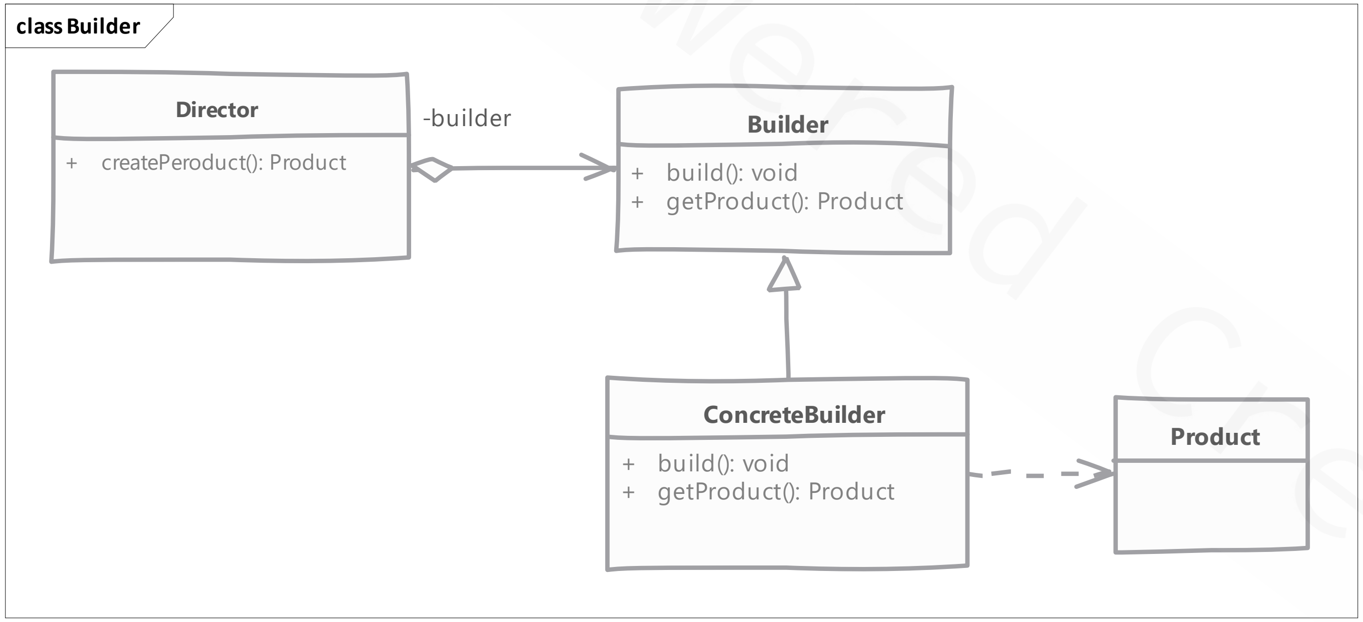
Builder

Creational Design Patterns

Design Patterns in Java

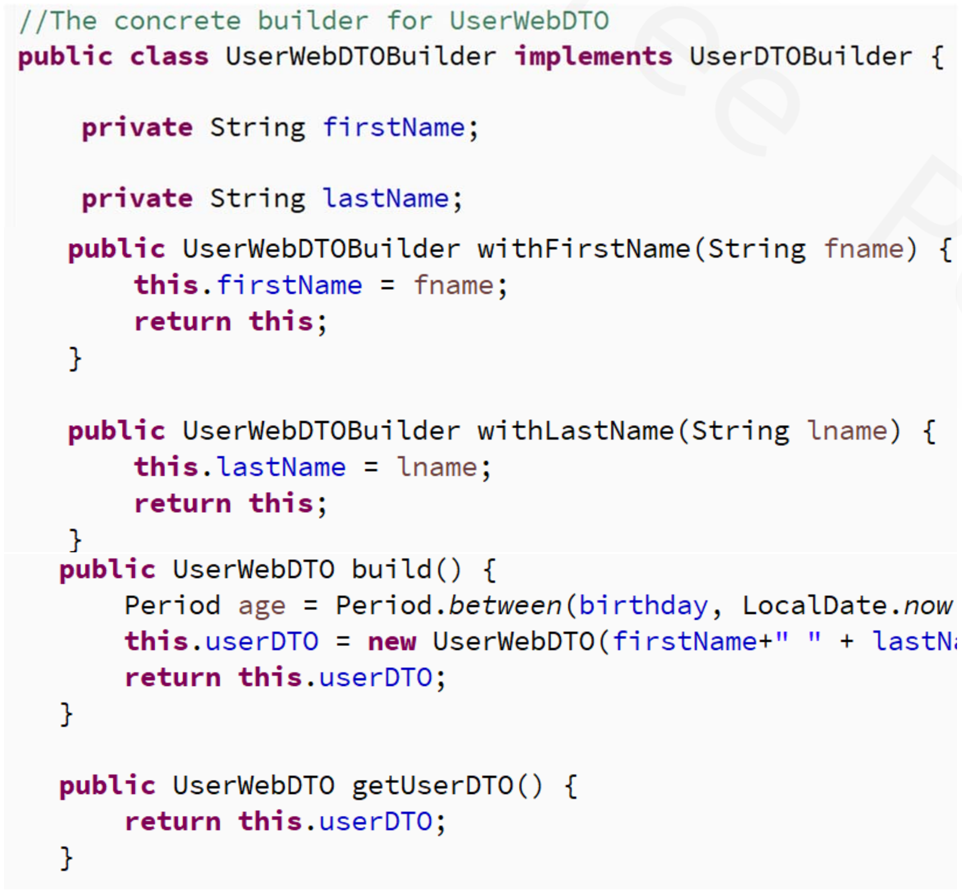
In-A-Hurry Summary

- Think of builder pattern when you have a complex constructor or an object is built in multiple steps.



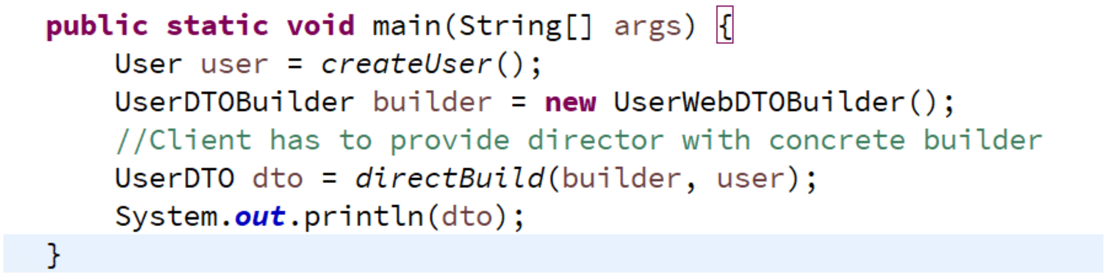
In-A-Hurry Summary

Builder



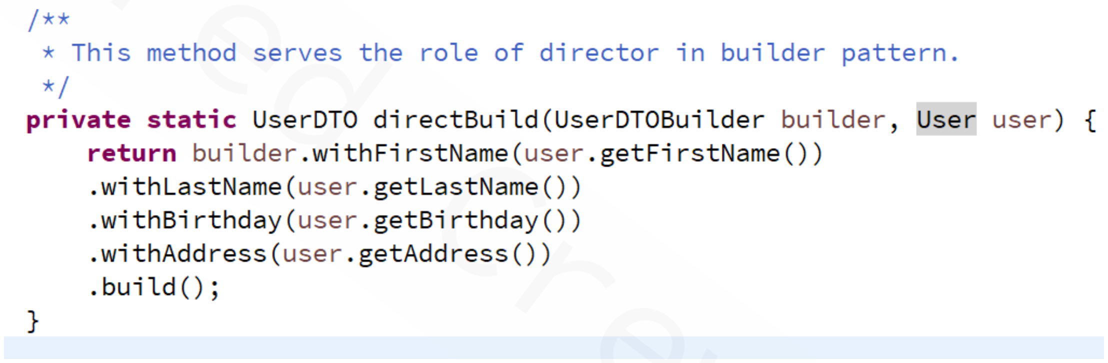
```java
//The concrete builder for UserWebDTO
public class UserWebDTOBuilder implements UserDTOBuilder {

    private String firstName;

    private String lastName;

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

   public UserWebDTOBuilder withLastName(String lname) {
       this.lastName = lname;
       return this;
   }
  public UserWebDTO build() {
      Period age = Period.between(birthday, LocalDate.now
      this.userDTO = new UserWebDTO(firstName+" " + lastN
      return this.userDTO;
  }

  public UserWebDTO getUserDTO() {
      return this.userDTO;
  }
```

Client



```java
public static void main(String[] args) {
    User user = createUser();
    UserDTOBuilder builder = new UserWebDTOBuilder();
    //Client has to provide director with concrete builder
    UserDTO dto = directBuild(builder, user);
    System.out.println(dto);
}
```

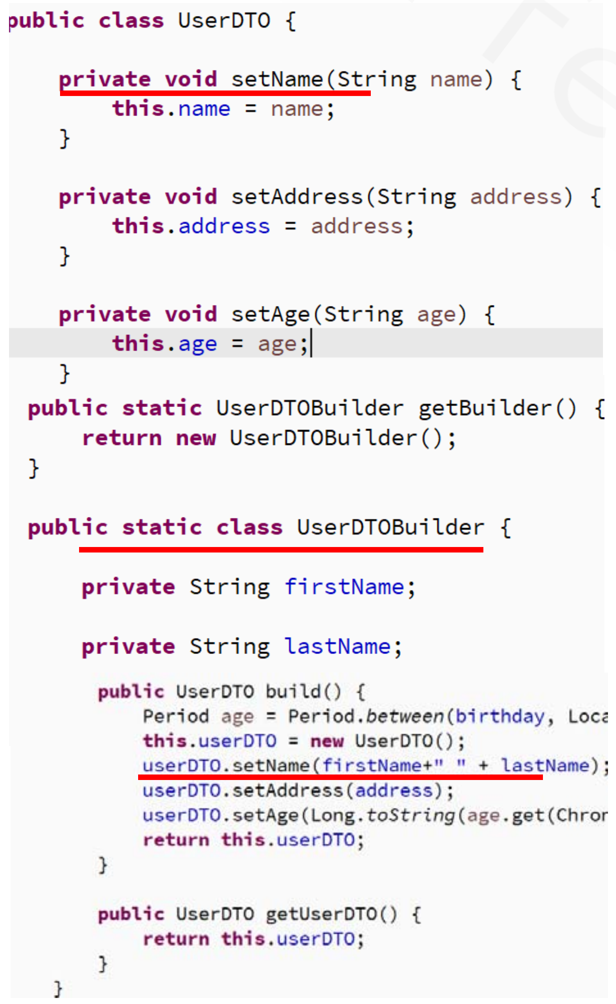
Director (Role played by Client)



```java
/**
 * This method serves the role of director in builder pattern.
 */
private static UserDTO directBuild(UserDTOBuilder builder, User user) {
    return builder.withFirstName(user.getFirstName())
    .withLastName(user.getLastName())
    .withBirthday(user.getBirthday())
    .withAddress(user.getAddress())
    .build();
}
```

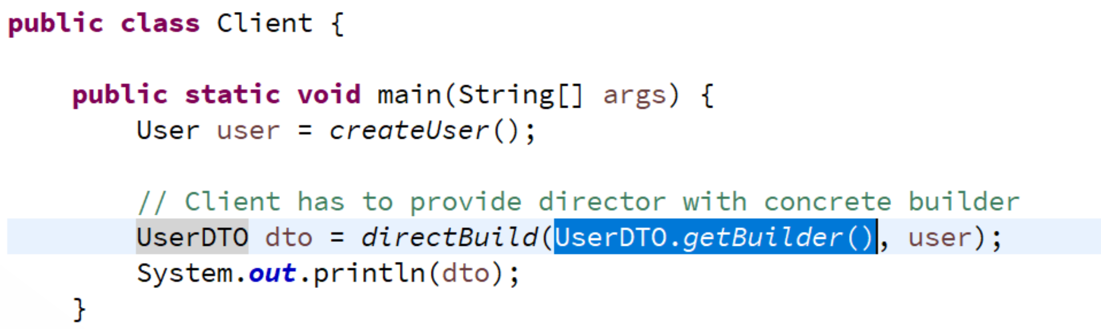
In-A-Hurry Summary

Builder as inner class



```
public class UserDTO {

    private void setName(String name) {
        this.name = name;
    }

    private void setAddress(String address) {
        this.address = address;
    }

    private void setAge(String age) {
        this.age = age;
    }
  public static UserDTOBuilder getBuilder() {
      return new UserDTOBuilder();
  }

  public static class UserDTOBuilder {

      private String firstName;

      private String lastName;

        public UserDTO build() {
            Period age = Period.between(birthday, Loca
            this.userDTO = new UserDTO();
            userDTO.setName(firstName+" " + lastName);
            userDTO.setAddress(address);
            userDTO.setAge(Long.toString(age.get(Chro
            return this.userDTO;
        }

        public UserDTO getUserDTO() {
            return this.userDTO;
        }
    }
```

Client



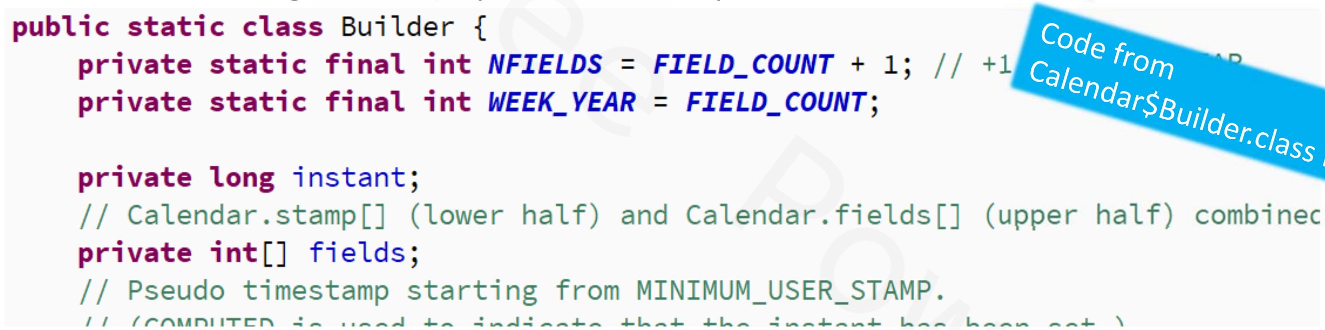
```
public class Client {

    public static void main(String[] args) {
        User user = createUser();

        // Client has to provide director with concrete builder
        UserDTO dto = directBuild(UserDTO.getBuilder(), user);
        System.out.println(dto);
    }
```

Example of a Builder Pattern

- The java.lang.StringBuilder class as well as various buffer classes in java.nio package like ByteBuffer, CharBuffer are often given as examples of builder pattern.
- In my humble opinion they can be given as examples of builder pattern, but with an understanding that they don't match 100% with GoF definition. These classes do allow us to build final object in steps, providing only a part of final object in one step. In this way they can be thought of as an implementation of builder pattern.
- So a StringBuilder satisfies the intent/purpose of builder pattern. However as soon we start looking at structure of the StringBuilder things start to fall apart. GoF definition also states that, builder has potential to build different representations of product interface using same steps.

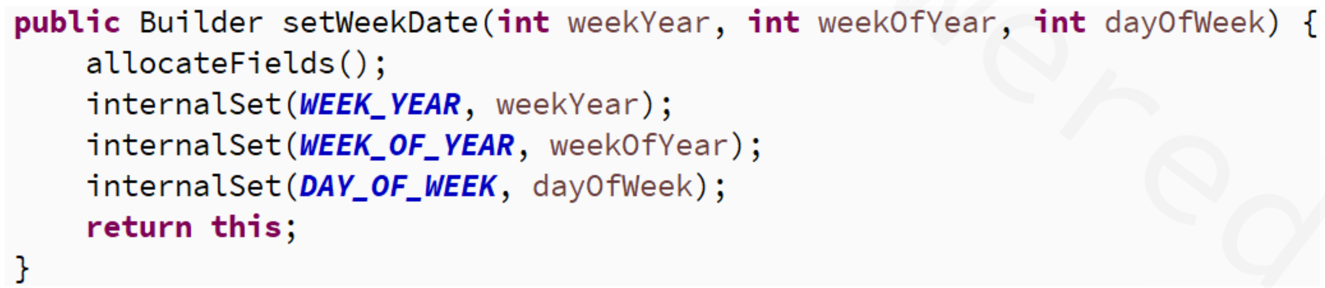
Example of a Builder Pattern

- There is another great example of builder pattern in Java 8. The java.util.Calendar.Builder class is a builder.

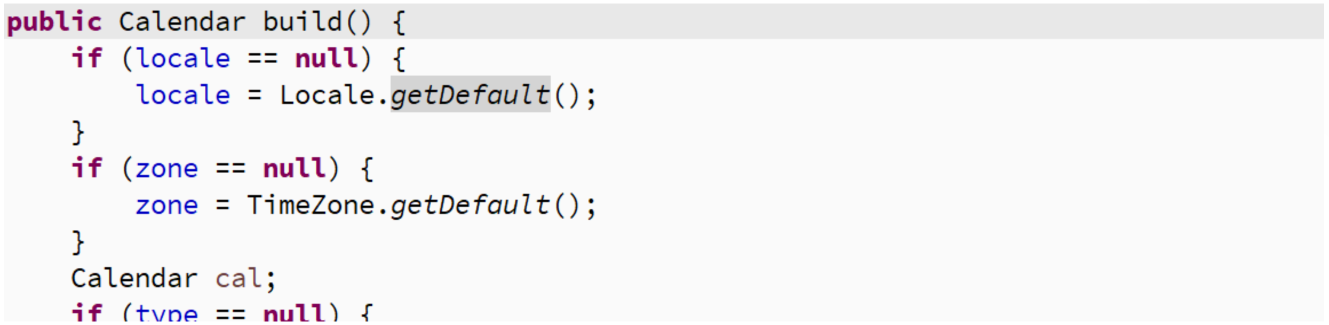


```
public static class Builder {
    private static final int NFIELDS = FIELD_COUNT + 1; // +1
    private static final int WEEK_YEAR = FIELD_COUNT;

    private long instant;
    // Calendar.stamp[] (lower half) and Calendar.fields[] (upper half) combinec
    private int[] fields;
    // Pseudo timestamp starting from MINIMUM_USER_STAMP.
    // (COMPUTED is used to indicate that the instant has been set.)
```

Code from Calendar$Builder.class in rt.jar



```
public Builder setWeekDate(int weekYear, int weekOfYear, int dayOfWeek) {
    allocateFields();
    internalSet(WEEK_YEAR, weekYear);
    internalSet(WEEK_OF_YEAR, weekOfYear);
    internalSet(DAY_OF_WEEK, dayOfWeek);
    return this;
}
```



```
public Calendar build() {
    if (locale == null) {
        locale = Locale.getDefault();
    }
    if (zone == null) {
        zone = TimeZone.getDefault();
    }
    Calendar cal;
    if (type == null) {
```

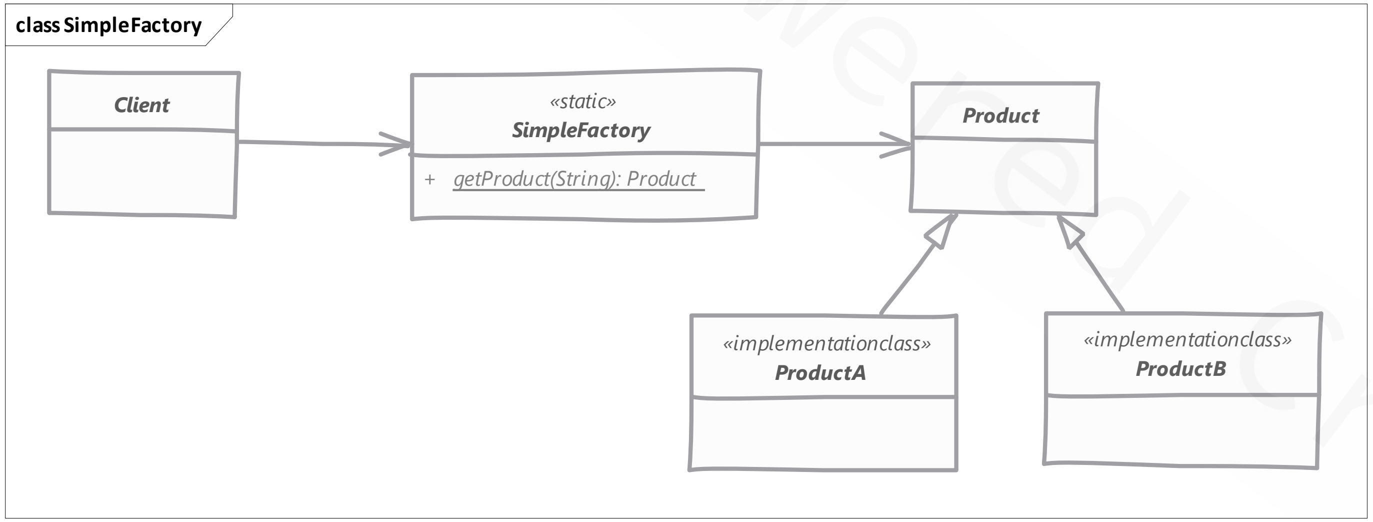
Simple Factory

Creational Design Patterns

Design Patterns in Java

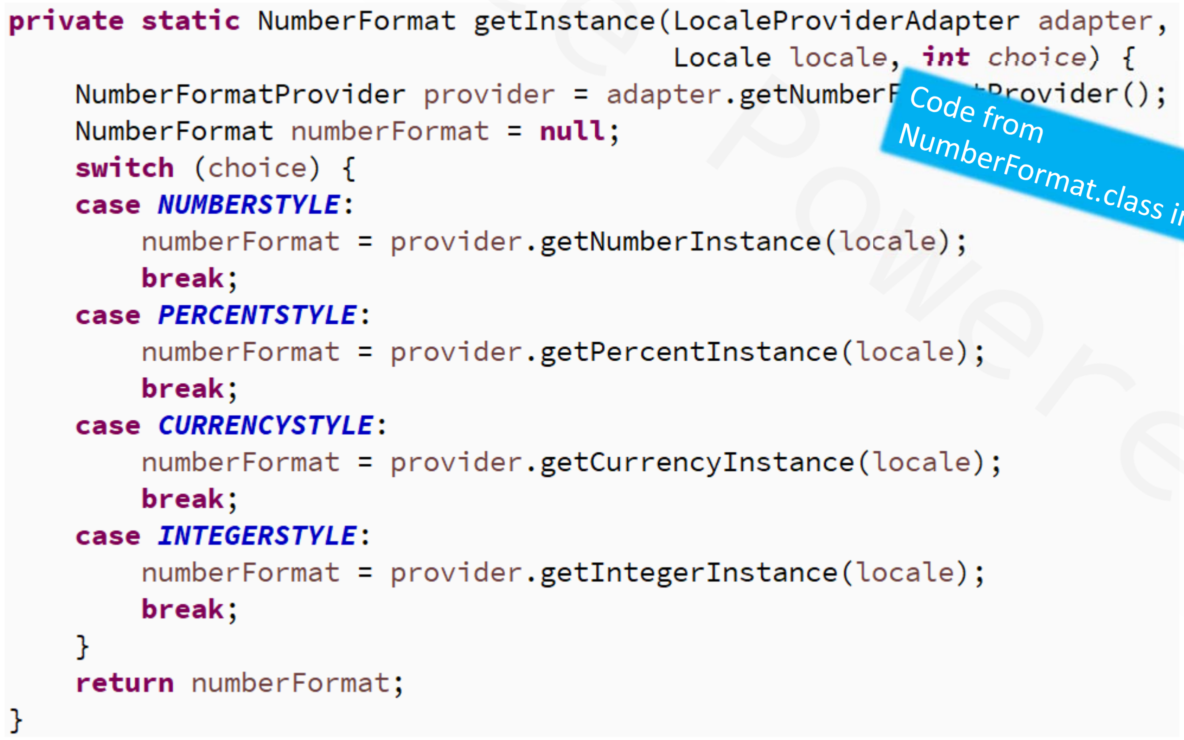
In-A-Hurry Summary

- Simple factory encapsulates away the object instantiation in a separate method.
- We can pass an argument to this method to indicate product type and/or additional arguments to help create objects



Example of a Simple Factory

- The java.text.NumberFormat class has getInstance method, which is an example of simple factory.



```
private static NumberFormat getInstance(LocaleProviderAdapter adapter,
                                        Locale locale, int choice) {
    NumberFormatProvider provider = adapter.getNumberF[illegible]Provider();
    NumberFormat numberFormat = null;
    switch (choice) {
    case NUMBERSTYLE:
        numberFormat = provider.getNumberInstance(locale);
        break;
    case PERCENTSTYLE:
        numberFormat = provider.getPercentInstance(locale);
        break;
    case CURRENCYSTYLE:
        numberFormat = provider.getCurrencyInstance(locale);
        break;
    case INTEGERSTYLE:
        numberFormat = provider.getIntegerInstance(locale);
        break;
    }
    return numberFormat;
}
```

Code from NumberFormat.class in rt.jar

(*Many examples advertised online as “Factory Method” are actually Simple Factories.)

