, strength, agility, intelligence);  }   @Override  public void display() {  System.out.println("I am a warrior named " + name + ".");  }   @Override  public void attack() {  System.out.println("I swing my sword!");  } }  public class Mage extends Character {  public Mage(String name, int health, int strength, int agility, int intelligence) {  super(name, health, strength, agility, intelligence);  }   @Override  public void display() {  System.out.println("I am a mage named " + name + ".");  }   @Override  public void attack() {  System.out.println("I cast a spell!");  } }  public class Rogue extends Character {  public Rogue(String name, int health, int strength, int agility, int intelligence) {  super(name, health, strength, agility, intelligence);  }   @Override  public void display() {  System.out.println("I am a rogue named " + name + ".");  }   @Override  public void attack() {  System.out.println("I strike from the shadows!");  } } |
| --- |

Next, we define a CharacterFactory interface that has a createCharacter method that returns a Character object. We can then create concrete factories that implement this interface to create specific types of characters:

| public interface CharacterFactory {  public Character createCharacter(String name); }  public class WarriorFactory implements CharacterFactory {  public Character createCharacter(String name) {  return new Warrior(name, 100, 10, 5, 5);  } }  public class MageFactory implements CharacterFactory {  public Character createCharacter(String name) {  return new Mage(name, 50, 5, 5, 10);  } }  public class RogueFactory implements CharacterFactory {  public Character createCharacter(String name) {  return new Rogue(name, 75, 7, 10, 3);  } } |
| --- |

Finally, we can use the factory method pattern to dynamically create characters in our game:

| public class Game {  public static void main(String[] args) {  CharacterFactory warriorFactory = new WarriorFactory();  CharacterFactory mageFactory = new MageFactory();  CharacterFactory rogueFactory = new RogueFactory();   Character warrior = warriorFactory.createCharacter("Thor");  Character mage = mageFactory.createCharacter("Gandalf");  Character rogue = rogueFactory.createCharacter("Legolas");   warrior.display();  warrior.attack();   mage.display();  mage.attack();   rogue.display();  rogue.attack();  } } |
| --- |

#### Difference between Simple Factory & Factory Method:

Simple Factory pattern encapsulates the object creation logic in a single factory class, which takes in a parameter and creates the desired object based on that parameter. In other words, the client only interacts with the factory class to get the object it needs.

On the other hand, Factory Method pattern delegates the object creation to the subclasses by defining an abstract method in the creator class. Each subclass implements this method to create the specific object it is responsible for. The client interacts with the creator class and each subclass is responsible for creating its own objects.

So, in short, the key difference is in how the objects are created. In Simple Factory, the factory class creates the objects, while in Factory Method, the subclasses are responsible for creating the objects.

##### Here is a simplified example to illustrate the difference between the two:

Suppose we have a Vehicle interface and two concrete classes, Car and Bike, that implement the interface.

In Simple Factory, we would have a VehicleFactory class that takes a parameter and returns the appropriate object:

| public class VehicleFactory {  public static Vehicle createVehicle(String type) {  if (type.equals("car")) {  return new Car();  } else if (type.equals("bike")) {  return new Bike();  } else {  throw new IllegalArgumentException("Invalid vehicle type: " + type);  }  } } |
| --- |

The client would use this factory class to create the object it needs, like this:

| Vehicle car = VehicleFactory.createVehicle("car"); |
| --- |

In Factory Method, we would define an abstract VehicleFactory class with an abstract method createVehicle():

| public abstract class VehicleFactory {  public abstract Vehicle createVehicle(); } |
| --- |

Then, we would have two concrete subclasses, CarFactory and BikeFactory, that extend the VehicleFactory class and implement the createVehicle() method:

| public class CarFactory extends VehicleFactory {  public Vehicle createVehicle() {  return new Car();  } }  public class BikeFactory extends VehicleFactory {  public Vehicle createVehicle() {  return new Bike();  } } |
| --- |

The client would interact with the creator class, like this:

| VehicleFactory factory = new CarFactory(); Vehicle car = factory.createVehicle(); |
| --- |

#### Abstract Factory Design Patterns:

The Abstract Factory pattern is used when we want to create families of related objects without specifying their concrete classes. It provides an interface for creating families of related objects, without specifying their concrete classes. The key idea behind the Abstract Factory pattern is to provide an abstract factory interface (or class) that defines a set of factory methods, one for each product type that can be produced. Each concrete factory class that implements the abstract factory interface is responsible for creating a family of related products.