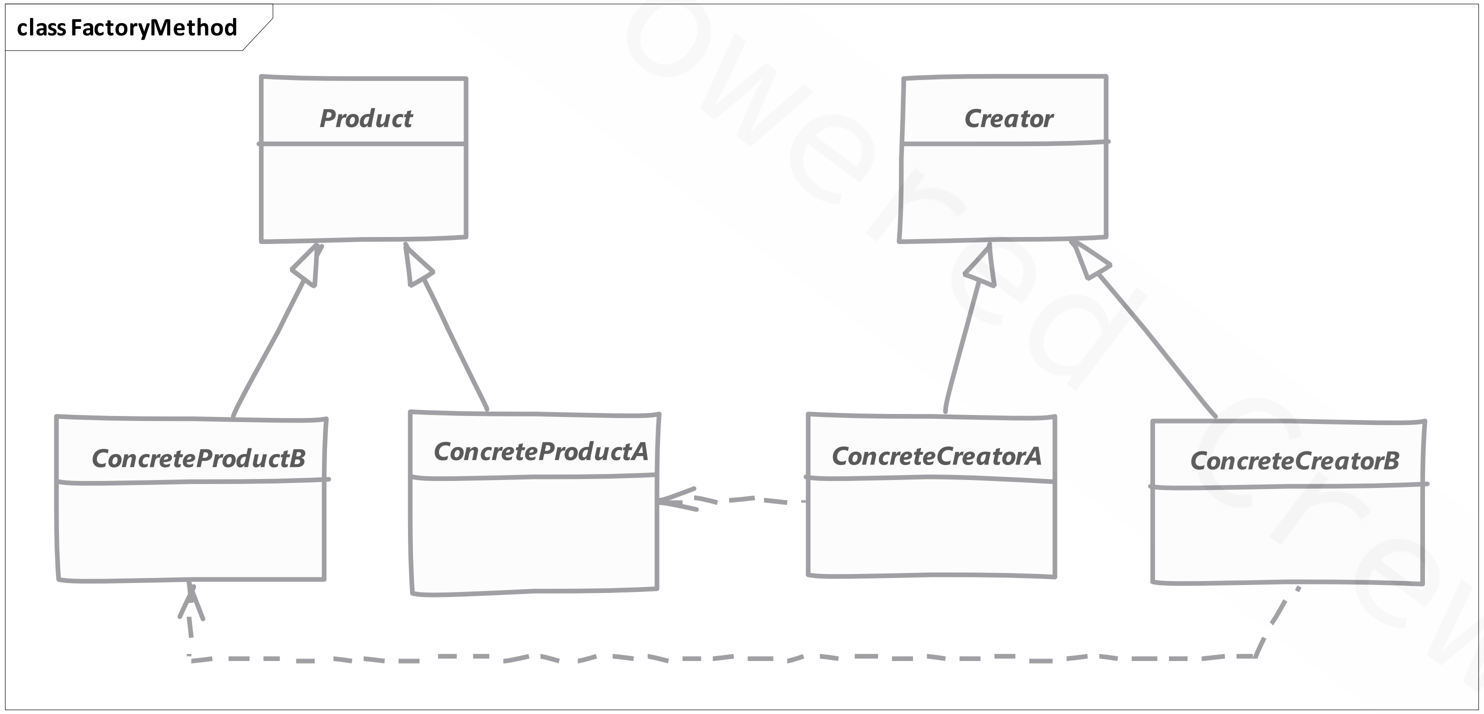
Factory Method

Creational Design Patterns

Design Patterns in Java

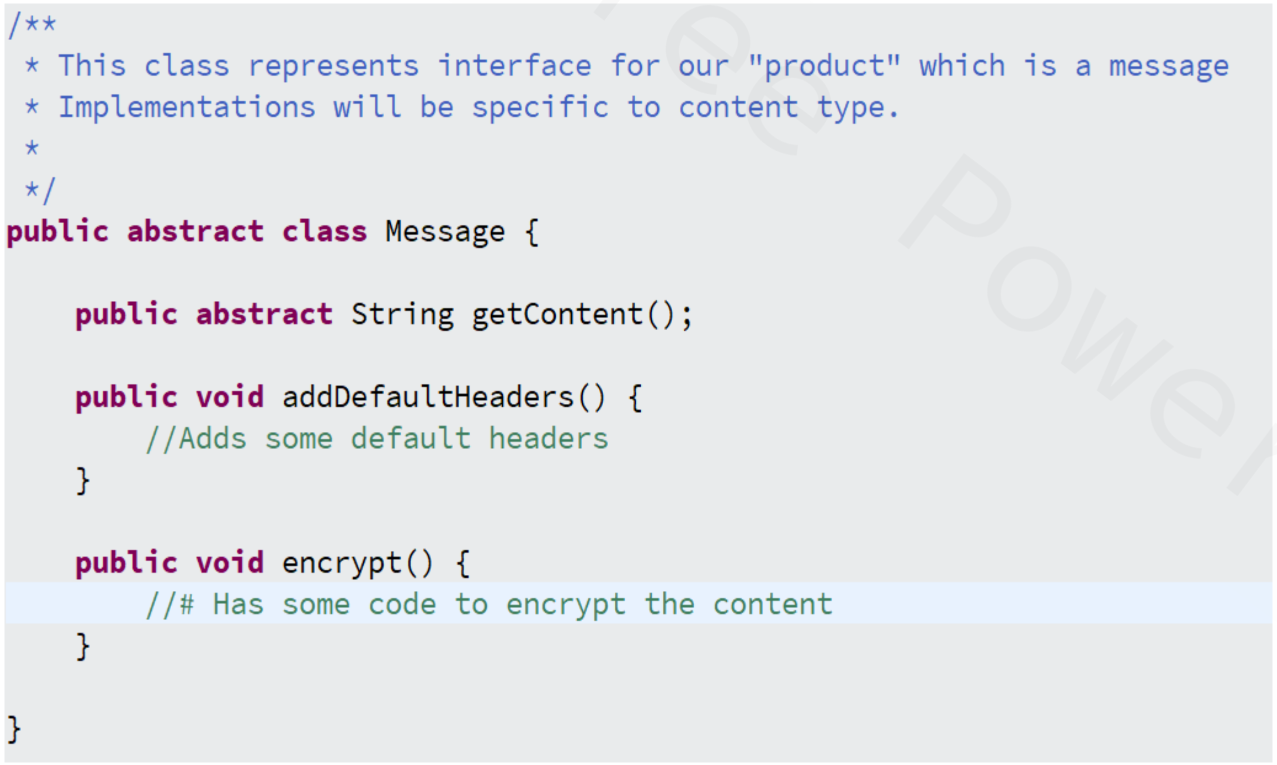
In-A-Hurry Summary

- Use factory method pattern when you want to delegate object instantiation to subclasses, you'd want to do this when you have "product" inheritance hierarchy and possibility of future additions to that.

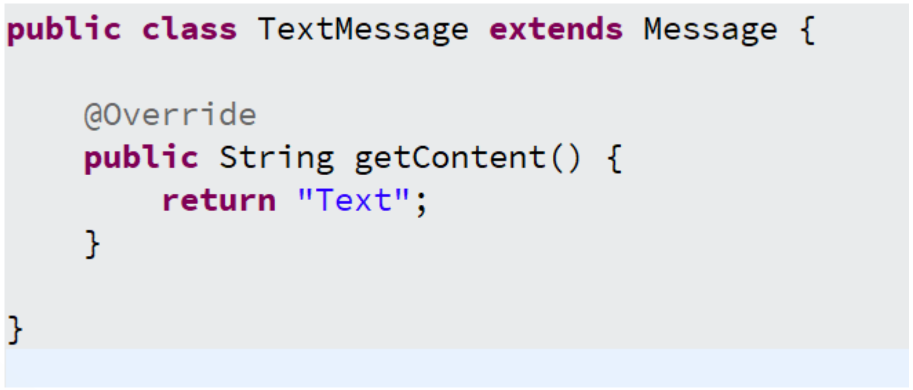


In-A-Hurry Summary

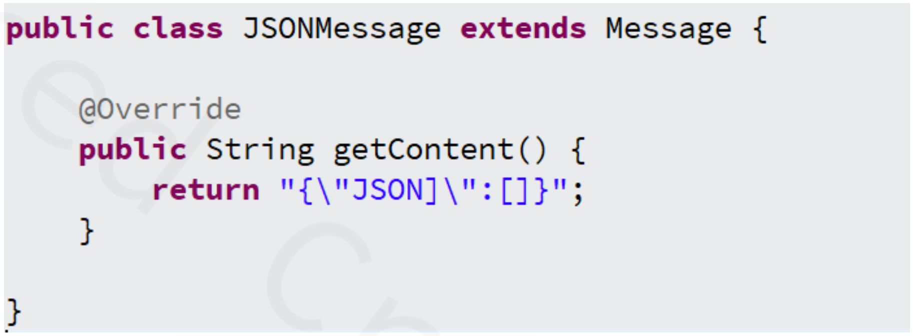
Product



```java
/**
 * This class represents interface for our "product" which is a message
 * Implementations will be specific to content type.
 *
 */
public abstract class Message {

    public abstract String getContent();

    public void addDefaultHeaders() {
        //Adds some default headers
    }

    public void encrypt() {
        //# Has some code to encrypt the content
    }

}
```



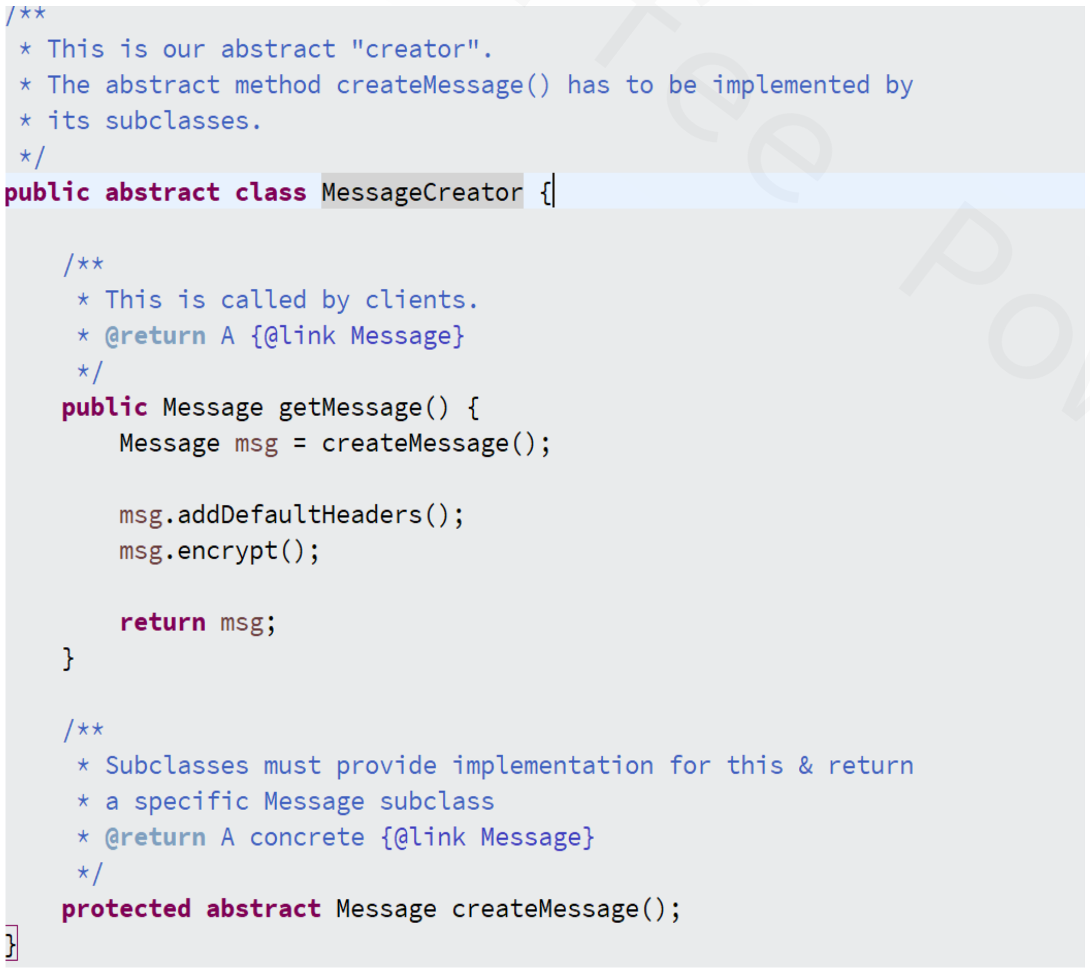
```java
public class TextMessage extends Message {

    @Override
    public String getContent() {
        return "Text";
    }

}
```



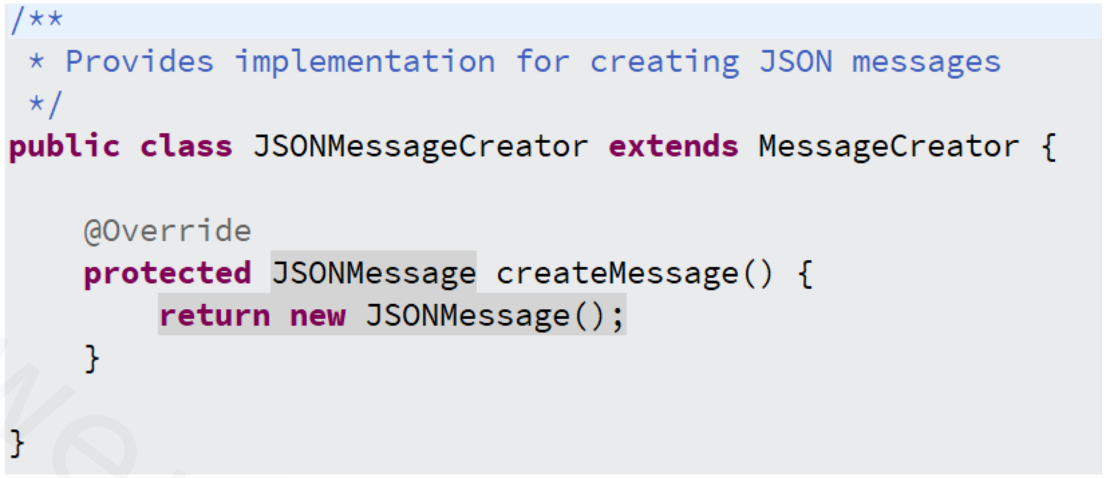
```java
public class JSONMessage extends Message {

    @Override
    public String getContent() {
        return "{\"JSON]\":[]}";
    }

}
```

In-A-Hurry Summary

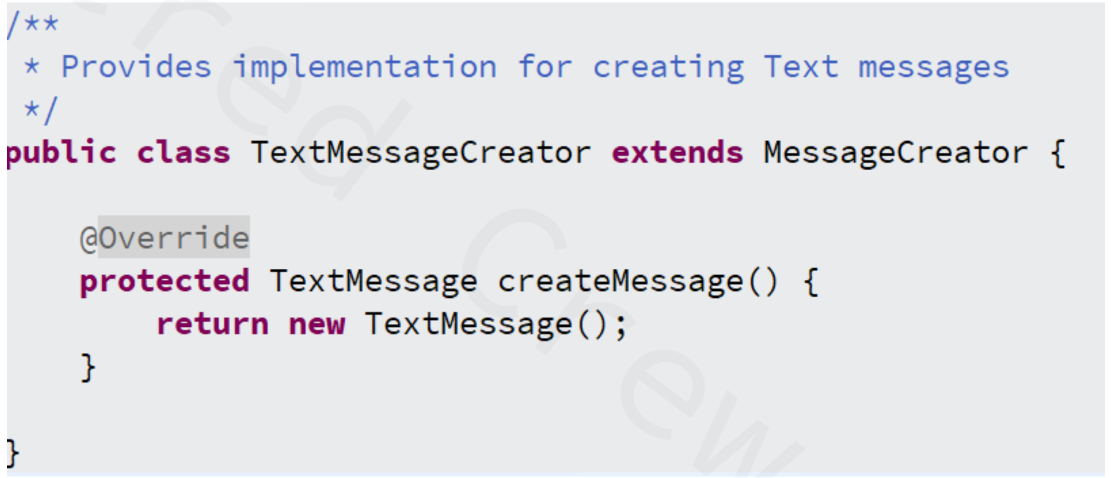
Creator



```java
/**
 * This is our abstract "creator".
 * The abstract method createMessage() has to be implemented by
 * its subclasses.
 */
public abstract class MessageCreator {

    /**
     * This is called by clients.
     * @return A {@link Message}
     */
    public Message getMessage() {
        Message msg = createMessage();

        msg.addDefaultHeaders();
        msg.encrypt();

        return msg;
    }

    /**
     * Subclasses must provide implementation for this & return
     * a specific Message subclass
     * @return A concrete {@link Message}
     */
    protected abstract Message createMessage();
}
```



```java
/**
 * Provides implementation for creating JSON messages
 */
public class JSONMessageCreator extends MessageCreator {

    @Override
    protected JSONMessage createMessage() {
        return new JSONMessage();
    }

}
```



```java
/**
 * Provides implementation for creating Text messages
 */
public class TextMessageCreator extends MessageCreator {

    @Override
    protected TextMessage createMessage() {
        return new TextMessage();
    }

}
```

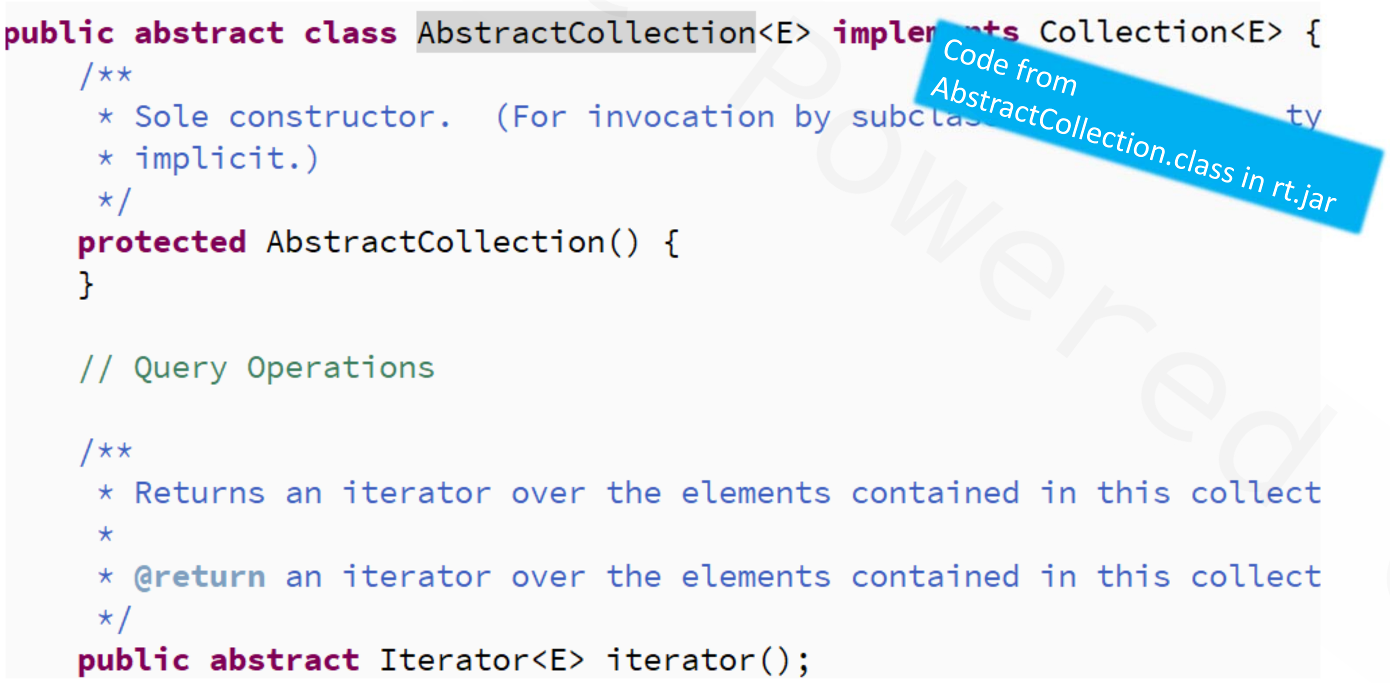
In-A-Hurry Summary

Client

```
public static void main(String[] args) {
    //Using creator to create a product, choice of creator determines
    //type of product created
    printMessage(new JSONMessageCreator());
    //Using another creator to create another product
    printMessage(new TextMessageCreator());
}

public static void printMessage(MessageCreator creator) {
    Message msg = creator.getMessage();
    System.out.println(msg.getContent());
}
```

Examples of a Factory Method

- The java.util.Collection (or java.util.AbstractCollection) has an abstract method called iterator(). This method is an example of a factory method.



```
public abstract class AbstractCollection<E> implem      nts Collection<E> {
    /**
     * Sole constructor.  (For invocation by subcla                    ty
     * implicit.)
     */
    protected AbstractCollection() {
    }

    // Query Operations

    /**
     * Returns an iterator over the elements contained in this collect
     *
     * @return an iterator over the elements contained in this collect
     */
    public abstract Iterator<E> iterator();
```

- Remember, the most defining characteristic of factory method pattern is “subclasses providing the actual instance”. So static methods returning object instances are technically not GoF factory methods.

Prototype

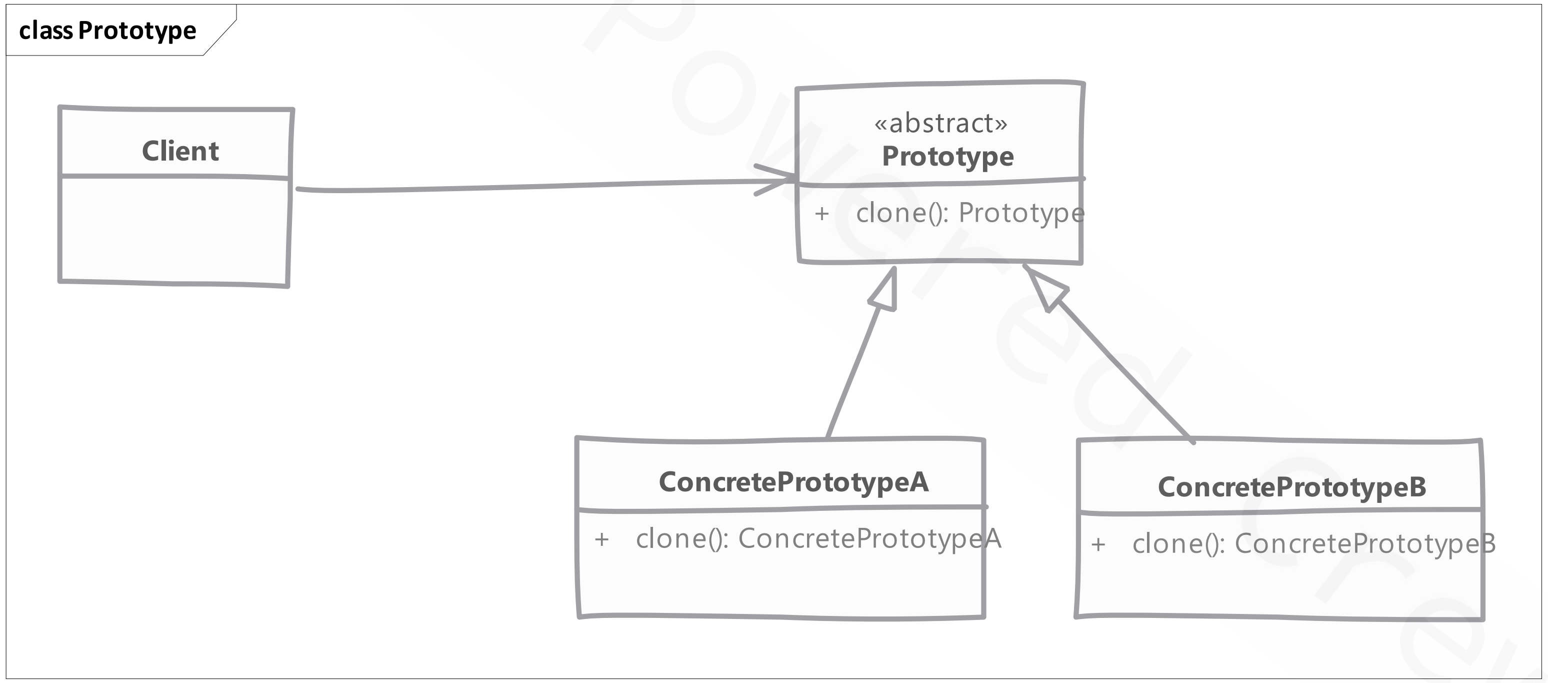
Creational Design Patterns

Design Patterns in Java

In-A-Hurry Summary

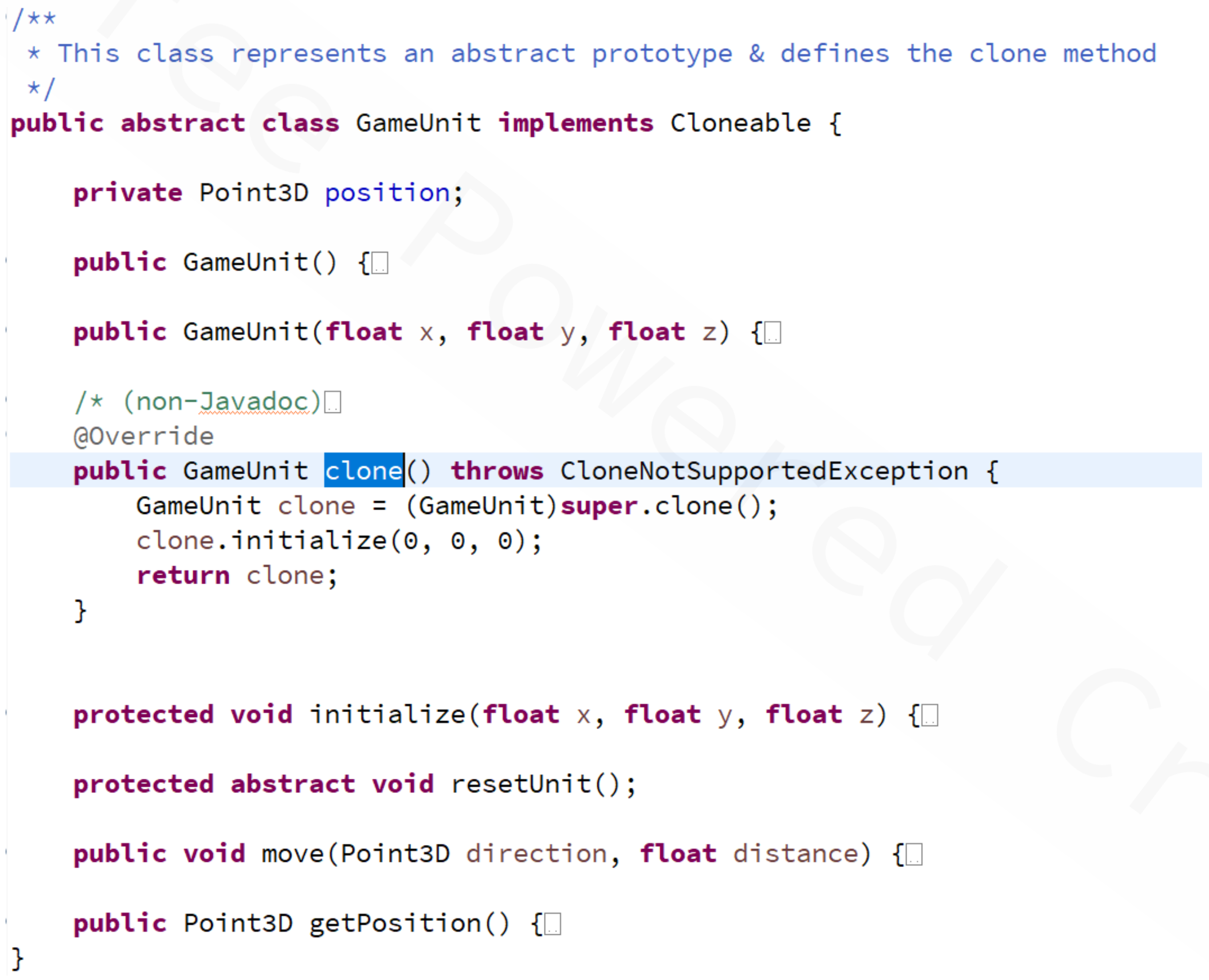
- Think of prototype pattern when you have an object where construction of a new instance is costly or not possible (object is supplied to your code).
- In Java we typically implement this pattern with clone method.
- Objects which have a majority of their state as immutable are good candidates for prototypes.
- When implementing clone method pay attention to the requirement of deep or shallow copy of object state.
- Also we’ve to insure that clone is “initialized”; that is appropriate states are reset before returning the copy to outside world.

In-A-Hurry Summary



In-A-Hurry Summary

Prototype



```java
/**
 * This class represents an abstract prototype & defines the clone method
 */
public abstract class GameUnit implements Cloneable {

    private Point3D position;

    public GameUnit() {

    public GameUnit(float x, float y, float z) {

    /* (non-Javadoc)
    @Override
    public GameUnit clone() throws CloneNotSupportedException {
        GameUnit clone = (GameUnit)super.clone();
        clone.initialize(0, 0, 0);
        return clone;
    }


    protected void initialize(float x, float y, float z) {

    protected abstract void resetUnit();

    public void move(Point3D direction, float distance) {

    public Point3D getPosition() {
}
```

Example of a Prototype

- Actually the Object.clone() method is an example of a prototype!
- This method is provided by Java and can clone an existing object, thus allowing any object to act as a prototype. Classes still need to be Cloneable but the method does the job of cloning object.
- We also use the same method to implement our own prototype, as it is a well known method and has same meaning as we want in a prototype method.

Abstract Factory

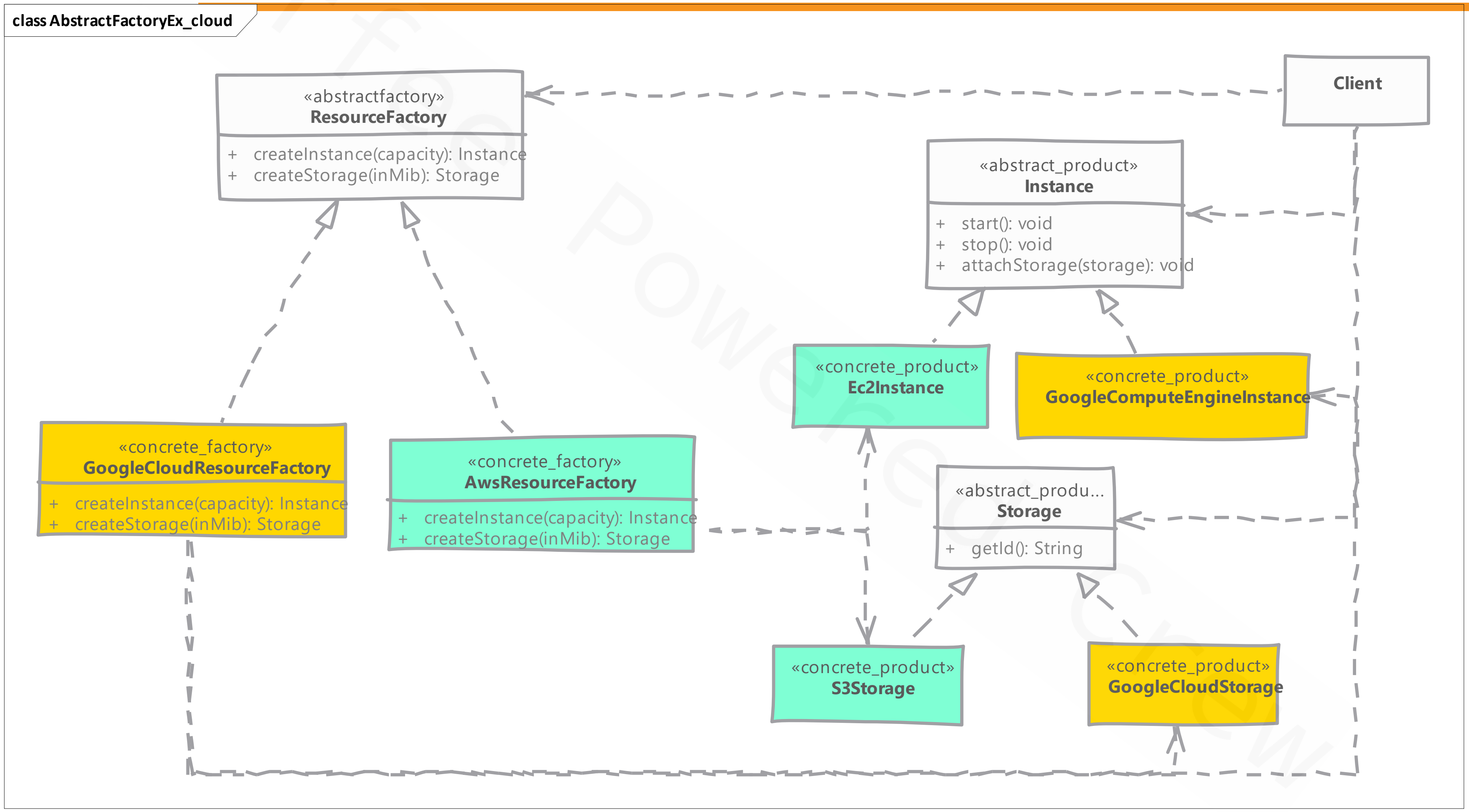
Creational Design Patterns

Design Patterns in Java

In-A-Hurry Summary

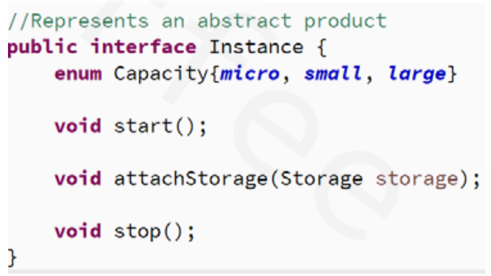
- When you have multiple sets of objects where objects in one set work together then you can use abstract factory pattern to isolate client code from concrete objects & their factories.
- Abstract factory itself uses factory method pattern and you can think of them as objects with multiple factory methods.
- Adding a new product type needs changes to base factory and all its implementations.
- Concrete factories can be singleton as we need only one instance of them in code.
- We provide client code with concrete factory instance. Factories can be changed at runtime.

In-A-Hurry Summary

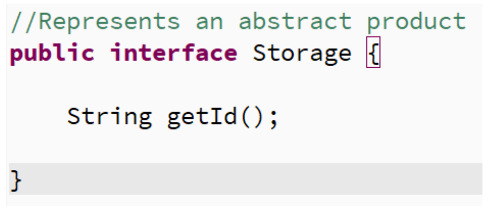

class AbstractFactoryEx_cloud
«abstractfactory»
ResourceFactory
+ createInstance(capacity): Instance
+ createStorage(inMib): Storage
Client
«abstract_product»
Instance
+ start(): void
+ stop(): void
+ attachStorage(storage): void
«concrete_product»
Ec2Instance
«concrete_product»
GoogleComputeEngineInstance
«concrete_factory»
GoogleCloudResourceFactory
+ createInstance(capacity): Instance
+ createStorage(inMib): Storage
«concrete_factory»
AwsResourceFactory
+ createInstance(capacity): Instance
+ createStorage(inMib): Storage
«abstract_produ...
Storage
+ getId(): String
«concrete_product»
S3Storage
«concrete_product»
GoogleCloudStorage

In-A-Hurry Summary

Abstract Products

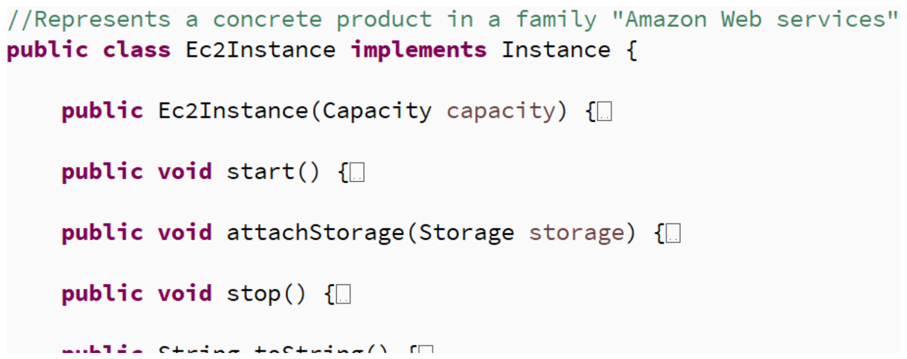


```
//Represents an abstract product
public interface Instance {
    enum Capacity{micro, small, large}

    void start();

    void attachStorage(Storage storage);

    void stop();
}
```

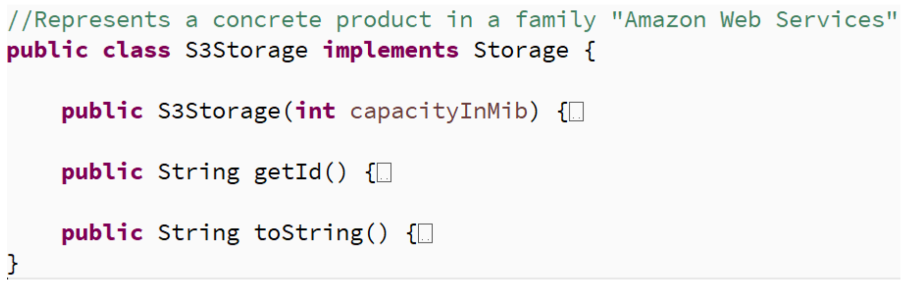


```
//Represents an abstract product
public interface Storage {

    String getId();

}
```

“Amazon Web Services” Product Family

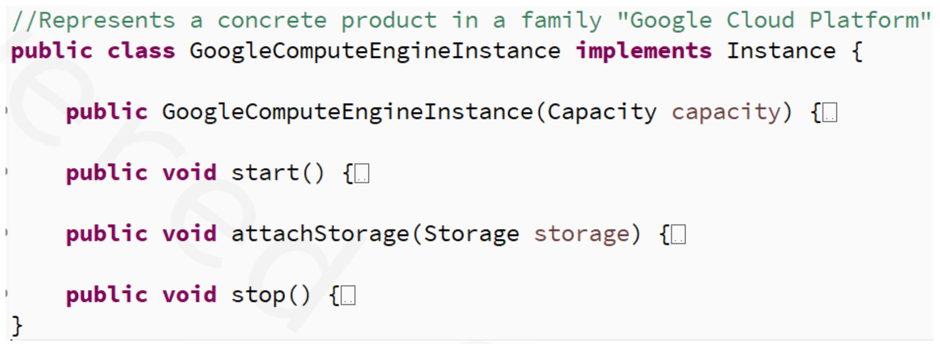


```
//Represents a concrete product in a family "Amazon Web services"
public class Ec2Instance implements Instance {

    public Ec2Instance(Capacity capacity) {

    public void start() {

    public void attachStorage(Storage storage) {

    public void stop() {

    public String toString() {
```

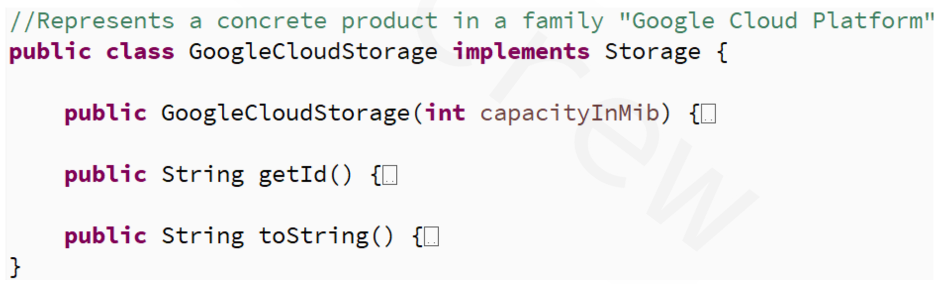


```
//Represents a concrete product in a family "Amazon Web Services"
public class S3Storage implements Storage {

    public S3Storage(int capacityInMib) {

    public String getId() {

    public String toString() {
}
```

“Google Cloud Platform” Product Family



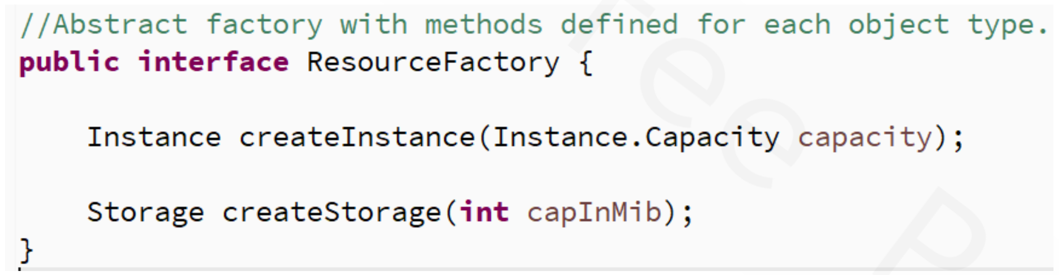
```
//Represents a concrete product in a family "Google Cloud Platform"
public class GoogleComputeEngineInstance implements Instance {

    public GoogleComputeEngineInstance(Capacity capacity) {

    public void start() {

    public void attachStorage(Storage storage) {

    public void stop() {
}
```



```
//Represents a concrete product in a family "Google Cloud Platform"
public class GoogleCloudStorage implements Storage {

    public GoogleCloudStorage(int capacityInMib) {

    public String getId() {

    public String toString() {
}
```

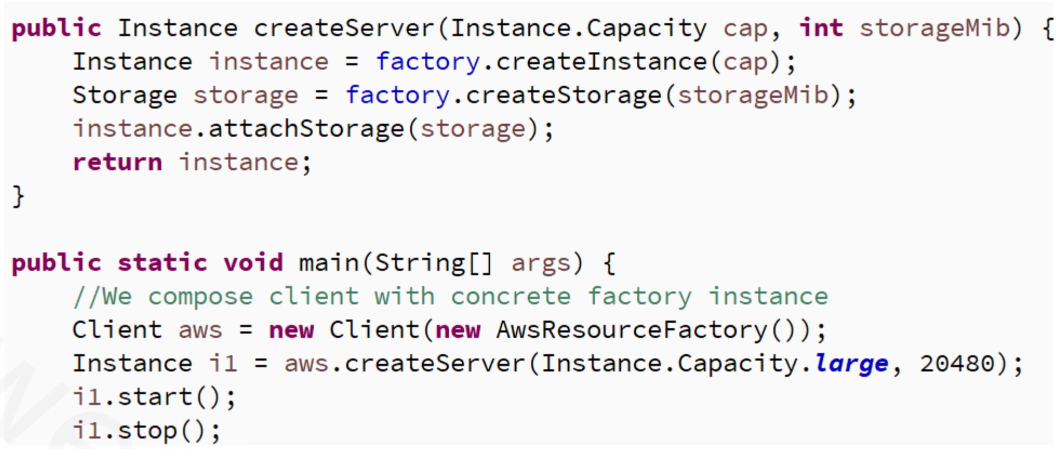
In-A-Hurry Summary

Abstract Factory



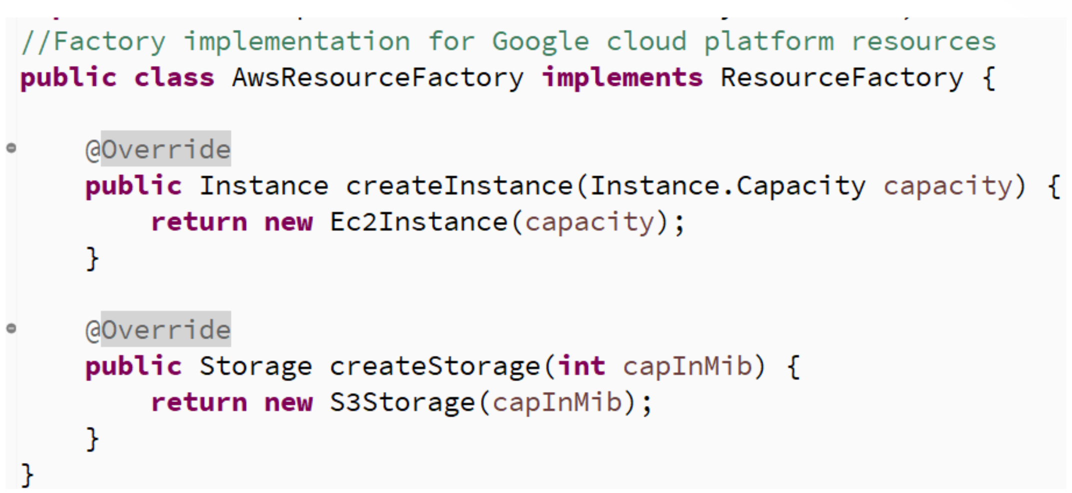
```java
//Abstract factory with methods defined for each object type.
public interface ResourceFactory {

    Instance createInstance(Instance.Capacity capacity);

    Storage createStorage(int capInMib);
}
```

Client

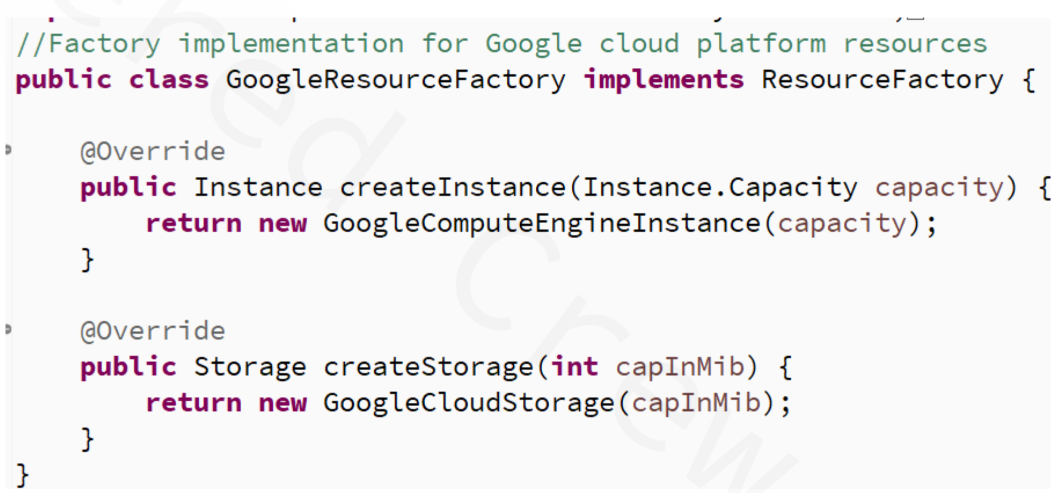


```java
public Instance createServer(Instance.Capacity cap, int storageMib) {
    Instance instance = factory.createInstance(cap);
    Storage storage = factory.createStorage(storageMib);
    instance.attachStorage(storage);
    return instance;
}

public static void main(String[] args) {
    //We compose client with concrete factory instance
    Client aws = new Client(new AwsResourceFactory());
    Instance i1 = aws.createServer(Instance.Capacity.large, 20480);
    i1.start();
    i1.stop();
```

Implementations of Abstract Factory



```java
//Factory implementation for Google cloud platform resources
public class AwsResourceFactory implements ResourceFactory {

    @Override
    public Instance createInstance(Instance.Capacity capacity) {
        return new Ec2Instance(capacity);
    }

    @Override
    public Storage createStorage(int capInMib) {
        return new S3Storage(capInMib);
    }
}
```



```java
//Factory implementation for Google cloud platform resources
public class GoogleResourceFactory implements ResourceFactory {

    @Override
    public Instance createInstance(Instance.Capacity capacity) {
        return new GoogleComputeEngineInstance(capacity);
    }

    @Override
    public Storage createStorage(int capInMib) {
        return new GoogleCloudStorage(capInMib);
    }
}
```

Example of an Abstract Factory

- The javax. xml.parsers.DocumentBuilderFactory is a good example of an abstract factory pattern.
- However this implementation doesn’t match 100% with the UML of abstract factory from GoF. The class has a ***static*** newInstance() method which returns actual factory class object.
- The newInstance() method however uses classpath scanning, system properties, an external property file as ways to find the factory class & creates the factory object. So we *can* change the factory class being used, even if this is a static method.
- Another great example is the java.util.prefs.PreferencesFactory interface which matches more closely with the UML we studied.