##### Example 1:

Let's take an example of a car manufacturing company. The company produces two types of cars - luxury cars and sports cars. Each car type has its own set of related objects such as engine, wheels, and seats.

To implement the Abstract Factory pattern, we can create an abstract factory interface called CarFactory that defines a set of factory methods for creating these related objects. The CarFactory interface will have two concrete implementations - LuxuryCarFactory and SportsCarFactory - which will create the related objects for luxury and sports cars respectively.

Here's the code example:

| // Abstract Factory interface public interface CarFactory {  public Engine createEngine();  public Wheels createWheels();  public Seats createSeats(); }  // Concrete factory for luxury cars public class LuxuryCarFactory implements CarFactory {  public Engine createEngine() {  return new LuxuryEngine();  }  public Wheels createWheels() {  return new LuxuryWheels();  }  public Seats createSeats() {  return new LuxurySeats();  } }  // Concrete factory for sports cars public class SportsCarFactory implements CarFactory {  public Engine createEngine() {  return new SportsEngine();  }  public Wheels createWheels() {  return new SportsWheels();  }  public Seats createSeats() {  return new SportsSeats();  } }  // Abstract Product interface for Engine public interface Engine {  public void design(); }  // Concrete Product for Luxury Engine public class LuxuryEngine implements Engine {  public void design() {  System.out.println("Designing Luxury Engine");  } }  // Concrete Product for Sports Engine public class SportsEngine implements Engine {  public void design() {  System.out.println("Designing Sports Engine");  } }  // Abstract Product interface for Wheels public interface Wheels {  public void design(); }  // Concrete Product for Luxury Wheels public class LuxuryWheels implements Wheels {  public void design() {  System.out.println("Designing Luxury Wheels");  } }  // Concrete Product for Sports Wheels public class SportsWheels implements Wheels {  public void design() {  System.out.println("Designing Sports Wheels");  } }  // Abstract Product interface for Seats public interface Seats {  public void design(); }  // Concrete Product for Luxury Seats public class LuxurySeats implements Seats {  public void design() {  System.out.println("Designing Luxury Seats");  } }  // Concrete Product for Sports Seats public class SportsSeats implements Seats {  public void design() {  System.out.println("Designing Sports Seats");  } }  // Client code that uses the abstract factory to create related objects public class Car {  private Engine engine;  private Wheels wheels;  private Seats seats;    public Car(CarFactory factory) {  engine = factory.createEngine();  wheels = factory.createWheels();  seats = factory.createSeats();  }    public void design() {  engine.design();  wheels.design();  seats.design();  } }  // Example usage public class Main {  public static void main(String[] args) {  CarFactory factory = new LuxuryCarFactory();  Car luxuryCar = new Car(factory);  luxuryCar.design();    factory = new SportsCarFactory();  Car sportsCar = new Car(factory);  sportsCar.design();  } } |
| --- |

In this example, we have an abstract factory interface called CarFactory with three factory methods to create related objects - Engine, Wheels, and Seats. We also have two concrete implementations of the CarFactory interface - LuxuryCarFactory and SportsCarFactory, which create related objects for luxury and sports cars respectively.

We also have abstract product interfaces for Engine, Wheels, and Seats, with concrete product implementations for luxury and sports cars.

The client code uses the abstract factory to create a car by passing in a factory instance, which creates the related objects for the specific car type. The Car class then uses these objects to perform operations such as designing the car.

In the main method, we create a luxury car using the LuxuryCarFactory and a sports car using the SportsCarFactory, demonstrating how the abstract factory pattern can be used to create related objects for different car types.

##### Example 2:

Let's say we have a GUI (Graphical User Interface) library that can render GUI components such as buttons, text boxes, and labels. We want to support multiple platforms such as Windows, macOS, and Linux, and we want to ensure that the GUI components are consistent across platforms.