Singleton

Creational Design Patterns

Design Patterns in Java

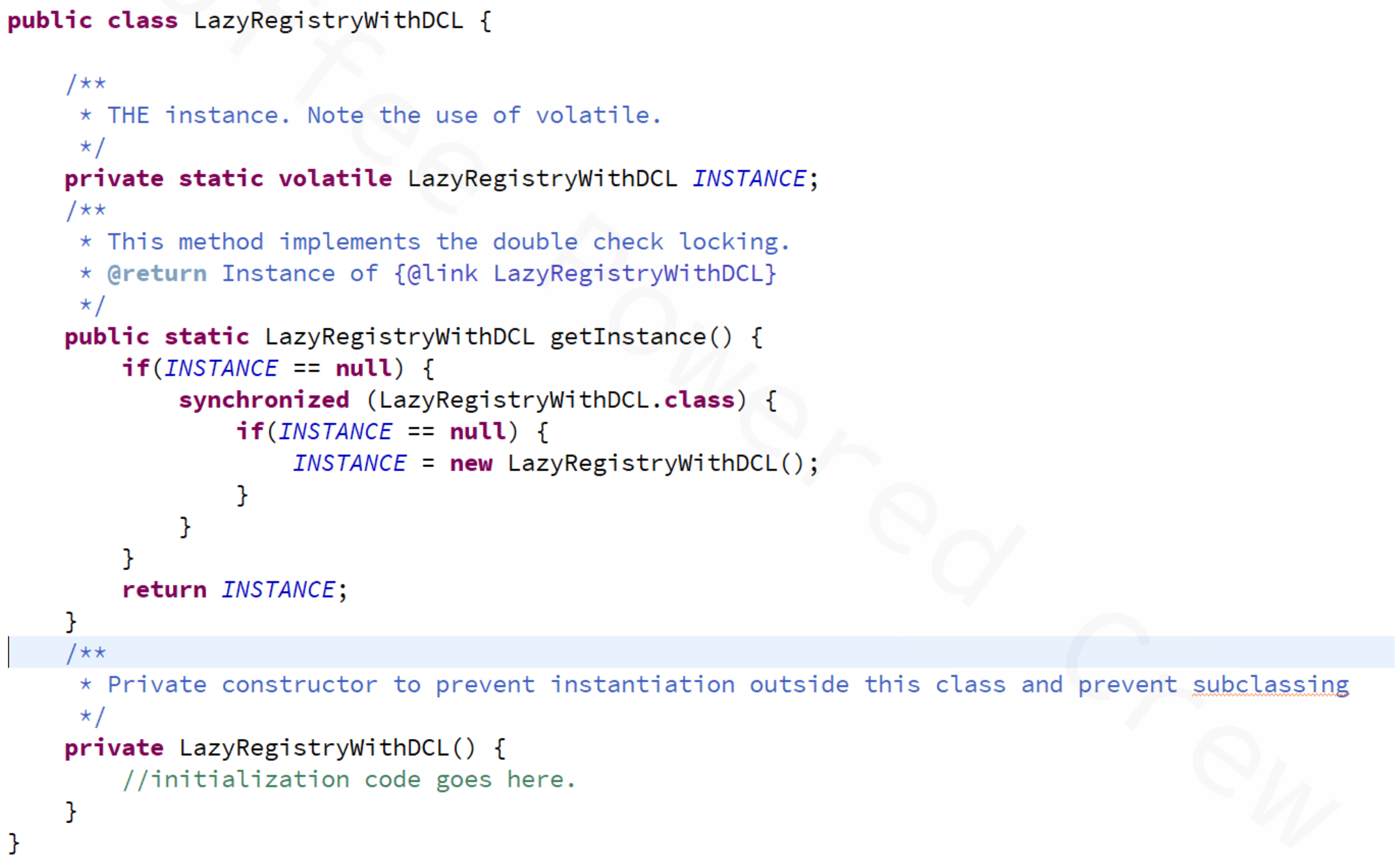
In-A-Hurry Summary

- Singleton pattern is used when you want to ensure that only one instance of a class exists in application.
- In Java we achieve this by making constructor private, this also prevents inheritance & providing a public static method which returns the singleton instance
- Implementation wise we have two broad choices –
  1. In eager loading singleton, we create instance as soon as class is loaded by classloader.
  2. In lazy loading singleton, we defer creation until some code actually requests the instance.
- Always prefer the eager loading instance unless creation cost is high and start-up time impact is noticeable.

• Eager Singleton

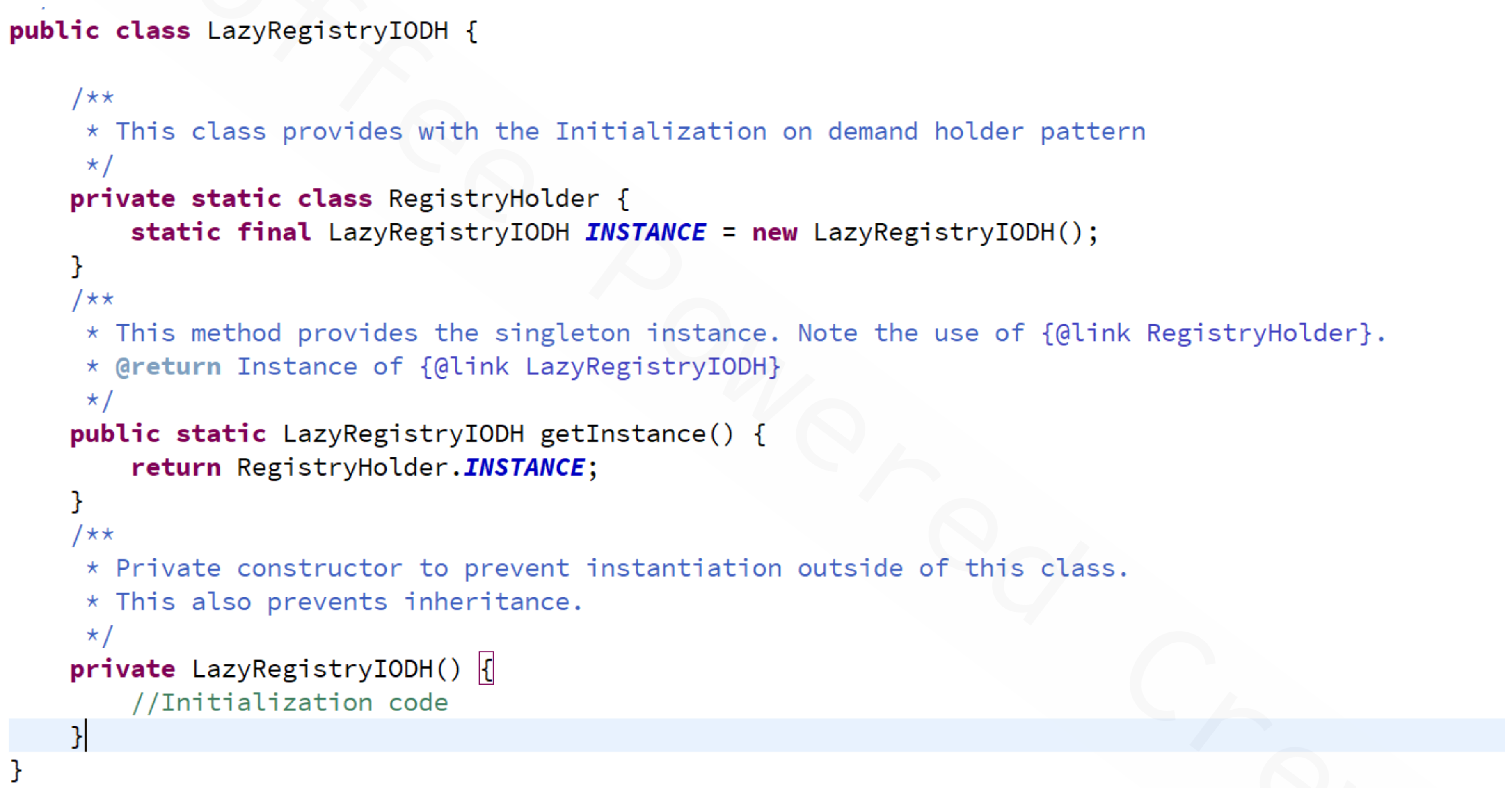
```java
/**
 * This class uses eager initialization of singleton instance.
 */
public class EagerRegistry {

    /**
     * The single instance. Eagerly initialized singleton
     */
    private static final EagerRegistry INSTANCE = new EagerRegistry();

    /**
     * This method returns the singleton instance to outside world.
     * @return Instance of EagerRegistry
     */
    public static EagerRegistry getInstance() {
        return INSTANCE;
    }

    /**
     * By making constructor private, we prevent object instantiation outside of this class
     * & this will also prevent inheritance
     */
    private EagerRegistry() {
        //initialization code
    }

}
```

• Lazy Singleton with Double check Locking



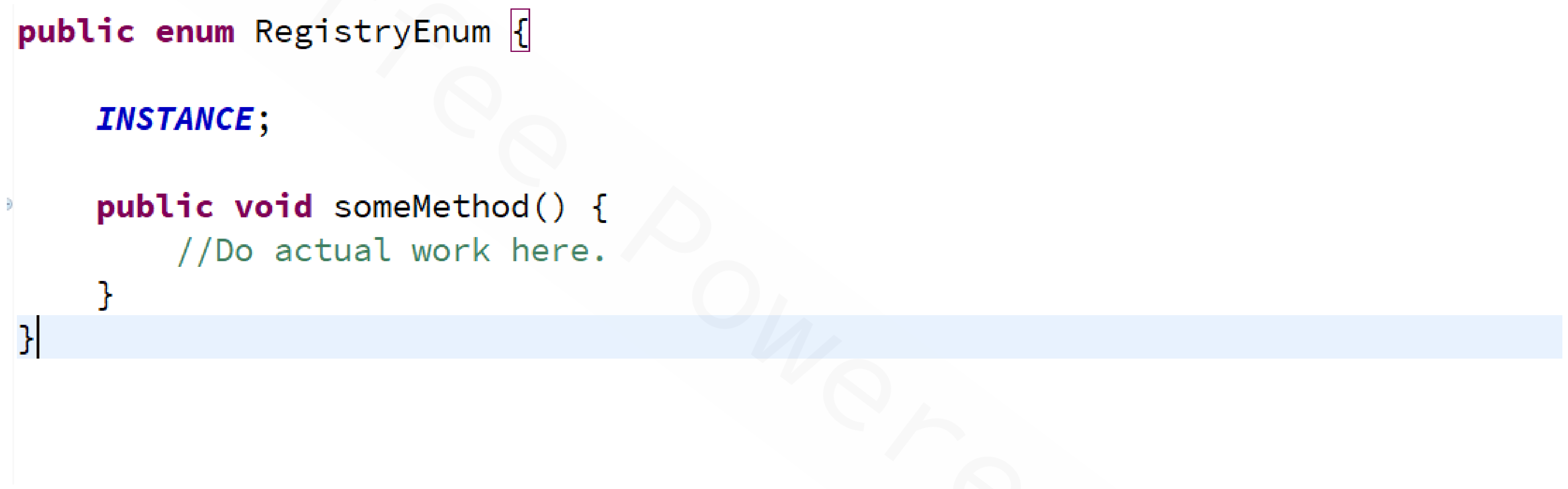
```java
public class LazyRegistryWithDCL {

    /**
     * THE instance. Note the use of volatile.
     */
    private static volatile LazyRegistryWithDCL INSTANCE;
    /**
     * This method implements the double check locking.
     * @return Instance of {@link LazyRegistryWithDCL}
     */
    public static LazyRegistryWithDCL getInstance() {
        if(INSTANCE == null) {
            synchronized (LazyRegistryWithDCL.class) {
                if(INSTANCE == null) {
                    INSTANCE = new LazyRegistryWithDCL();
                }
            }
        }
        return INSTANCE;
    }
    /**
     * Private constructor to prevent instantiation outside this class and prevent subclassing
     */
    private LazyRegistryWithDCL() {
        //initialization code goes here.
    }
}
```

• Lazy Singleton with Initialization Holder



```java
public class LazyRegistryIODH {

    /**
     * This class provides with the Initialization on demand holder pattern
     */
    private static class RegistryHolder {
        static final LazyRegistryIODH INSTANCE = new LazyRegistryIODH();
    }
    /**
     * This method provides the singleton instance. Note the use of {@link RegistryHolder}.
     * @return Instance of {@link LazyRegistryIODH}
     */
    public static LazyRegistryIODH getInstance() {
        return RegistryHolder.INSTANCE;
    }
    /**
     * Private constructor to prevent instantiation outside of this class.
     * This also prevents inheritance.
     */
    private LazyRegistryIODH() {
        //Initialization code
    }
}
```

- Singleton as enum



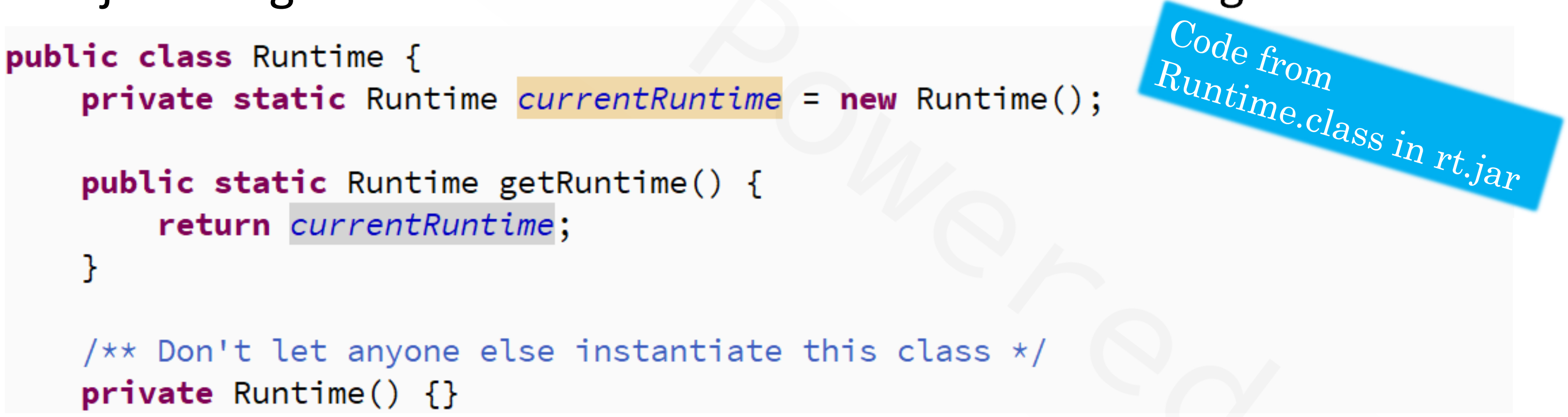
```java
public enum RegistryEnum {

    INSTANCE;

    public void someMethod() {
        //Do actual work here.
    }
}
```

In-A-Hurry Summary

- There are *very few* situations where a Singleton is really a good choice.
- Application configuration values can be tracked in a singleton. Typically these are read from file at start and then referred to by other parts of application.
- Logging frameworks also make use of Singleton pattern.
- Spring framework treats all beans by default as singletons. In spring we don’t have to make any changes to ensure single instance, Spring handles that for us.

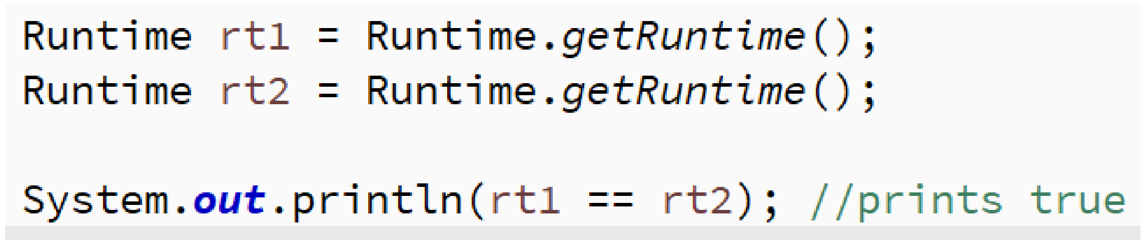
Example of a Singleton Pattern

- The java.lang.Runtime class in standard Java API is a singleton.



```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}
```

Code from Runtime.class in rt.jar

- 

```
Runtime rt1 = Runtime.getRuntime();
Runtime rt2 = Runtime.getRuntime();

System.out.println(rt1 == rt2); //prints true
```

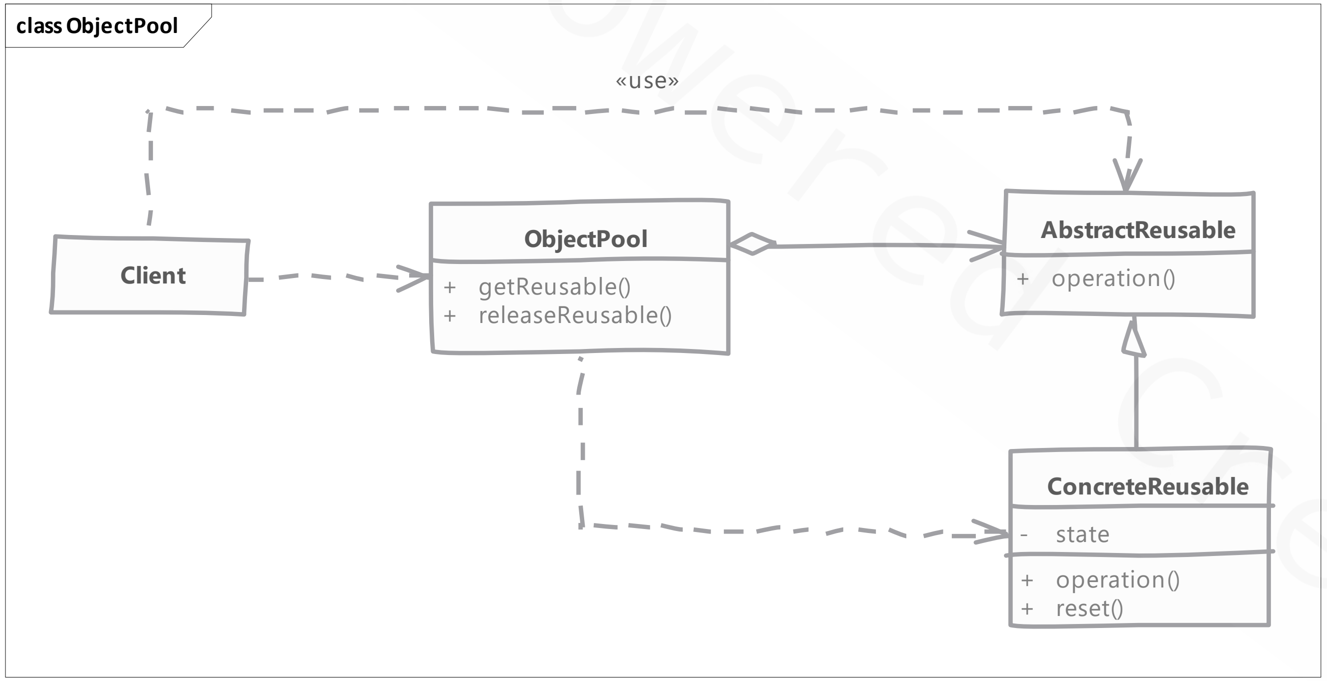
Object Pool

Creational Design Patterns

Design Patterns in Java

In-A-Hurry Summary

- If cost of creating instances of a class is very high and you need many such objects throughout your application for short duration then you can pool them with object pool.
- Typically objects that represent fixed external system resources like threads, connections or other system resources are good candidates for pooling

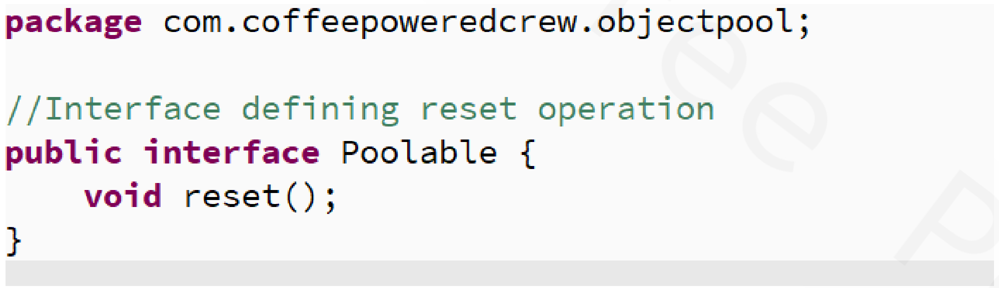


In-A-Hurry Summary

- Objects to be pooled should provide a method to “reset” their state so they can be reused. This operation should be efficient as well, otherwise release operation will be costly.
- Pool must handle synchronization issues efficiently and reset object state before adding them to pool for reuse.
- Client code must release pooled objects back into the pool so they can be reused. Failing to do so will break the system. Thread pools can work around this since the a thread can know when its work is done.
- Difficult to optimize as pools are sensitive to system load at runtime (demand of pooled objects).
- Pools are good choice when the pooled objects represent a fixed quantity of externally available resource like thread or a connection.
- If you create objects when pool is empty then we have to make sure that pool size is maintained or else we can end up with large pool

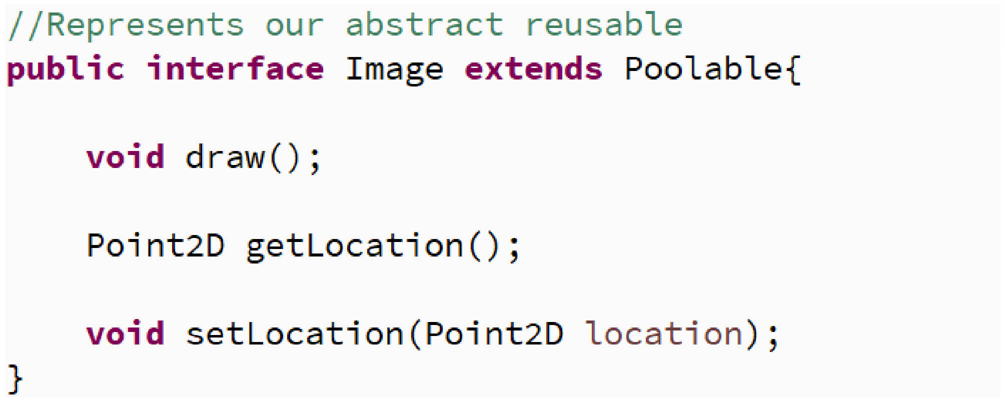
In-A-Hurry Summary

Poolable



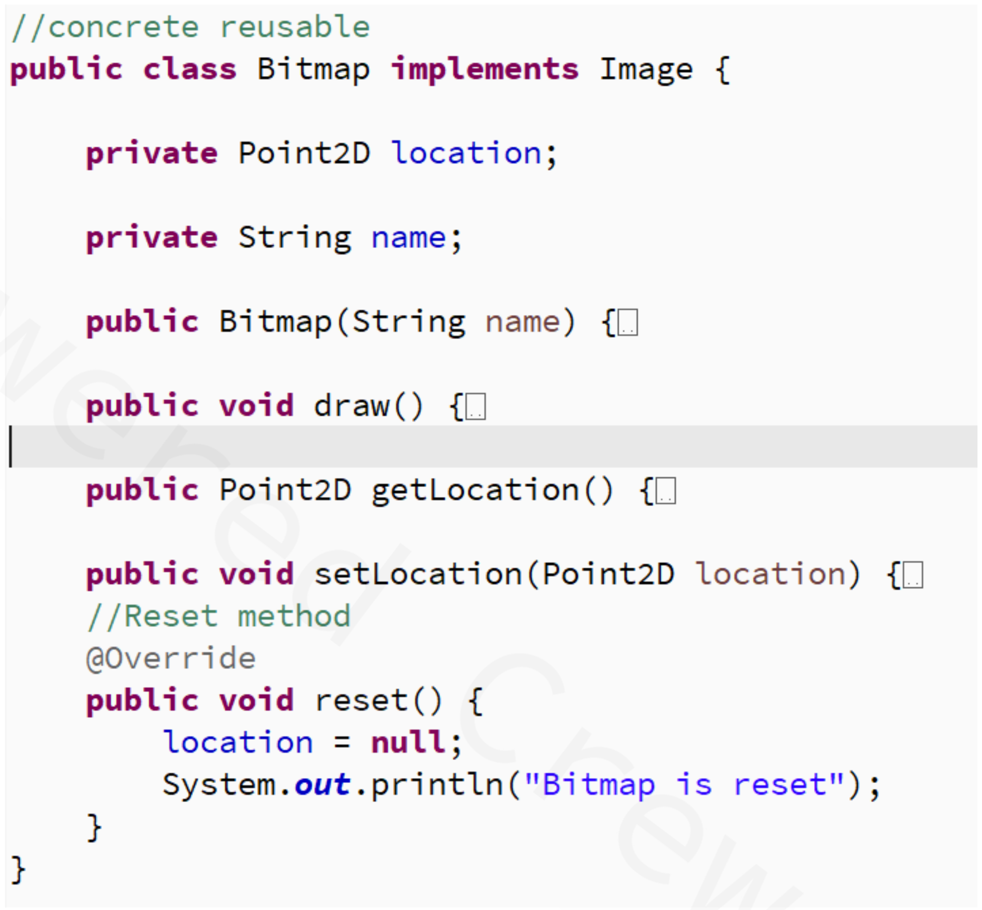
```
package com.coffeepoweredcrew.objectpool;

//Interface defining reset operation
public interface Poolable {
    void reset();
}
```

Reusable Product



```
//Represents our abstract reusable
public interface Image extends Poolable{

    void draw();

    Point2D getLocation();

    void setLocation(Point2D location);
}
```

Concrete Reusable



```
//concrete reusable
public class Bitmap implements Image {

    private Point2D location;

    private String name;

    public Bitmap(String name) {□

    public void draw() {□

    public Point2D getLocation() {□

    public void setLocation(Point2D location) {□
    //Reset method
    @Override
    public void reset() {
        location = null;
        System.out.println("Bitmap is reset");
    }
}
```

In-A-Hurry Summary

Object Pool

```java
//The object pool, here we are pre-creating all objects.
public class ObjectPool<T extends Poolable> {

    private BlockingQueue<T> availablePool;

    public ObjectPool(Supplier<T> creator, int preCache) {
        availablePool = new LinkedBlockingQueue<>();
        IntStream.range(0, preCache).forEach(i->availablePool.offer(creator.get()));
    }

    //get method must decide what to do if pool is empty. It can create new
    //object and return that or wait until one becomes available
    public T get() {
        try {
            //We are going to wait if none free. NOTE this has severe -ve impact!
            return availablePool.take();
        } catch (InterruptedException e) {
            System.err.println("take() interrupted waiting on pooled queue");
        }
        return null;
    }

    public void release(T obj) {
        obj.reset();
        try {
            availablePool.put(obj);
        } catch (InterruptedException e) {
            System.err.println("put() interrupted waiting on pooled queue");
        }
    }
```

Client

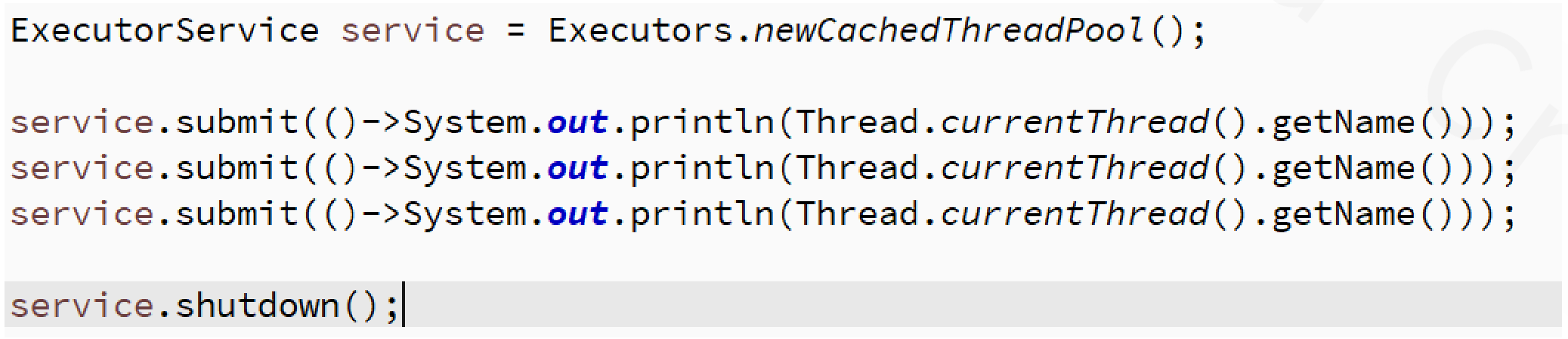
```java
    //create the pool with 5 objects
    public static final ObjectPool<Bitmap> bitmapPool = new ObjectPool<>(()->new Bitmap(

    public static void main(String[] args) {
        //get objects from pool and use them as regular objects
        Bitmap b1 = bitmapPool.get();
        b1.setLocation(new Point2D(10,10));
        Bitmap b2 = bitmapPool.get();
        b2.setLocation(new Point2D(20,10));
        b1.draw();
        b2.draw();

        //release objects when done
        bitmapPool.release(b1);
        bitmapPool.release(b2);
    }
```

Examples of object pool

- Using object pool for saving the memory allocation & GC cost is *almost* deprecated now. JVMs & hardware are more efficient & have access to more memory now.

- However it is still a very common pattern when we are interacting with external resources like threads, connections.

- java.util.concurrent.ThreadPoolExecutor is an example of object pool pattern which pools threads. Even though we can directly use this class, you'll often use it via ExecutorService interface using method like Executors like newCachedThreadPool().



```
ExecutorService service = Executors.newCachedThreadPool();

service.submit(()->System.out.println(Thread.currentThread().getName()));
service.submit(()->System.out.println(Thread.currentThread().getName()));
service.submit(()->System.out.println(Thread.currentThread().getName()));

service.shutdown();
```

Examples of object pool

- Apache commons dbcp library is used for database connection pooling. Class org.apache.commons.dbcp.BasicDataSource in dbcp package is an example of object pool pattern which pools database connections. This pool is commonly created and exposed via JNDI or as a Spring bean in applications.

```
// Construct BasicDataSource
//typically is bound to JNDI or set as spring bean
BasicDataSource dataSource = new BasicDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql://localhost/brooklyn_99_topsecret");
dataSource.setUsername("scully");
dataSource.setPassword("hitchcock");

//Then @ runtime
Connection conn = dataSource.getConnection();
//Use connection, and then close it to return it to pool.
conn.close();

//At application shutdown, close the pool
dataSource.close();
```

Adapter (aka Wrapper)

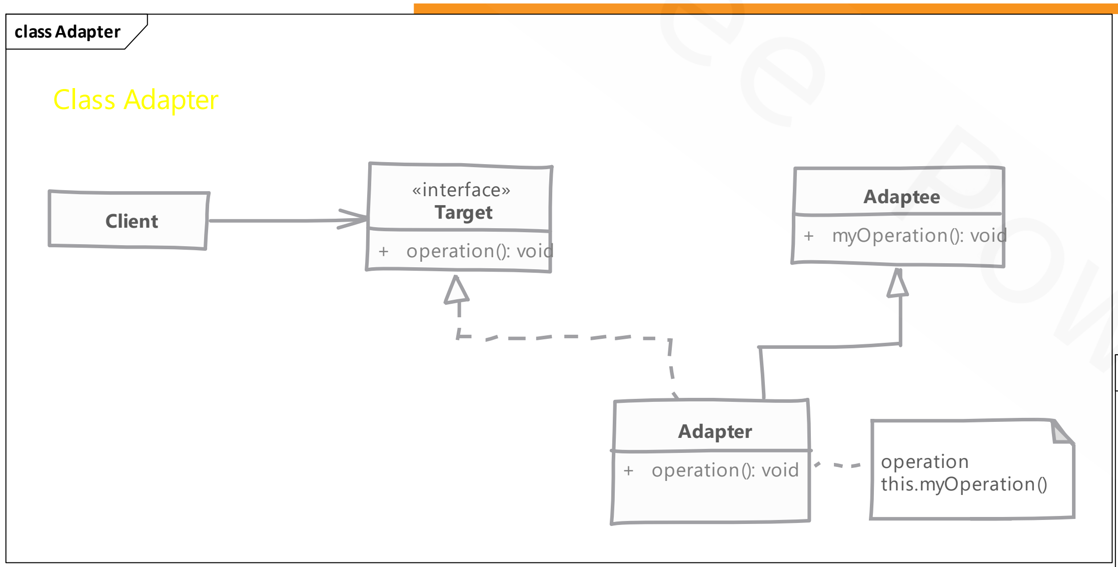
Structural Design Patterns

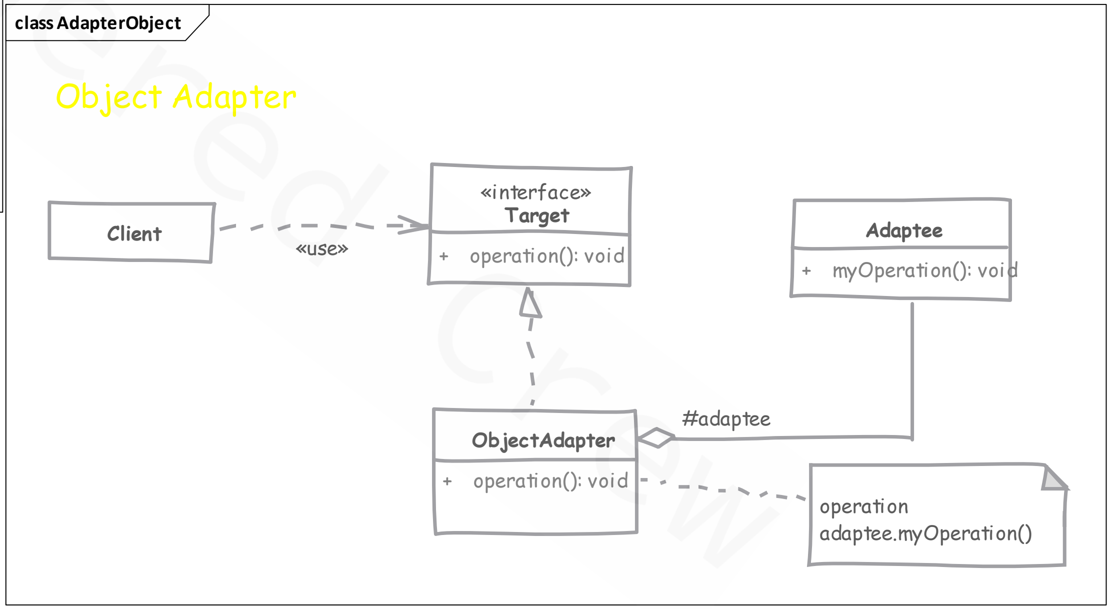
Design Patterns in Java

In-A-Hurry Summary

- We have an existing object with required functionality but the client code is expecting a different interface than our object.
- A class adapter is one where adapter inherits from class of object which is to be adapted and implements the interface required by client code. This adapter type should be avoided.
- An object adapter uses composition over inheritance. It'll implement the target interface and use an adaptee object composition to perform translation. This allows us to use subclasses of adaptee in adapter.

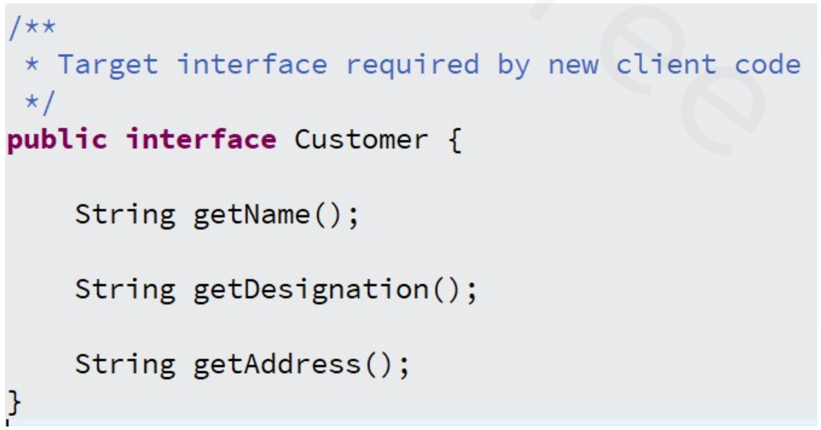
In-A-Hurry Summary





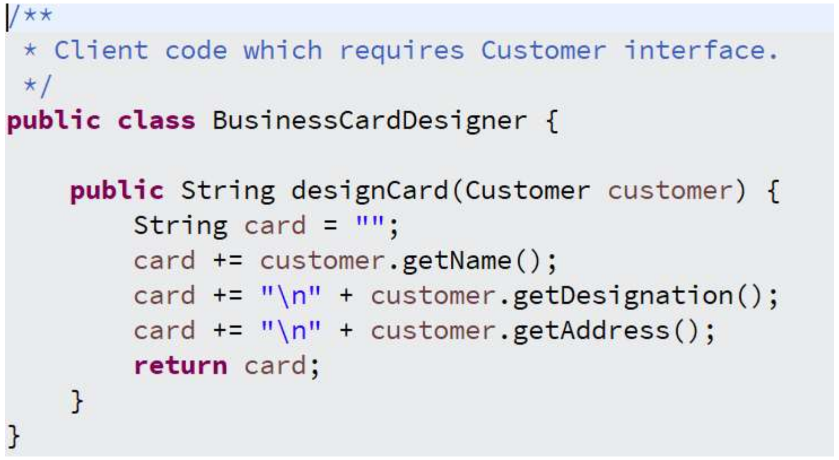
In-A-Hurry Summary

Target Interface



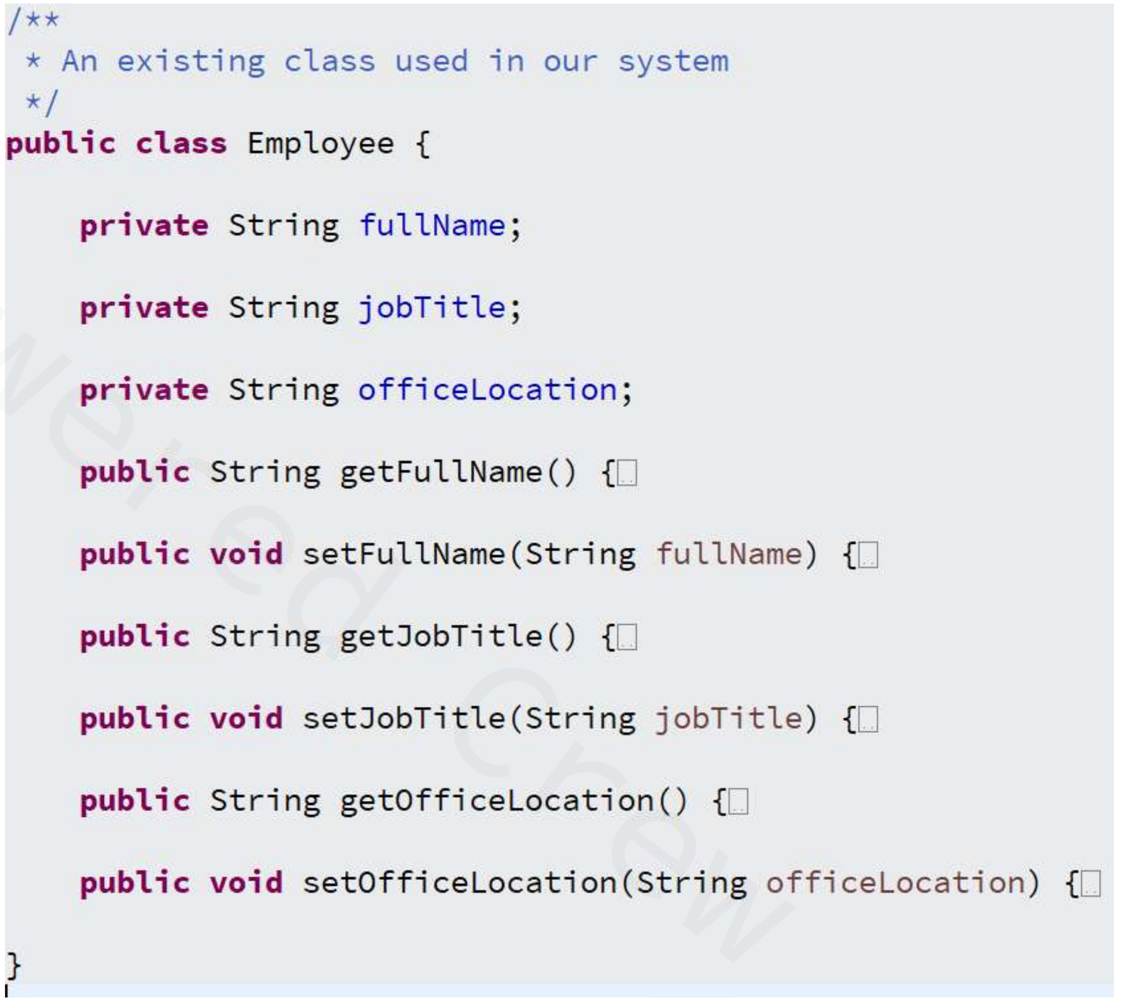
```
/**
 * Target interface required by new client code
 */
public interface Customer {

    String getName();

    String getDesignation();

    String getAddress();
}
```

Client Code expecting the target interface



```
/**
 * Client code which requires Customer interface.
 */
public class BusinessCardDesigner {

    public String designCard(Customer customer) {
        String card = "";
        card += customer.getName();
        card += "\n" + customer.getDesignation();
        card += "\n" + customer.getAddress();
        return card;
    }
}
```

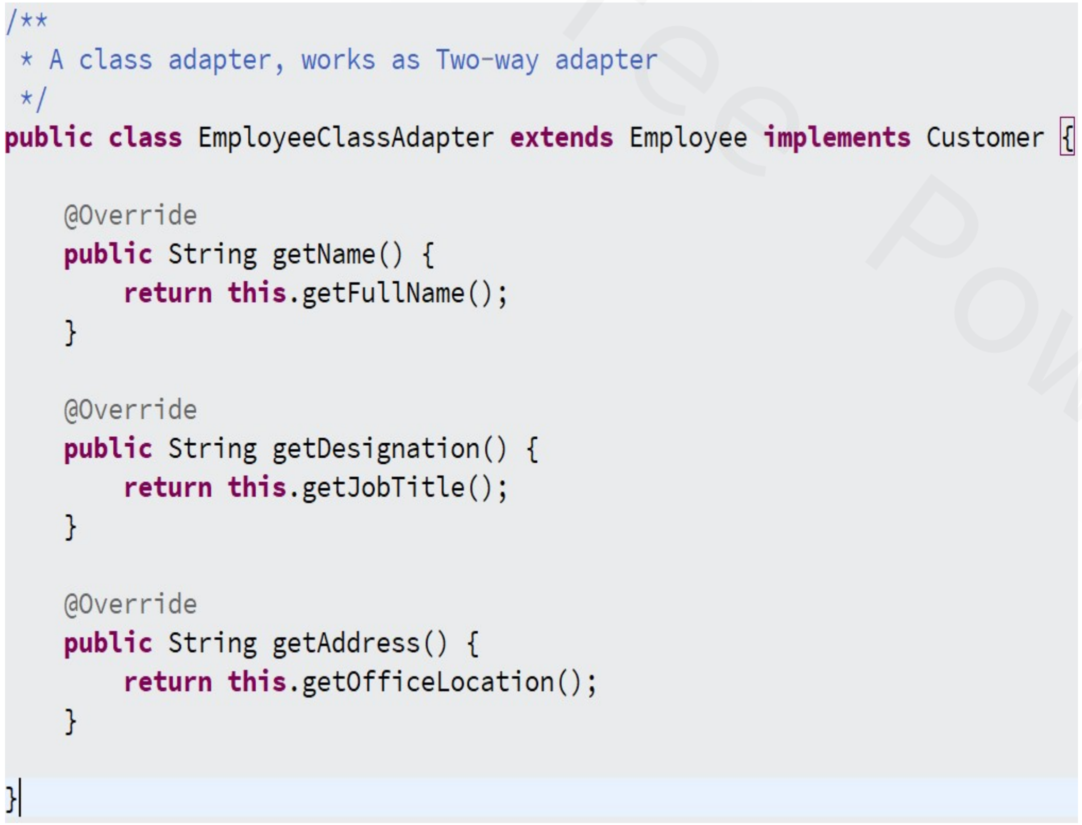
Our existing class (Adaptee)



```
/**
 * An existing class used in our system
 */
public class Employee {

    private String fullName;

    private String jobTitle;

    private String officeLocation;

    public String getFullName() {□

    public void setFullName(String fullName) {□

    public String getJobTitle() {□

    public void setJobTitle(String jobTitle) {□

    public String getOfficeLocation() {□

    public void setOfficeLocation(String officeLocation) {□

}
```

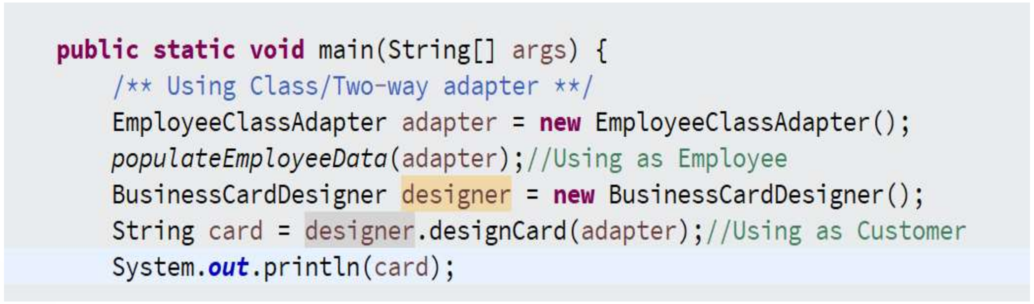
In-A-Hurry Summary

Class Adapter



```java
/**
 * A class adapter, works as Two-way adapter
 */
public class EmployeeClassAdapter extends Employee implements Customer {

    @Override
    public String getName() {
        return this.getFullName();
    }

    @Override
    public String getDesignation() {
        return this.getJobTitle();
    }

    @Override
    public String getAddress() {
        return this.getOfficeLocation();
    }

}
```

Using Class Adapter

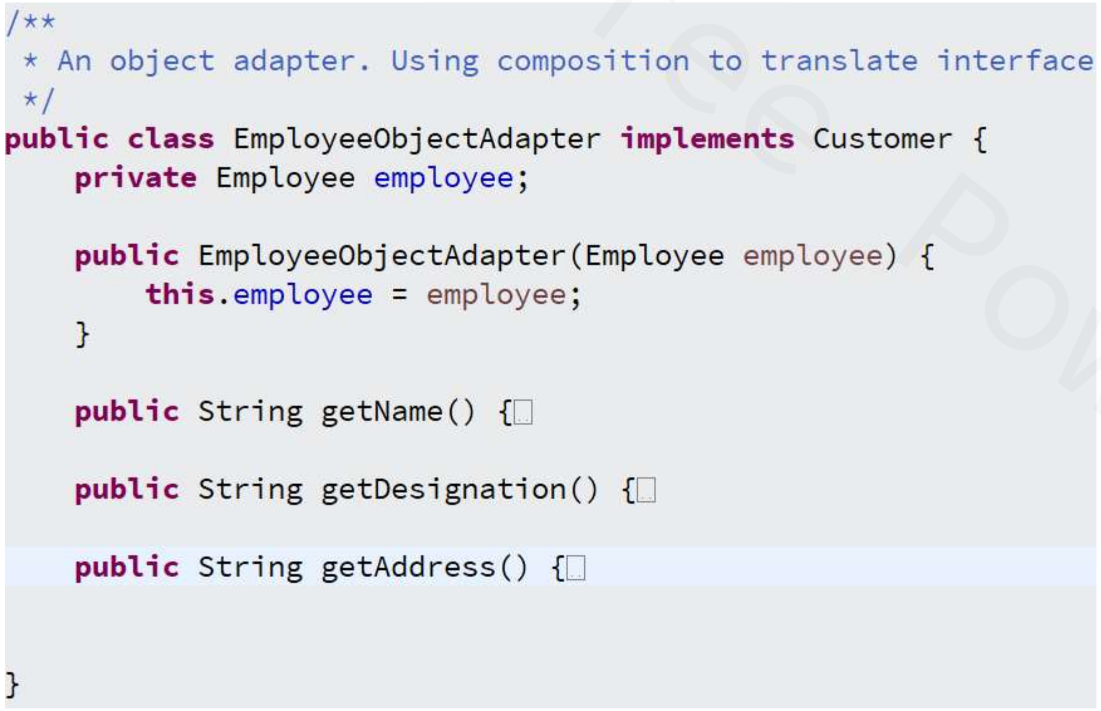


```java
public static void main(String[] args) {
    /** Using Class/Two-way adapter **/
    EmployeeClassAdapter adapter = new EmployeeClassAdapter();
    populateEmployeeData(adapter);//Using as Employee
    BusinessCardDesigner designer = new BusinessCardDesigner();
    String card = designer.designCard(adapter);//Using as Customer
    System.out.println(card);
```

- Here we end up with an object that has publicly available methods from both Adaptee and Target interface.
- Avoid class adapters!