To implement the Abstract Factory pattern, we can create an abstract factory interface called GUIFactory that defines a set of factory methods for creating GUI components. The GUIFactory interface will have three concrete implementations - WindowsGUIFactory, MacOSGUIFactory, and LinuxGUIFactory - which will create the GUI components for Windows, macOS, and Linux respectively.

| // Abstract Factory interface for creating GUI components public interface GUIFactory {  public Button createButton();  public TextBox createTextBox();  public Label createLabel(); }  // Concrete factory for creating GUI components for Windows platform public class WindowsGUIFactory implements GUIFactory {  public Button createButton() {  return new WindowsButton();  }  public TextBox createTextBox() {  return new WindowsTextBox();  }  public Label createLabel() {  return new WindowsLabel();  } }  // Concrete factory for creating GUI components for macOS platform public class MacOSGUIFactory implements GUIFactory {  public Button createButton() {  return new MacOSButton();  }  public TextBox createTextBox() {  return new MacOSTextBox();  }  public Label createLabel() {  return new MacOSLabel();  } }  // Concrete factory for creating GUI components for Linux platform public class LinuxGUIFactory implements GUIFactory {  public Button createButton() {  return new LinuxButton();  }  public TextBox createTextBox() {  return new LinuxTextBox();  }  public Label createLabel() {  return new LinuxLabel();  } }  // Abstract Product interface for Button public interface Button {  public void paint(); }  // Concrete Product for Windows Button public class WindowsButton implements Button {  public void paint() {  System.out.println("Painting a Windows button.");  } }  // Concrete Product for macOS Button public class MacOSButton implements Button {  public void paint() {  System.out.println("Painting a macOS button.");  } }  // Concrete Product for Linux Button public class LinuxButton implements Button {  public void paint() {  System.out.println("Painting a Linux button.");  } }  // Abstract Product interface for TextBox public interface TextBox {  public void paint(); }  // Concrete Product for Windows TextBox public class WindowsTextBox implements TextBox {  public void paint() {  System.out.println("Painting a Windows text box.");  } }  // Concrete Product for macOS TextBox public class MacOSTextBox implements TextBox {  public void paint() {  System.out.println("Painting a macOS text box.");  } }  // Concrete Product for Linux TextBox public class LinuxTextBox implements TextBox {  public void paint() {  System.out.println("Painting a Linux text box.");  } }  // Abstract Product interface for Label public interface Label {  public void paint(); }  // Concrete Product for Windows Label public class WindowsLabel implements Label {  public void paint() {  System.out.println("Painting a Windows label.");  } }  // Concrete Product for macOS Label public class MacOSLabel implements Label {  public void paint() {  System.out.println("Painting a macOS label.");  } }  // Concrete Product for Linux Label public class LinuxLabel implements Label {  public void paint() {  System.out.println("Painting a Linux label.");  } }  // Client code that uses the abstract factory to create related objects public class Application {  private Button button;  private TextBox textBox;  private Label label;  public Application(GUIFactory factory) {  button = factory.createButton();  textBox = factory.createTextBox();  label = factory.createLabel();  }   public void paint() {  button.paint();  textBox.paint();  label.paint();  } }  // Sample client code that creates an Application for Windows public class WindowsClient {  public static void main(String[] args) {  GUIFactory factory = new WindowsGUIFactory();  Application app = new Application(factory);  app.paint();  } }  // Sample client code that creates an Application for macOS public class MacOSClient {  public static void main(String[] args) {  GUIFactory factory = new MacOSGUIFactory();  Application app = new Application(factory);  app.paint();  } }  // Sample client code that creates an Application for Linux public class LinuxClient {  public static void main(String[] args) {  GUIFactory factory = new LinuxGUIFactory();  Application app = new Application(factory);  app.paint();  } } |
| --- |

In this code, we define an abstract factory interface GUIFactory that declares methods for creating different types of GUI components such as buttons, text boxes, and labels. Then we define three concrete factories WindowsGUIFactory, MacOSGUIFactory, and LinuxGUIFactory that implement the GUIFactory interface to create GUI components specific to the Windows, macOS, and Linux platforms respectively.

We also define three concrete product classes for each of the GUI components: WindowsButton, WindowsTextBox, WindowsLabel for Windows; MacOSButton, MacOSTextBox, MacOSLabel for macOS; and LinuxButton, LinuxTextBox, LinuxLabel for Linux. Each of these classes implements the corresponding abstract product interface.

Finally, we define an Application class that takes a concrete factory as a parameter and uses it to create GUI components for the application. We also define three client classes WindowsClient, MacOSClient, and LinuxClient that create an Application object with the appropriate concrete factory based on the platform they are running on. When the paint() method is called on the Application object, it calls the paint() method on each of the GUI components, which in turn prints out a message indicating which type of component is being painted.