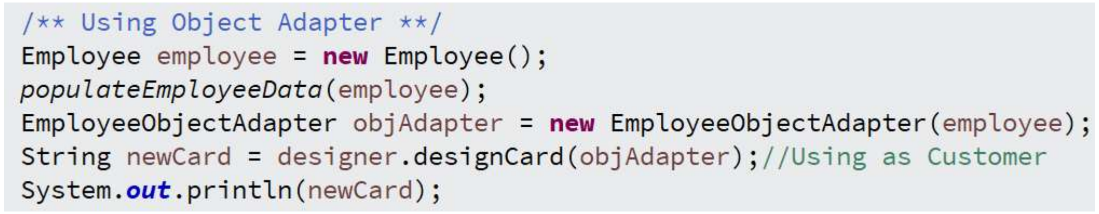
In-A-Hurry Summary

Object Adapter



```
/**
 * An object adapter. Using composition to translate interface
 */
public class EmployeeObjectAdapter implements Customer {
    private Employee employee;

    public EmployeeObjectAdapter(Employee employee) {
        this.employee = employee;
    }

    public String getName() {□

    public String getDesignation() {□

    public String getAddress() {□


}
```

Using Object Adapter

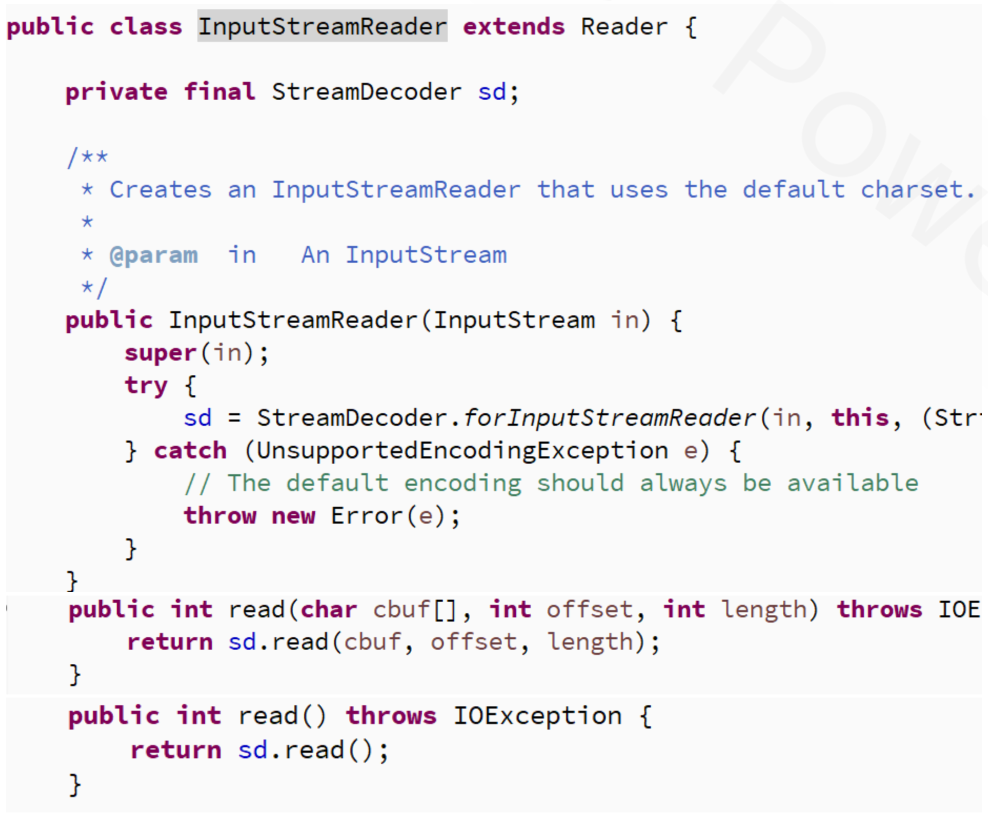


```
/** Using Object Adapter **/
Employee employee = new Employee();
populateEmployeeData(employee);
EmployeeObjectAdapter objAdapter = new EmployeeObjectAdapter(employee);
String newCard = designer.designCard(objAdapter);//Using as Customer
System.out.println(newCard);
```

Example of an Adapter

- The java.io.InputStreamReader and java.io.OutputStreamWriter classes are examples of object adapters.

- These classes adapt existing InputStream/OutputStream object to a Reader/Writer interface.



```
public class InputStreamReader extends Reader {

    private final StreamDecoder sd;

    /**
     * Creates an InputStreamReader that uses the default charset.
     *
     * @param  in   An InputStream
     */
    public InputStreamReader(InputStream in) {
        super(in);
        try {
            sd = StreamDecoder.forInputStreamReader(in, this, (Str
        } catch (UnsupportedEncodingException e) {
            // The default encoding should always be available
            throw new Error(e);
        }
    }
    public int read(char cbuf[], int offset, int length) throws IOE
        return sd.read(cbuf, offset, length);
    }
    public int read() throws IOException {
        return sd.read();
    }
```

Bridge

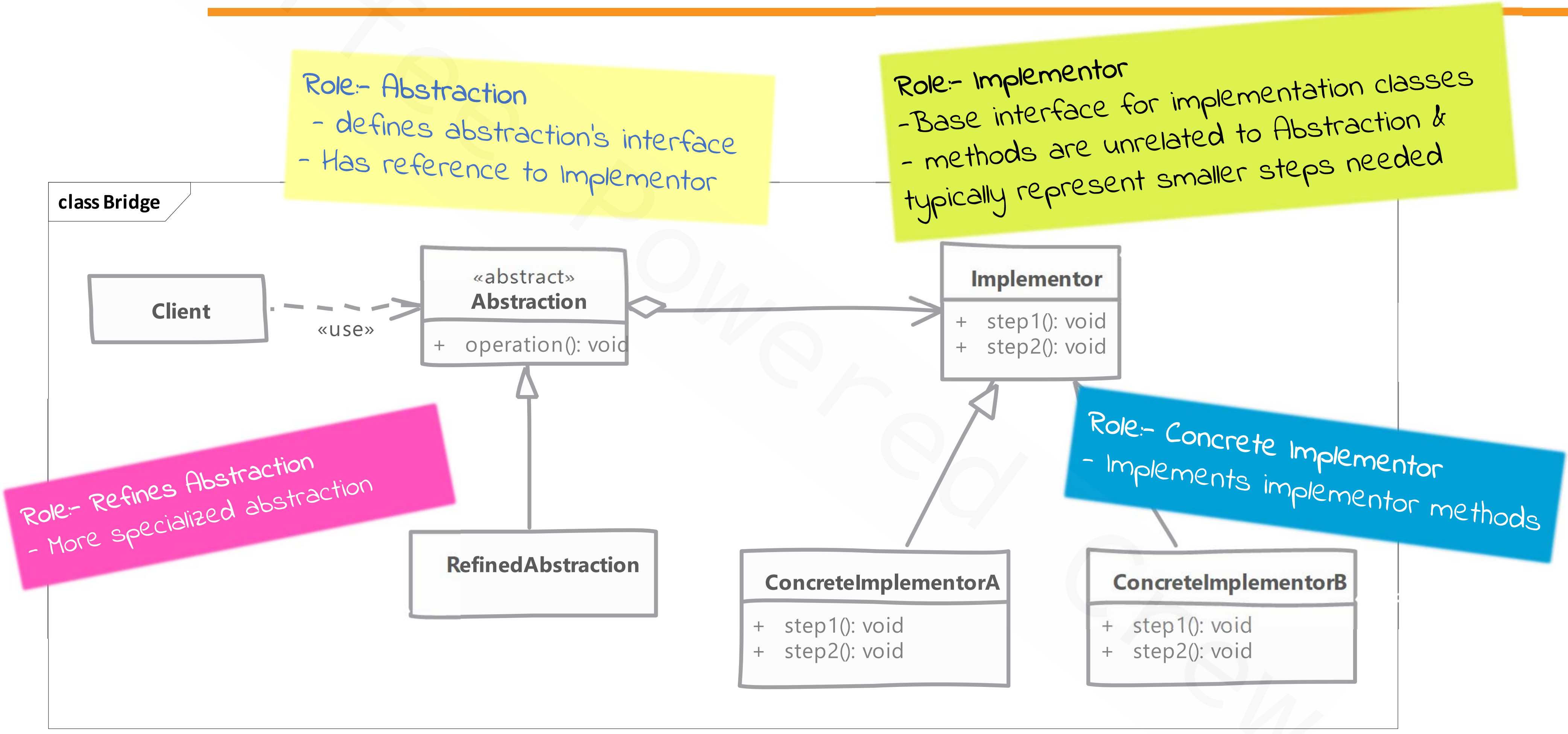
Structural Design Patterns

Design Patterns in Java

In-A-Hurry Summary

- We use bridge pattern when we want our abstractions and implementations to be decoupled.
- Bridge pattern defines separate inheritance hierarchies for abstraction & implementations and bridge these two together using composition.
- Implementations do not HAVE to define methods that match up with methods in abstraction. It is fairly common to have primitive methods; methods which do small work; in implementor. Abstraction uses these methods to provide its functionality.

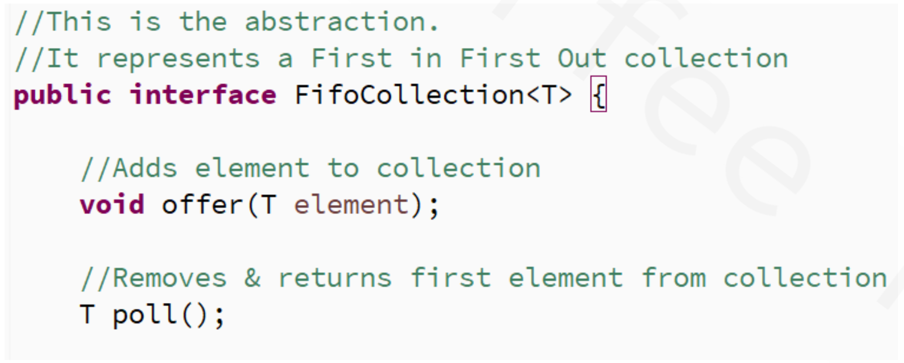
In-A-Hurry Summary



class Bridge
Client
«use»
«abstract»
Abstraction
+ operation(): void
Implementor
+ step1(): void
+ step2(): void
RefinedAbstraction
ConcreteImplementorA
+ step1(): void
+ step2(): void
ConcreteImplementorB
+ step1(): void
+ step2(): void
Role:- Abstraction
- defines abstraction's interface
- Has reference to Implementor
Role:- Implementor
-Base interface for implementation classes
- methods are unrelated to Abstraction & typically represent smaller steps needed
Role:- Refines Abstraction
- More specialized abstraction
Role:- Concrete Implementor
- Implements implementor methods

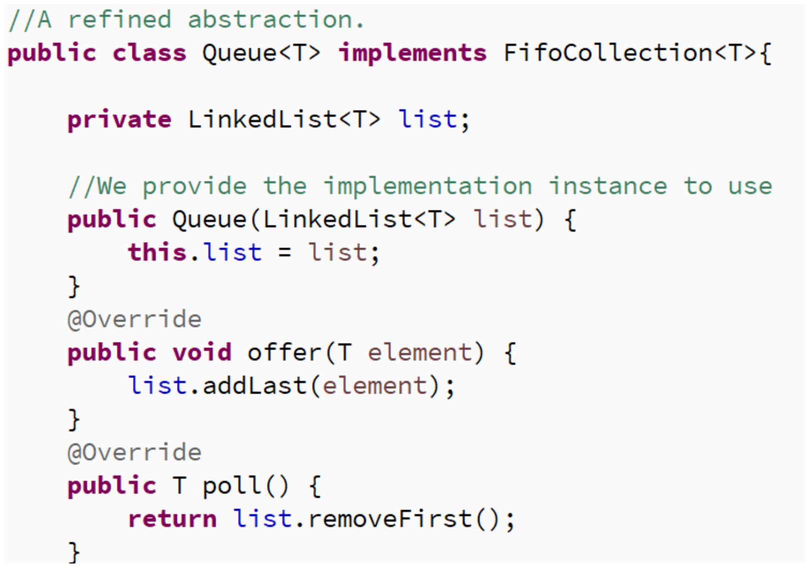
In-A-Hurry Summary

Abstraction



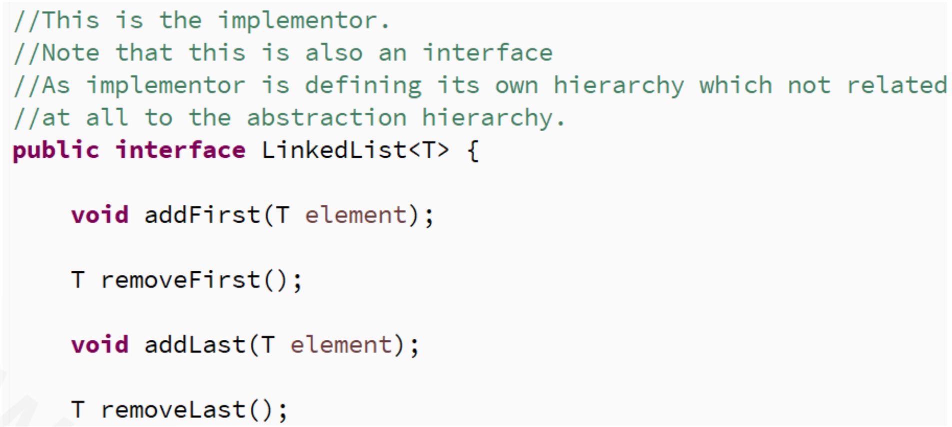
```java
//This is the abstraction.
//It represents a First in First Out collection
public interface FifoCollection<T> {

    //Adds element to collection
    void offer(T element);

    //Removes & returns first element from collection
    T poll();
```

Refined Abstraction



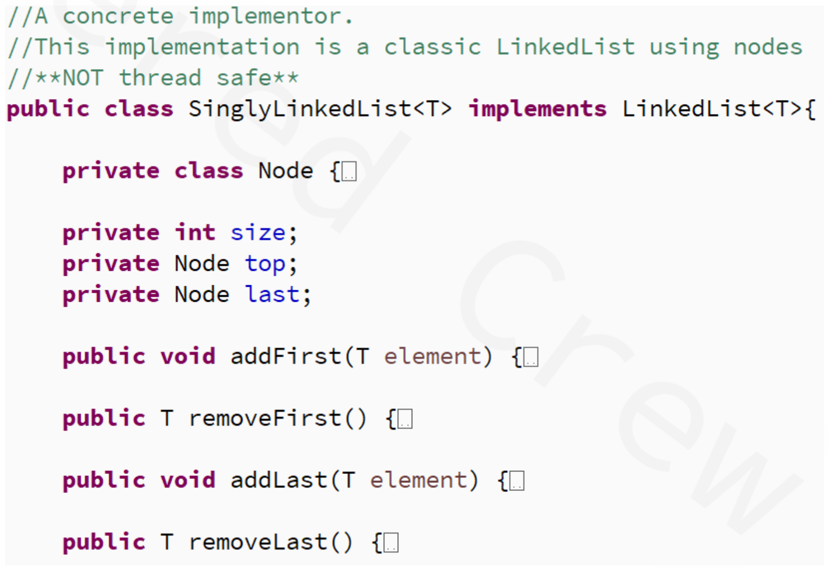
```java
//A refined abstraction.
public class Queue<T> implements FifoCollection<T>{

    private LinkedList<T> list;

    //We provide the implementation instance to use
    public Queue(LinkedList<T> list) {
        this.list = list;
    }
    @Override
    public void offer(T element) {
        list.addLast(element);
    }
    @Override
    public T poll() {
        return list.removeFirst();
    }
```

Implementor



```java
//This is the implementor.
//Note that this is also an interface
//As implementor is defining its own hierarchy which not related
//at all to the abstraction hierarchy.
public interface LinkedList<T> {

    void addFirst(T element);

    T removeFirst();

    void addLast(T element);

    T removeLast();
```

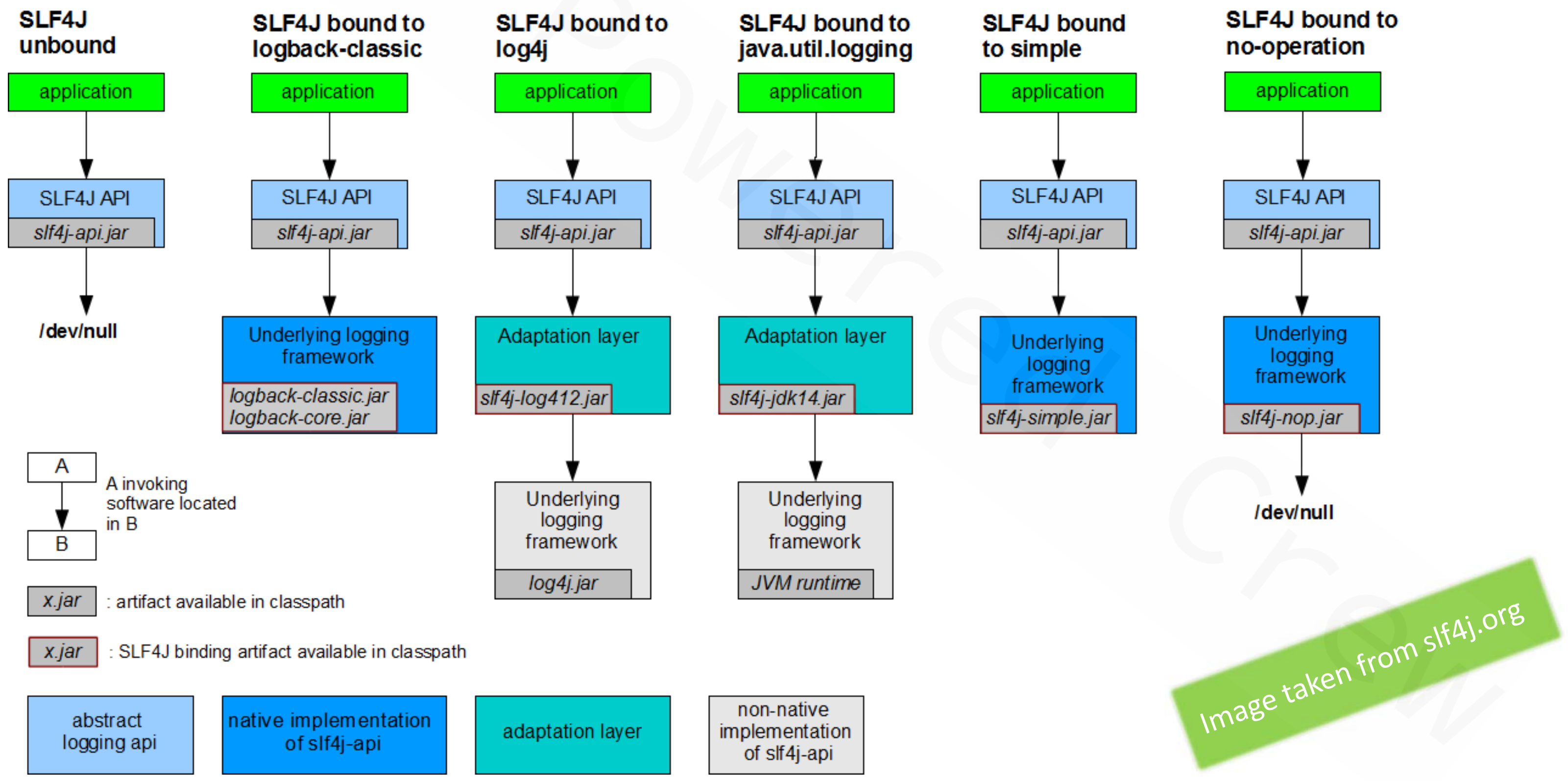
Concrete Implementor



```java
//A concrete implementor.
//This implementation is a classic LinkedList using nodes
//**NOT thread safe**
public class SinglyLinkedList<T> implements LinkedList<T>{

    private class Node {□

    private int size;
    private Node top;
    private Node last;

    public void addFirst(T element) {□

    public T removeFirst() {□

    public void addLast(T element) {□

    public T removeLast() {□
```

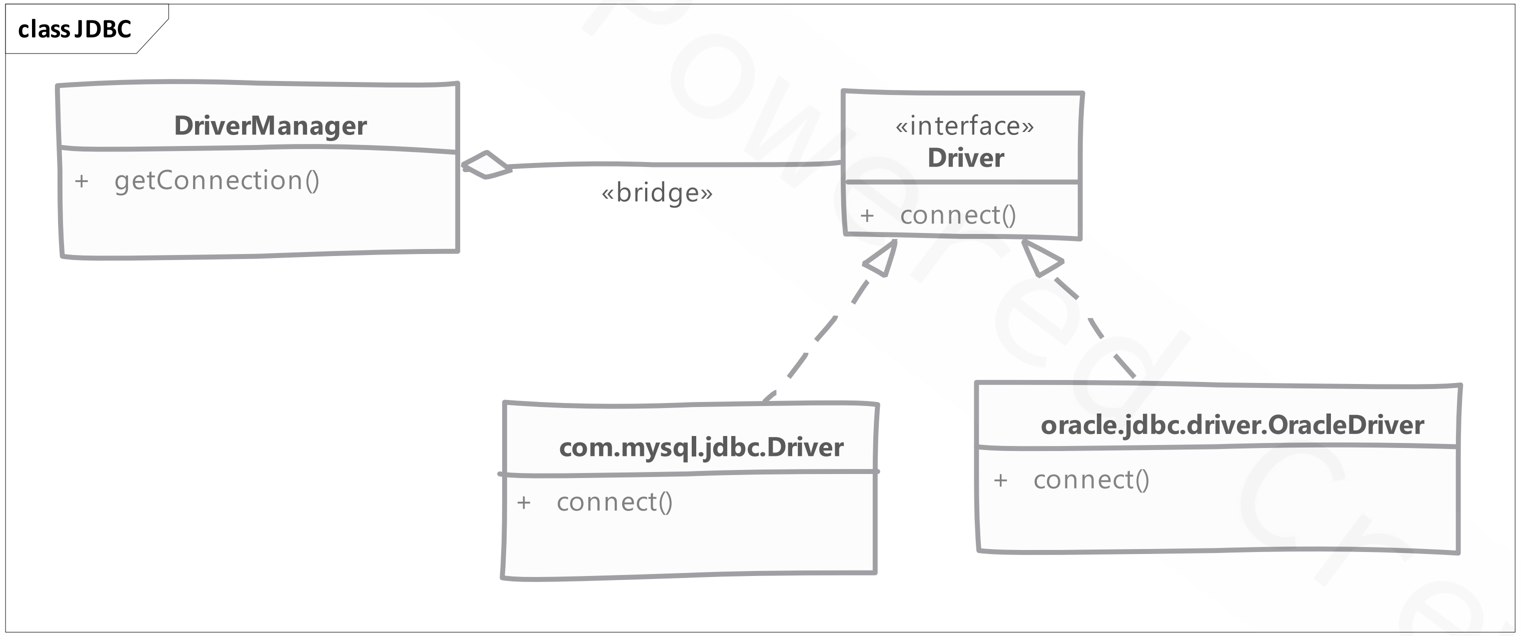
Example of a Bridge

- An example of bridge pattern is the SLF4J logging framework. Using slf4j we can switch the actual logging framework being used by dropping jar files in classpath. The client code using slf4j API remains unaffected. So the abstraction (slf4j API) can vary independent of implementation (log4j, logback etc)


SLF4J unbound
SLF4J bound to logback-classic
SLF4J bound to log4j
SLF4J bound to java.util.logging
SLF4J bound to simple
SLF4J bound to no-operation
application
SLF4J API
slf4j-api.jar
/dev/null
Underlying logging framework
logback-classic.jar
logback-core.jar
Adaptation layer
slf4j-log412.jar
log4j.jar
slf4j-jdk14.jar
JVM runtime
slf4j-simple.jar
slf4j-nop.jar
A
B
A invoking software located in B
x.jar : artifact available in classpath
x.jar : SLF4J binding artifact available in classpath
abstract logging api
native implementation of slf4j-api
adaptation layer
non-native implementation of slf4j-api
Image taken from slf4j.org

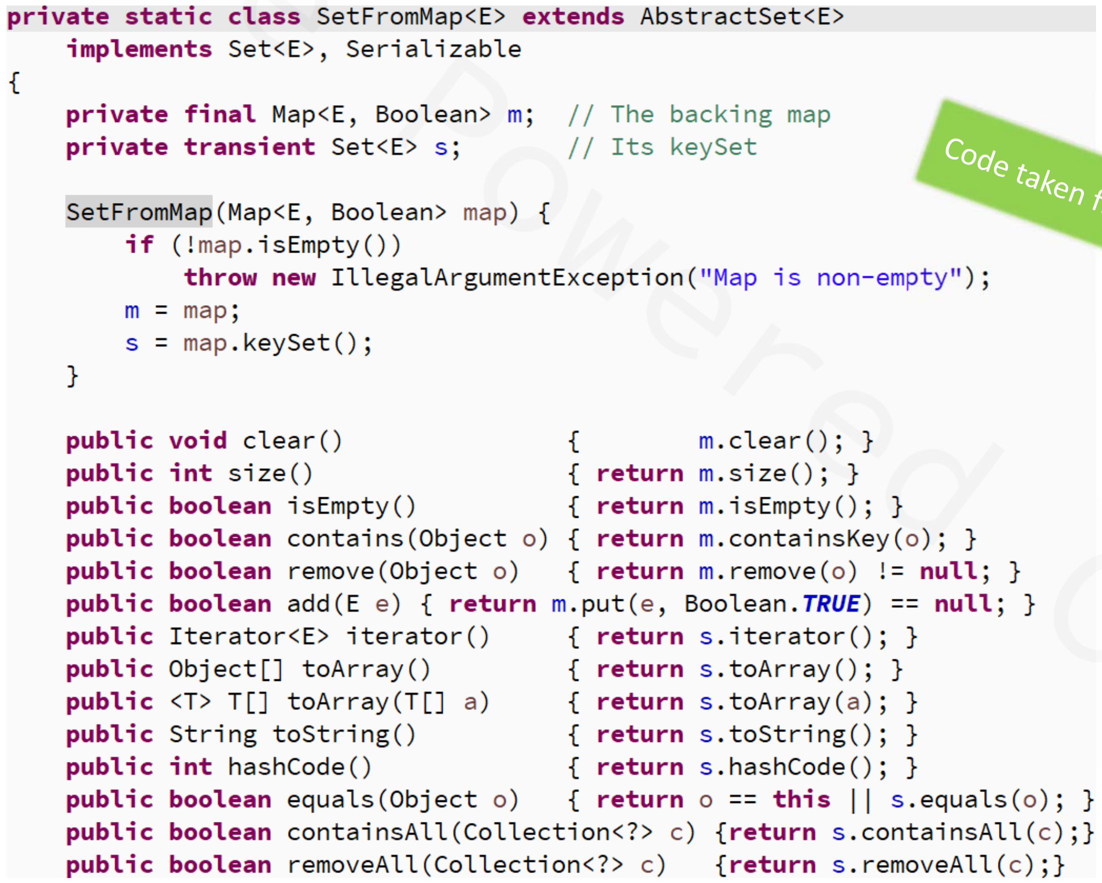
Example of a Bridge

- An example of bridge pattern often given is the JDBC API. More specifically the java.sql.DriverManager class with the java.sql.Driver interface form a bridge pattern.



Example of a Bridge

- An example of bridge pattern often given is the Collections.newSetFromMap() method. This method returns a Set which is backed by given map object.



```java
private static class SetFromMap<E> extends AbstractSet<E>
    implements Set<E>, Serializable
{
    private final Map<E, Boolean> m;  // The backing map
    private transient Set<E> s;       // Its keySet

    SetFromMap(Map<E, Boolean> map) {
        if (!map.isEmpty())
            throw new IllegalArgumentException("Map is non-empty");
        m = map;
        s = map.keySet();
    }

    public void clear()               {        m.clear(); }
    public int size()                 { return m.size(); }
    public boolean isEmpty()          { return m.isEmpty(); }
    public boolean contains(Object o) { return m.containsKey(o); }
    public boolean remove(Object o)   { return m.remove(o) != null; }
    public boolean add(E e) { return m.put(e, Boolean.TRUE) == null; }
    public Iterator<E> iterator()     { return s.iterator(); }
    public Object[] toArray()         { return s.toArray(); }
    public <T> T[] toArray(T[] a)     { return s.toArray(a); }
    public String toString()          { return s.toString(); }
    public int hashCode()             { return s.hashCode(); }
    public boolean equals(Object o)   { return o == this || s.equals(o); }
    public boolean containsAll(Collection<?> c) {return s.containsAll(c);}
    public boolean removeAll(Collection<?> c)   {return s.removeAll(c);}
```

Code taken from Collections.class

Decorator

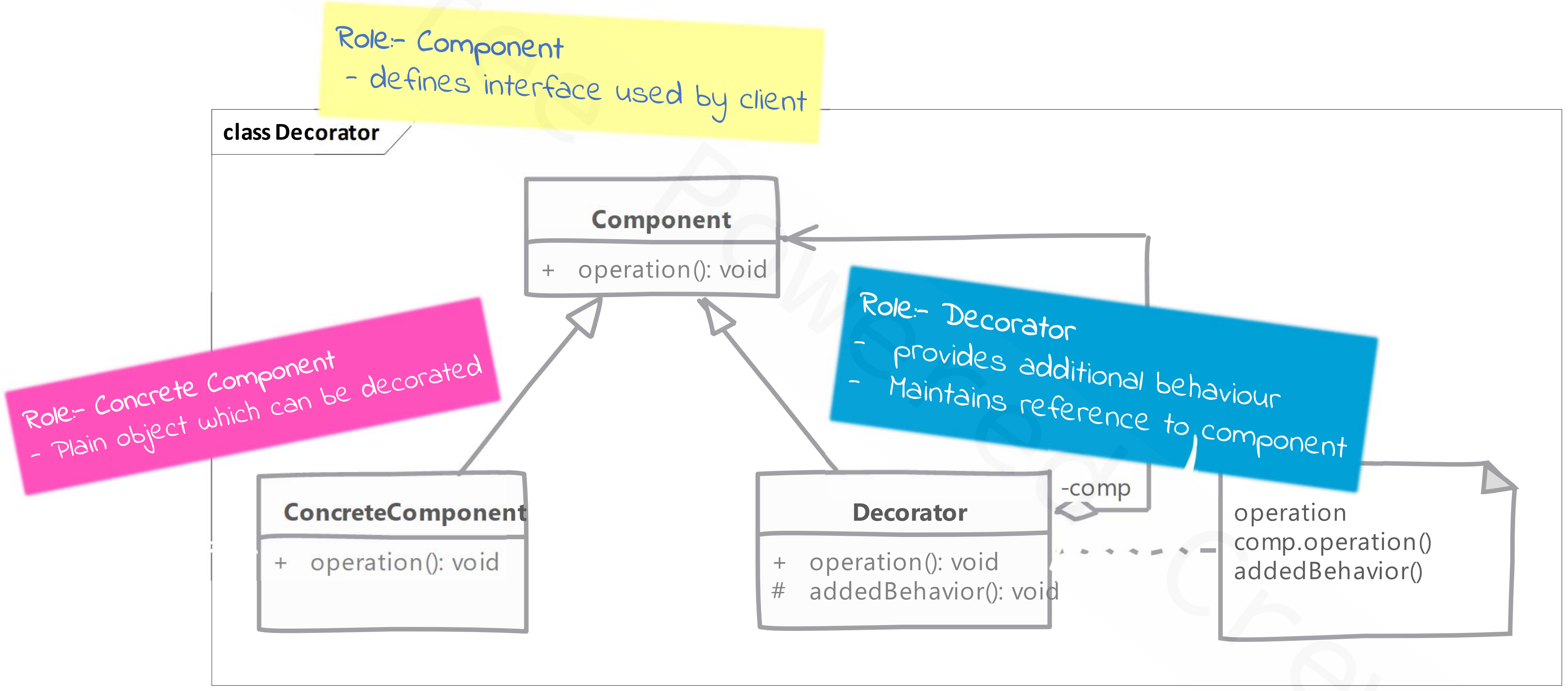
Structural Design Patterns

Design Patterns in Java

In-A-Hurry Summary

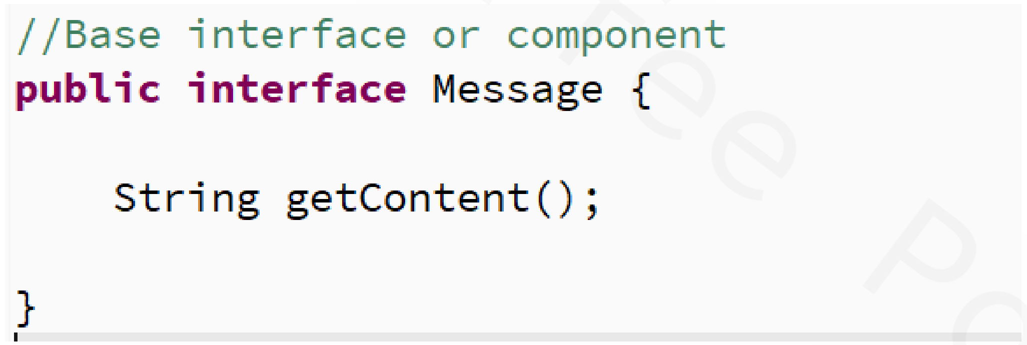
- We use decorator when we want to add small behaviour on top of existing object.
- A decorator has same interface as the object it decorates or contains.
- Decorators allow you to dynamically construct behaviour by using composition. A decorator can wrap another decorator which in turn wraps original object.
- Client of object is unaware of existence of decorator.

In-A-Hurry Summary



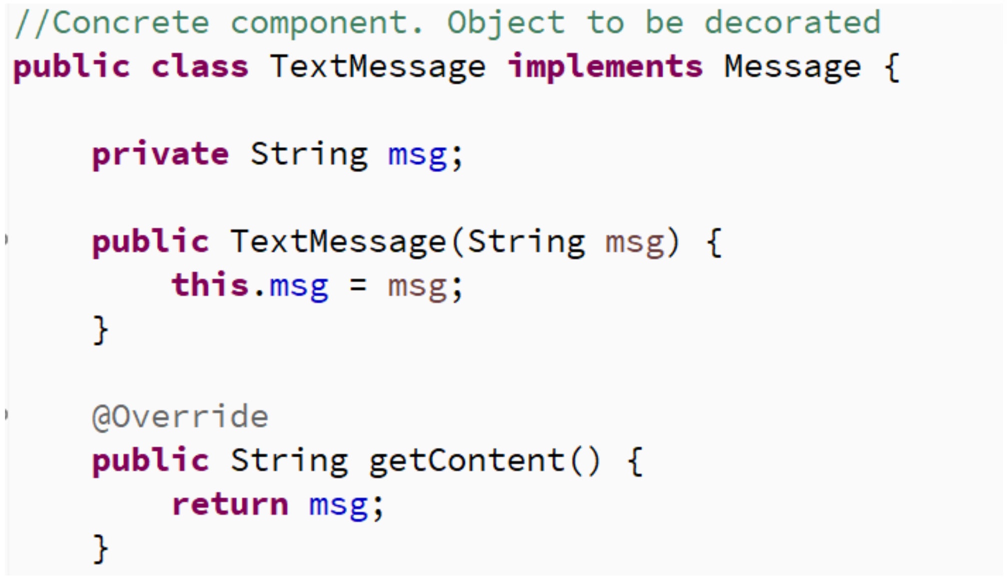
In-A-Hurry Summary

Component



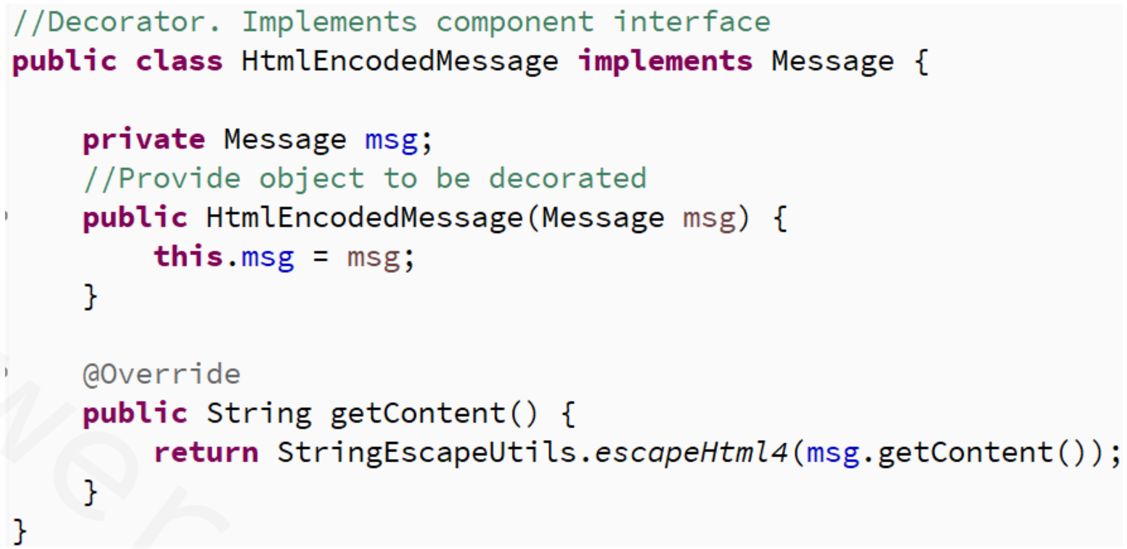
```java
//Base interface or component
public interface Message {

    String getContent();

}
```

Concrete Component



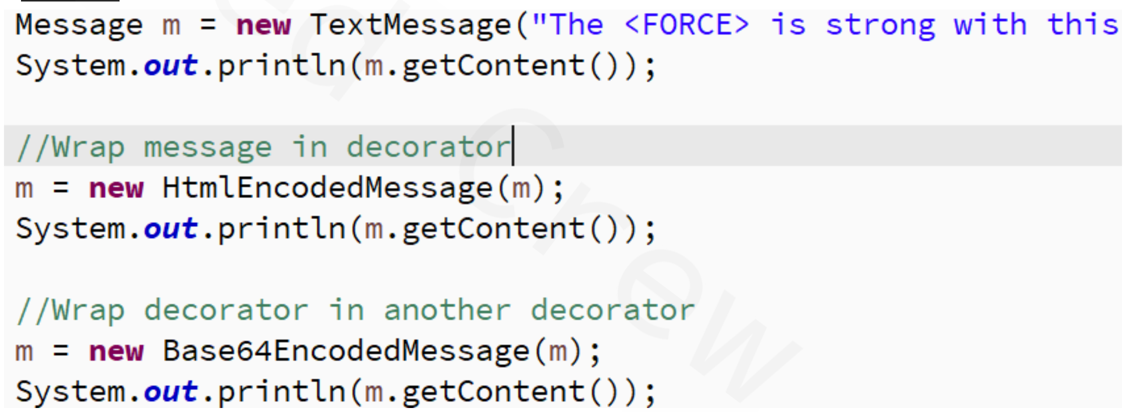
```java
//Concrete component. Object to be decorated
public class TextMessage implements Message {

    private String msg;

    public TextMessage(String msg) {
        this.msg = msg;
    }

    @Override
    public String getContent() {
        return msg;
    }
```

Decorator



```java
//Decorator. Implements component interface
public class HtmlEncodedMessage implements Message {

    private Message msg;
    //Provide object to be decorated
    public HtmlEncodedMessage(Message msg) {
        this.msg = msg;
    }

    @Override
    public String getContent() {
        return StringEscapeUtils.escapeHtml4(msg.getContent());
    }
}
```

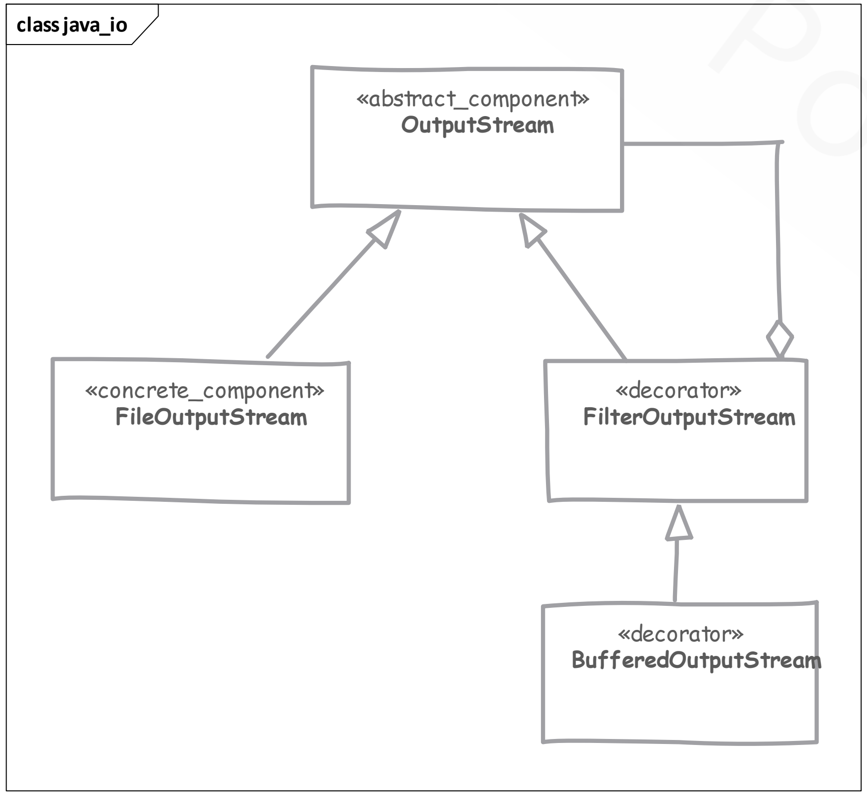
Client



```java
Message m = new TextMessage("The <FORCE> is strong with this
System.out.println(m.getContent());

//Wrap message in decorator
m = new HtmlEncodedMessage(m);
System.out.println(m.getContent());

//Wrap decorator in another decorator
m = new Base64EncodedMessage(m);
System.out.println(m.getContent());
```

Example of a Decorator

- Classes in Java's I/O package are great examples of decorator pattern.
- For example the java.io.BufferedOutputStream class decorates any java.io.OutputStream object and adds buffering to file writing operation. This improves the disk i/o performance by reducing number of writes.



class java_io
«abstract_component» OutputStream
«concrete_component» FileOutputStream
«decorator» FilterOutputStream
«decorator» BufferedOutputStream

```java
try (OutputStream os = new BufferedOutputStream(
        new FileOutputStream("xfiles_mulder_notes.txt")
        )){
    os.write('x');
    os.flush();
}
```

Composite

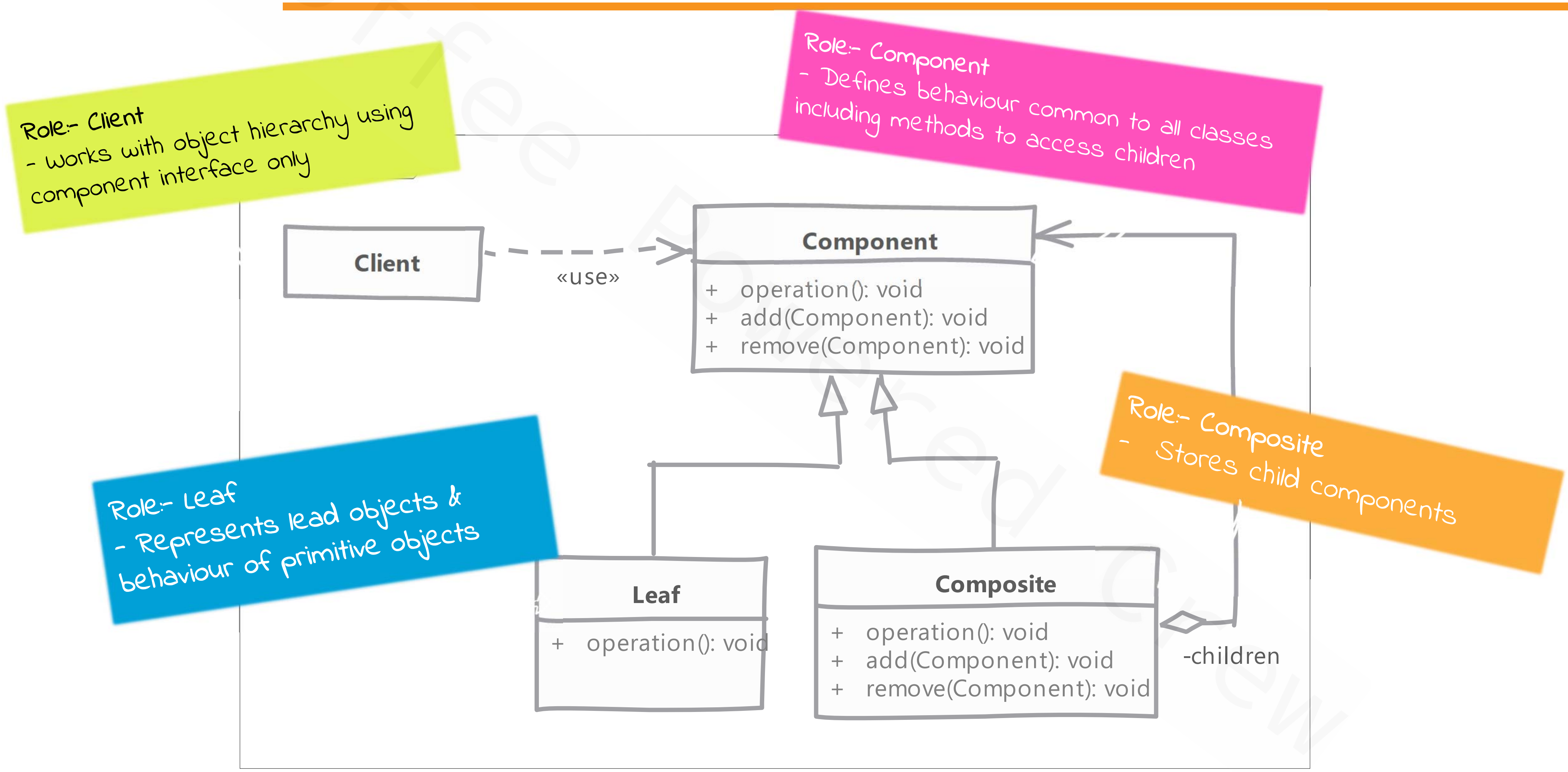
Structural Design Patterns

Design Patterns in Java

In-A-Hurry Summary

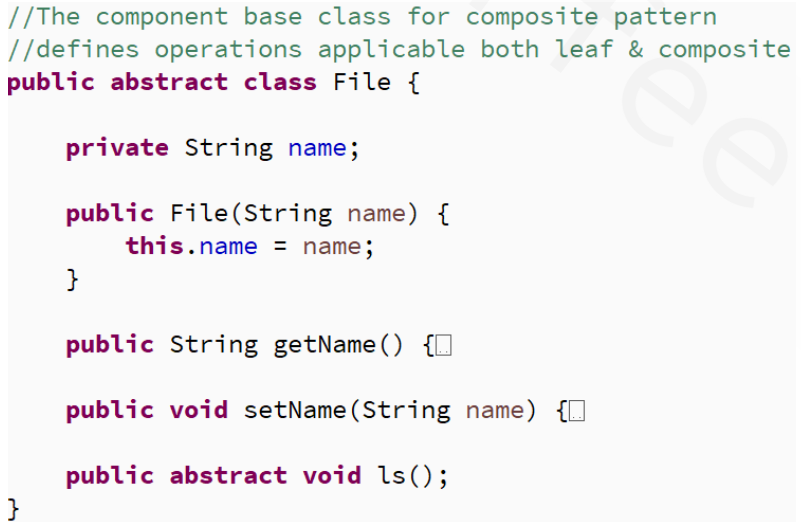
- We have a parent-child or whole-part relation between objects. We can use composite pattern to simplify dealing with such object arrangements.
- Goal of composite pattern is to simplify the client code by allowing it to treat the composites and leaf nodes in same way.
- Composites will delegate the operations to its children while leaf nodes implement the functionality.
- You have to decide which methods the base component will define. Adding all methods here will allow client to treat all nodes same. But it may force classes to implement behaviour which they don’t have.