### Prototype Design Pattern:

Prototype design pattern is a creational design pattern that allows creating new objects by cloning existing ones. Instead of creating new objects by invoking their constructors, the prototype pattern involves cloning an existing object and modifying it as required. This pattern is useful when creating new objects is time-consuming or complex, or when the constructor cannot be called directly.

The prototype pattern involves creating a prototype interface or class that declares the clone method, which returns a copy of the current object. The concrete prototype classes implement this interface or extend the prototype class and override the clone method to create a copy of the current object.

#### Example 1:

An example of the prototype pattern is creating an online quiz platform where there are different types of quizzes with different question sets, and each quiz type can have multiple instances with different sets of questions. In this scenario, instead of creating each quiz instance from scratch, we can create a prototype object for each quiz type with the default set of questions. When a new instance of a quiz is required, we can clone the prototype object and modify it with the required set of questions.

| public interface Quiz extends Cloneable {  public Quiz clone() throws CloneNotSupportedException;  public void setQuestions(String[] questions);  public void startQuiz(); }  public class ScienceQuiz implements Quiz {  private String[] questions;   public ScienceQuiz() {  // default set of questions for Science quiz  this.questions = new String[] {"What is the largest planet in our solar system?",  "What is the smallest planet in our solar system?",  "What is the most abundant gas in Earth's atmosphere?"};  }   public Quiz clone() throws CloneNotSupportedException {  ScienceQuiz cloned = new ScienceQuiz();  cloned.setQuestions(this.questions.clone());  return cloned;  }   public void setQuestions(String[] questions) {  this.questions = questions;  }   public void startQuiz() {  System.out.println("Starting Science Quiz with questions:");  for (String question : this.questions) {  System.out.println(question);  }  } }  public class QuizApp {  public static void main(String[] args) throws CloneNotSupportedException {  // create a prototype Science quiz object  Quiz scienceQuizPrototype = new ScienceQuiz();   // create a new Science quiz instance by cloning the prototype object  Quiz scienceQuizInstance1 = scienceQuizPrototype.clone();  scienceQuizInstance1.setQuestions(new String[] {"What is the speed of light?",  "What is the smallest particle in an element?",  "What is the equation for gravitational force?"});  scienceQuizInstance1.startQuiz();   // create another Science quiz instance by cloning the prototype object  Quiz scienceQuizInstance2 = scienceQuizPrototype.clone();  scienceQuizInstance2.setQuestions(new String[] {"What is the unit of electric charge?",  "What is the boiling point of water?",  "What is the process of converting solid directly to gas?"});  scienceQuizInstance2.startQuiz();  } } |
| --- |

In this example, the Quiz interface defines the clone, setQuestions, and startQuiz methods that are implemented by the ScienceQuiz class. The ScienceQuiz class implements the clone method to create a new instance of the ScienceQuiz object and set the questions based on the current object's questions. The startQuiz method displays the questions on the console.

The QuizApp class creates a prototype object of the ScienceQuiz class and then creates two new instances of the Science quiz by cloning the prototype object and setting the questions. The output of the program shows the questions for each quiz instance.

#### Example 2:

Suppose there is a document creation application that allows users to create different types of documents such as letters, reports, and memos. Instead of creating a new document from scratch each time, the application could use the Prototype pattern to clone an existing document object and modify it as needed.

| public abstract class Document implements Cloneable {  private String text;   public String getText() {  return text;  }   public void setText(String text) {  this.text = text;  }   @Override  public Object clone() throws CloneNotSupportedException {  return super.clone();  } }  public class Letter extends Document {  private String recipient;   public String getRecipient() {  return recipient;  }   public void setRecipient(String recipient) {  this.recipient = recipient;  } }  public class Report extends Document {  private String author;   public String getAuthor() {  return author;  }   public void setAuthor(String author) {  this.author = author;  } }  public class DocumentManager {  private Map<String, Document> prototypes = new HashMap<>();   public DocumentManager() {  Letter letter = new Letter();  letter.setText("Dear [recipient],\n\nSincerely,\n[author]");  prototypes.put("letter", letter);   Report report = new Report();  report.setText("This is a report written by [author]");  prototypes.put("report", report);  }   public void addPrototype(String key, Document prototype) {  prototypes.put(key, prototype);  }   public void removePrototype(String key) {  prototypes.remove(key);  }   public Document clone(String key) throws CloneNotSupportedException {  return (Document) prototypes.get(key).clone();  } }  public class Main {  public static void main(String[] args) throws CloneNotSupportedException {  DocumentManager documentManager = new DocumentManager();  documentManager.addPrototype("memo", new Document());   Document letter = documentManager.clone("letter");  ((Letter) letter).setRecipient("John Doe");  ((Letter) letter).setAuthor("Jane Smith");   Document report = documentManager.clone("report");  ((Report) report).setAuthor("Bob Johnson");   System.out.println(letter.getText());  System.out.println(report.getText());  } } |
| --- |

In this example, the Document class is the abstract prototype that defines the interface for creating new documents. The Letter and Report classes are concrete prototypes that extend the Document class and provide their own implementation of the getText() method. The DocumentManager class is responsible for managing the prototypes and creating new documents based on existing prototypes. The Main class demonstrates how the application can create new Letter and Report objects by cloning the corresponding prototypes and modifying the recipient and author fields as needed.

### Registry Design Pattern:

The Registry design pattern is a creational pattern that provides a centralized point of access for objects of a certain type, allowing them to be shared and reused across the system. The registry maintains a registry of objects and provides a way to retrieve and store them.

#### Example 1:

| public class Employee {  private String name;  private int age;  private String department;   public Employee(String name, int age, String department) {  this.name = name;  this.age = age;  this.department = department;  }   public String getName() {  return name;  }   public int getAge() {  return age;  }   public String getDepartment() {  return department;  }   @Override  public String toString() {  return "Employee [name=" + name + ", age=" + age + ", department=" + department + "]";  } }  public class EmployeeRegistry {  private static Map<String, Employee> registry = new HashMap<>();   public static void addEmployee(String id, Employee employee) {  registry.put(id, employee);  }   public static Employee getEmployee(String id) {  return registry.get(id);  }   public static void removeEmployee(String id) {  registry.remove(id);  } }  public class Main {  public static void main(String[] args) {  Employee employee1 = new Employee("John Doe", 30, "Sales");  Employee employee2 = new Employee("Jane Smith", 25, "Marketing");   EmployeeRegistry.addEmployee("001", employee1);  EmployeeRegistry.addEmployee("002", employee2);   Employee retrievedEmployee1 = EmployeeRegistry.getEmployee("001");  Employee retrievedEmployee2 = EmployeeRegistry.getEmployee("002");   System.out.println(retrievedEmployee1);  System.out.println(retrievedEmployee2);   EmployeeRegistry.removeEmployee("002");   Employee retrievedEmployee3 = EmployeeRegistry.getEmployee("002");   System.out.println(retrievedEmployee3);  } } |
| --- |

In this example, we have an Employee class that represents an employee with a name, age, and department. We also have an EmployeeRegistry class that acts as a registry of employees. It maintains a Map of employee IDs to Employee objects, and provides methods to add, retrieve, and remove employees from the registry.

In the Main class, we create two Employee objects and add them to the registry using their IDs. We then retrieve the employees from the registry using their IDs and print out their details. We also remove one of the employees from the registry and attempt to retrieve it again to demonstrate that it has been removed.

This example demonstrates how the Registry pattern can be used to maintain a central registry of objects that can be shared and reused across the system.

#### Example 2:

Suppose you are developing a software system for a hospital. The system needs to keep track of all the patients who are currently admitted to the hospital. Each patient has a unique patient ID and associated information such as name, age, gender, and medical history.

You could use the Registry pattern to manage a central registry of all patient objects in the system. This registry would allow you to quickly look up patient objects by their ID and perform operations on them, such as updating their medical records or discharging them from the hospital.

| public class Patient {  private int id;  private String name;  private int age;  private String gender;  private String medicalHistory;   // constructors, getters, and setters omitted for brevity }  public class PatientRegistry {  private static final Map<Integer, Patient> registry = new HashMap<>();   public static void addPatient(Patient patient) {  registry.put(patient.getId(), patient);  }   public static Patient getPatientById(int id) {  return registry.get(id);  }   public static void updatePatientMedicalHistory(int id, String medicalHistory) {  Patient patient = registry.get(id);  patient.setMedicalHistory(medicalHistory);  }   // other operations on patients omitted for brevity } |
| --- |

In this example, the PatientRegistry class acts as a centralized registry for all Patient objects in the system. The registry is implemented as a Map that maps patient IDs to patient objects. The registry provides several methods for adding, retrieving, and updating patient objects in the registry. Other classes in the system can use the registry to perform operations on patient objects without having to manage the objects themselves.

|  |
| --- |