In-A-Hurry Summary


Role:- Client
- Works with object hierarchy using component interface only
Role:- Component
- Defines behaviour common to all classes including methods to access children
Client
«use»
Component
+ operation(): void
+ add(Component): void
+ remove(Component): void
Role:- Composite
- Stores child components
Role:- Leaf
- Represents lead objects & behaviour of primitive objects
Leaf
+ operation(): void
Composite
+ operation(): void
+ add(Component): void
+ remove(Component): void
-children

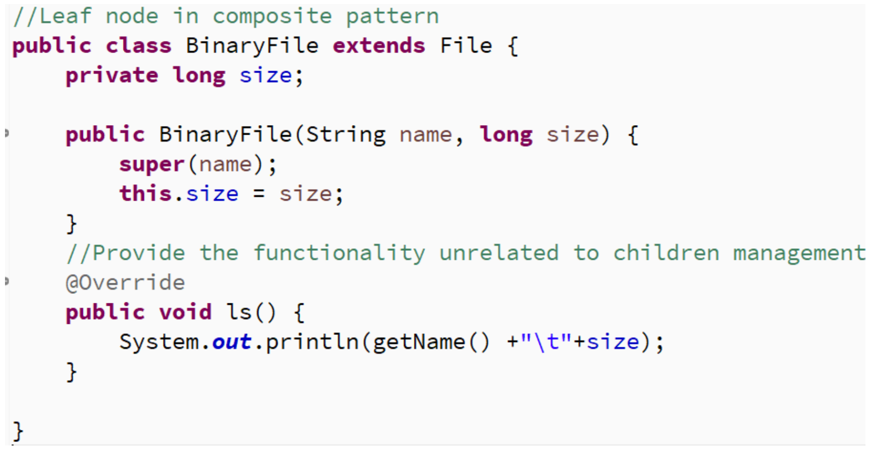
In-A-Hurry Summary

Component



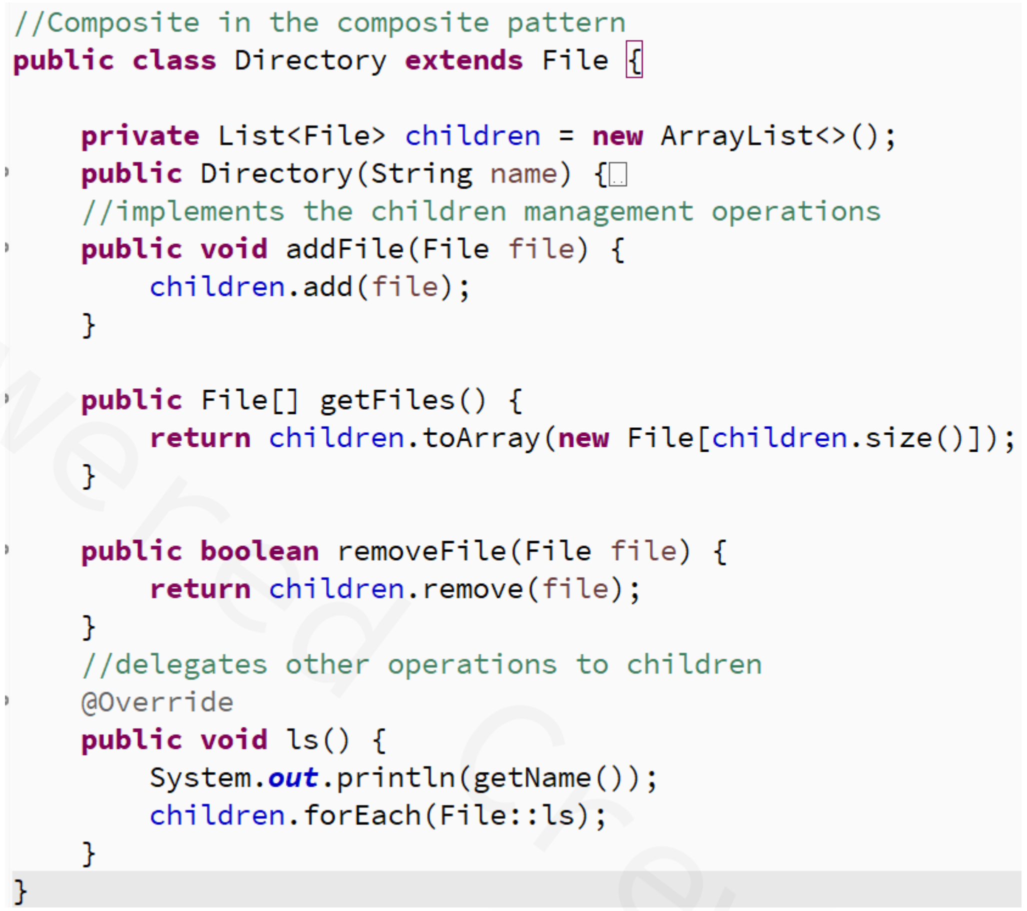
```java
//The component base class for composite pattern
//defines operations applicable both leaf & composite
public abstract class File {

    private String name;

    public File(String name) {
        this.name = name;
    }

    public String getName() {□

    public void setName(String name) {□

    public abstract void ls();
}
```

Leaf Node



```java
//Leaf node in composite pattern
public class BinaryFile extends File {
    private long size;

    public BinaryFile(String name, long size) {
        super(name);
        this.size = size;
    }
    //Provide the functionality unrelated to children management
    @Override
    public void ls() {
        System.out.println(getName() +"\t"+size);
    }

}
```

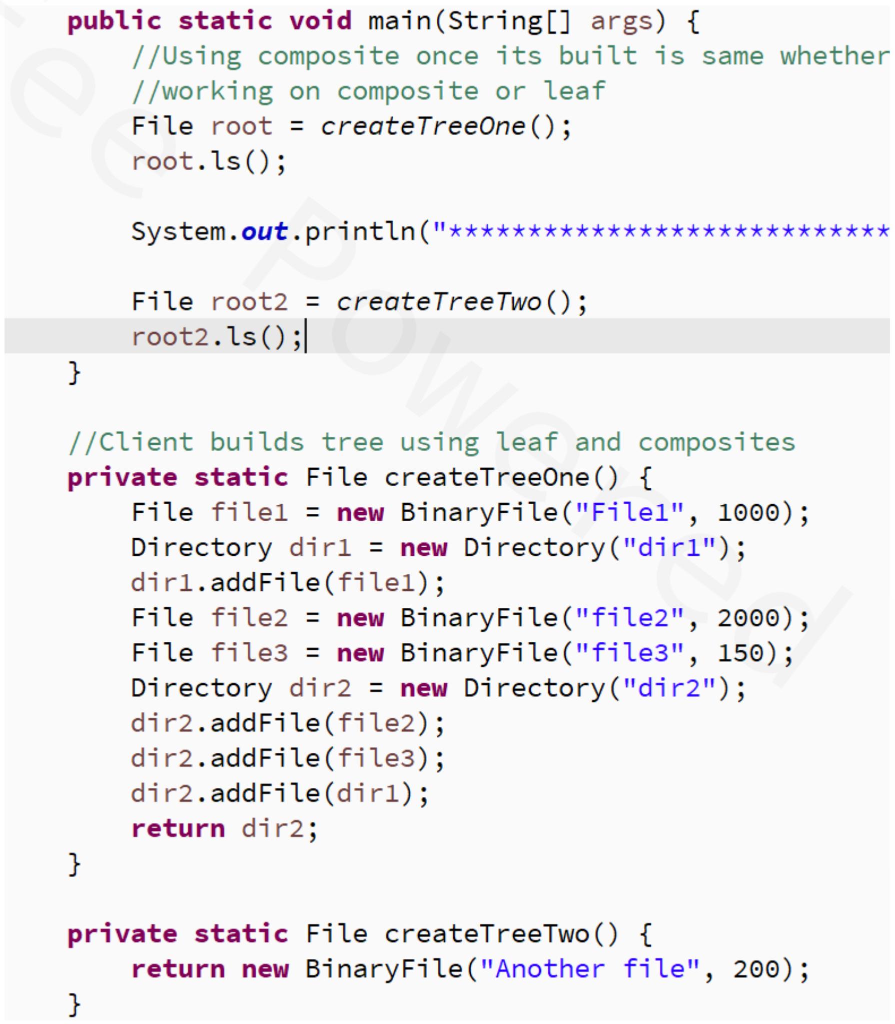
Composite



```java
//Composite in the composite pattern
public class Directory extends File {

    private List<File> children = new ArrayList<>();
    public Directory(String name) {□
    //implements the children management operations
    public void addFile(File file) {
        children.add(file);
    }

    public File[] getFiles() {
        return children.toArray(new File[children.size()]);
    }

    public boolean removeFile(File file) {
        return children.remove(file);
    }
    //delegates other operations to children
    @Override
    public void ls() {
        System.out.println(getName());
        children.forEach(File::ls);
    }
}
```

In-A-Hurry Summary

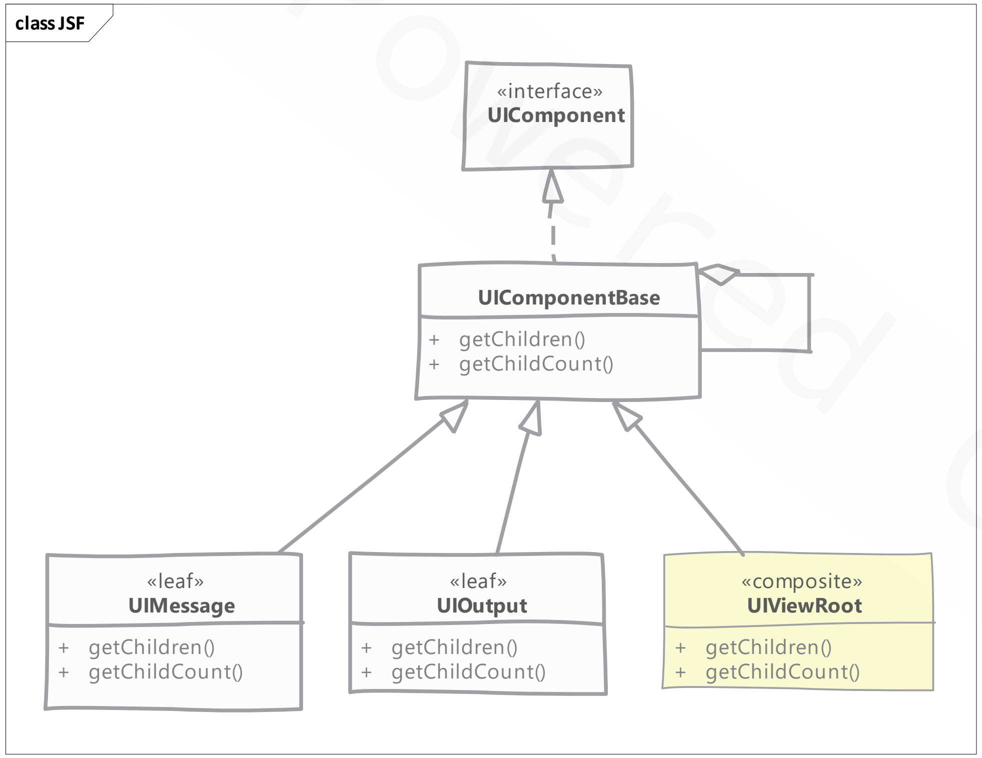
Client of composite pattern



```
public static void main(String[] args) {
    //Using composite once its built is same whether
    //working on composite or leaf
    File root = createTreeOne();
    root.ls();

    System.out.println("****************************

    File root2 = createTreeTwo();
    root2.ls();
}

//Client builds tree using leaf and composites
private static File createTreeOne() {
    File file1 = new BinaryFile("File1", 1000);
    Directory dir1 = new Directory("dir1");
    dir1.addFile(file1);
    File file2 = new BinaryFile("file2", 2000);
    File file3 = new BinaryFile("file3", 150);
    Directory dir2 = new Directory("dir2");
    dir2.addFile(file2);
    dir2.addFile(file3);
    dir2.addFile(dir1);
    return dir2;
}

private static File createTreeTwo() {
    return new BinaryFile("Another file", 200);
}
```

Example of a Composite

- Composite is used in many UI frameworks, since it easily allows to represent a tree of UI controls.
- In JSF we have UIViewRoot class which acts as composite. Other UIComponent implementations like UIOutput, UIMessage act as leaf nodes.



Facade

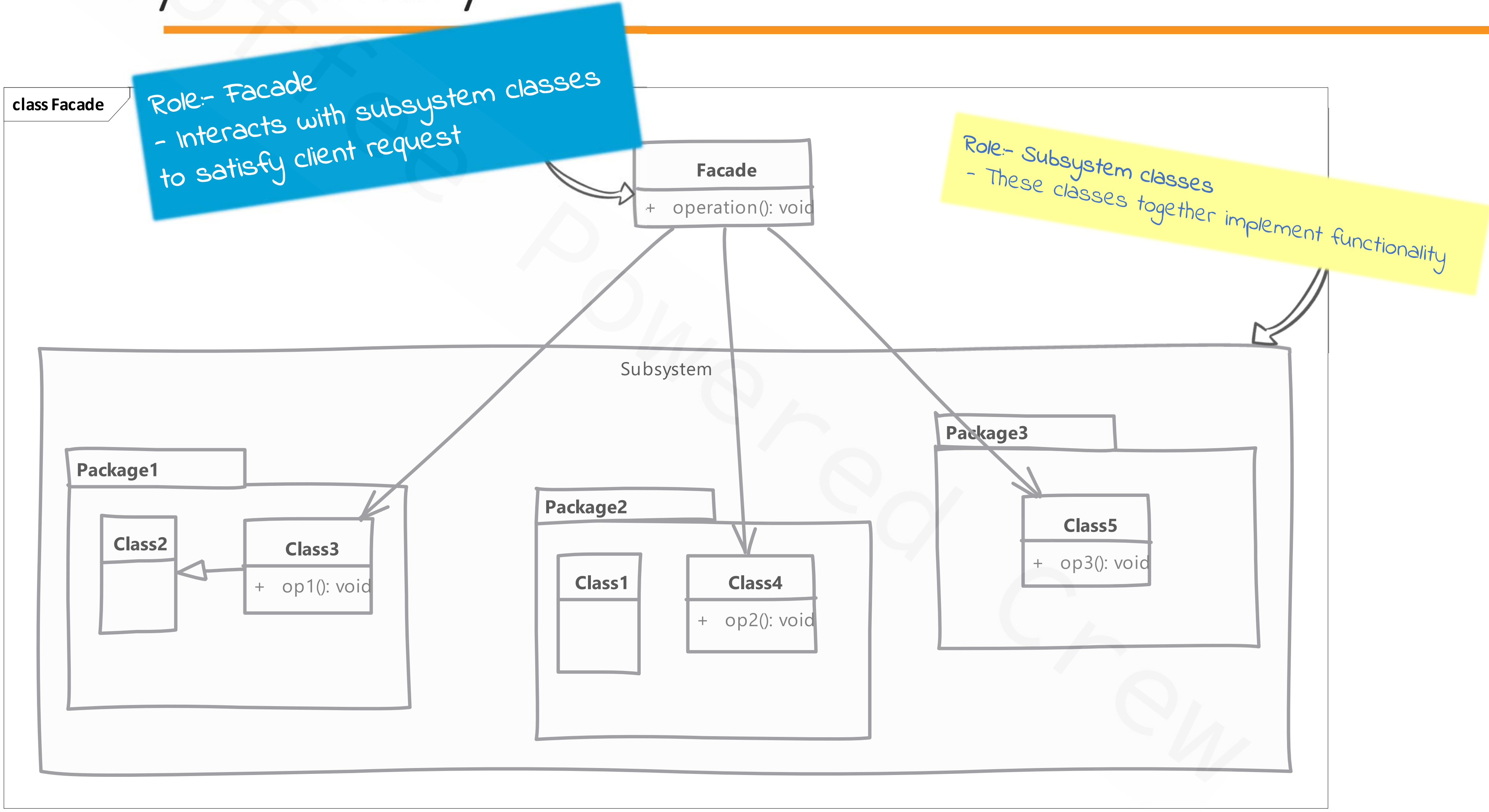
Structural Design Patterns

Design Patterns in Java

In-A-Hurry Summary

- We use façade when using our subsystem requires dealing with lots of classes & interfaces for client. Using façade we provide a simple interface which provides same functionality.
- Façade is not a simple method forwarding but façade methods encapsulate the subsystem class interactions which otherwise would have been done by client code.
- Facades are often added over existing legacy codes to simplify code usage & reduce coupling of client code to legacy code.

In-A-Hurry Summary



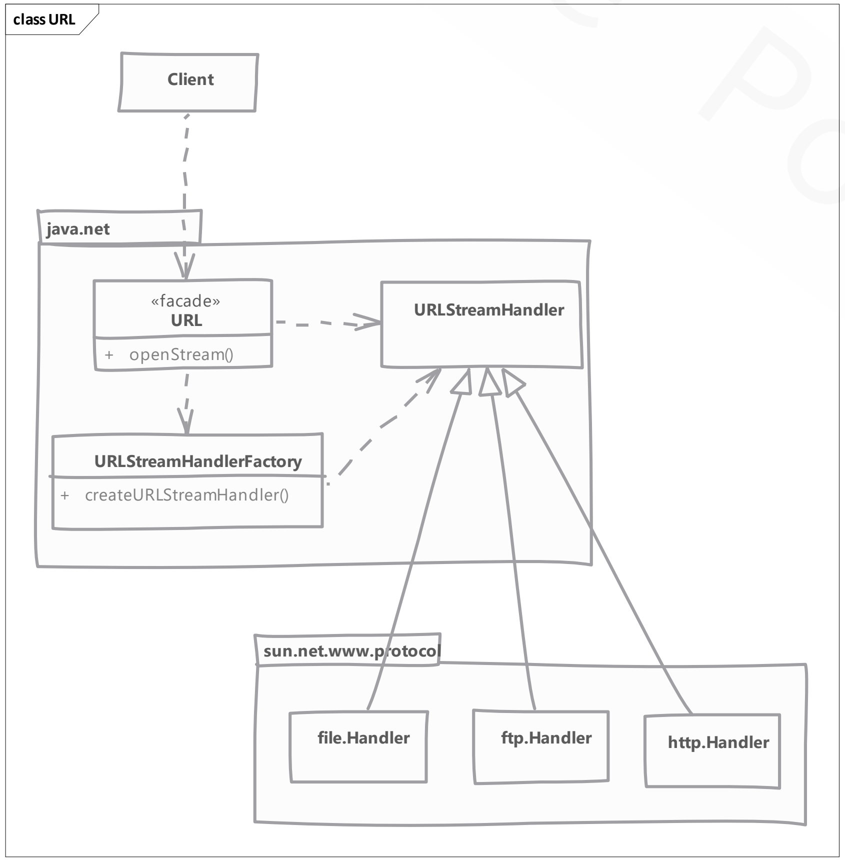
In-A-Hurry Summary

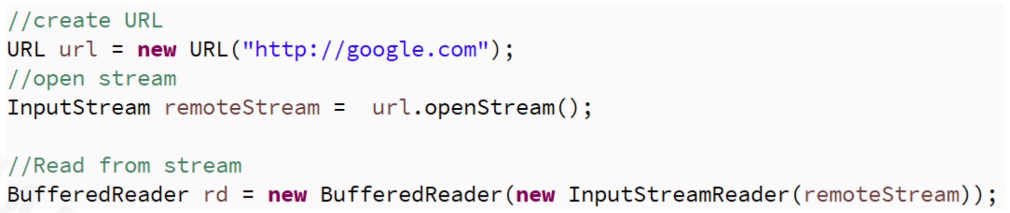
Facade

```java
//Facade provides simple methods for client to use
public class EmailFacade {
    //Method handles interactions with subsystem classes
    public boolean sendOrderEMail(Order order) {
        Template template = TemplateFactory.createTemplateFor(TemplateType.Email);
        Stationary stationary = StationaryFactory.createStationary();
        Email email = Email.getBuilder()
                      .withTemplate(template)
                      .withStationary(stationary)
                      .forObject(order)
                      .build();

        Mailer mailer = Mailer.getMailer();
        return mailer.send(email);
    }
}
```

Example of a Facade

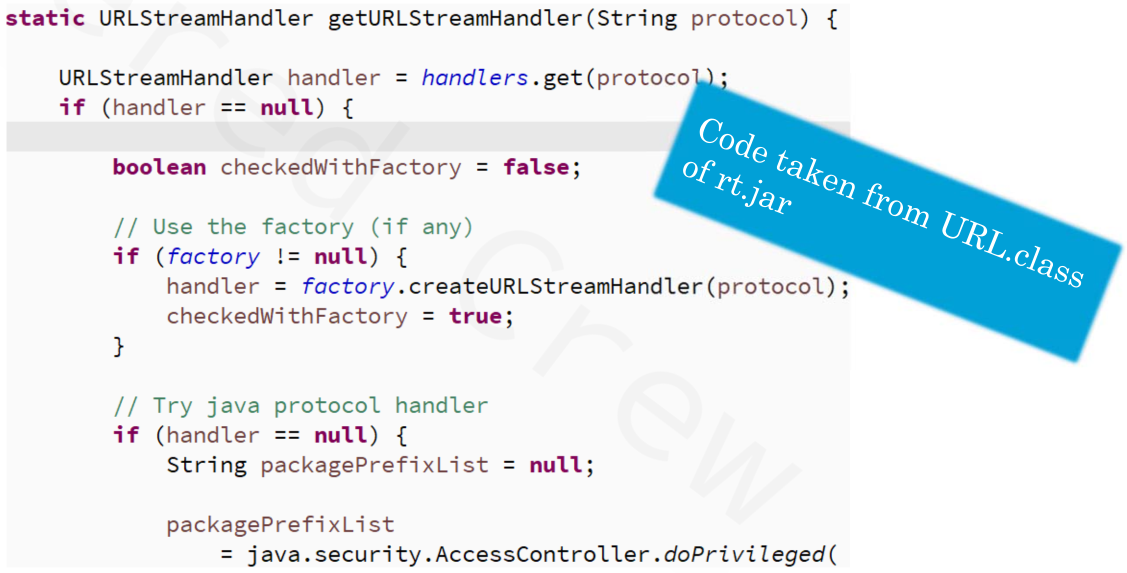
- The java.net.URL class is a great example of façade. This class provides a simple method called as openStream() and we get an input stream to the resource pointed at by the URL object.
- This class takes care of dealing with multiple classes from the java.net package as well as some internal sun packages.





```
//create URL
URL url = new URL("http://google.com");
//open stream
InputStream remoteStream =  url.openStream();

//Read from stream
BufferedReader rd = new BufferedReader(new InputStreamReader(remoteStream));
```

URL.class



```
static URLStreamHandler getURLStreamHandler(String protocol) {

    URLStreamHandler handler = handlers.get(protocol);
    if (handler == null) {

        boolean checkedWithFactory = false;

        // Use the factory (if any)
        if (factory != null) {
            handler = factory.createURLStreamHandler(protocol);
            checkedWithFactory = true;
        }

        // Try java protocol handler
        if (handler == null) {
            String packagePrefixList = null;

            packagePrefixList
                = java.security.AccessController.doPrivileged(
```

Flyweight

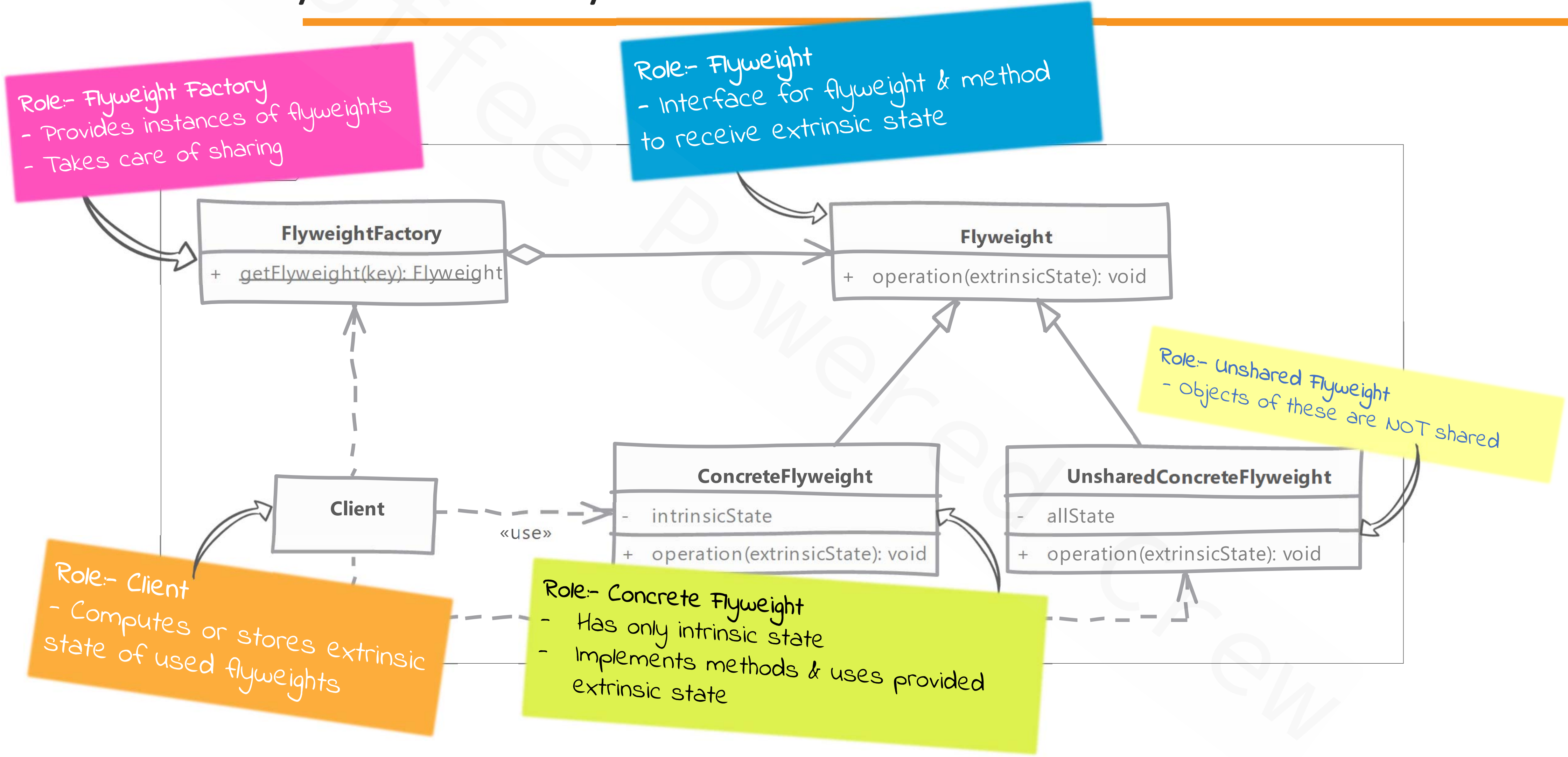
Structural Design Patterns

Design Patterns in Java

In-A-Hurry Summary

- We use flyweight design pattern if we need large number of objects of class where we can easily separate out state that can be shared and state that can be externalized.
- Flyweights store only “intrinsic” state or state that can be shared in any context.
- Code using flyweight instance provides the extrinsic state when calling methods on flyweight. Flyweight object then uses this state along with its inner state to carry out the work.
- Client code can store extrinsic per flyweight instance it uses or compute it on the fly.

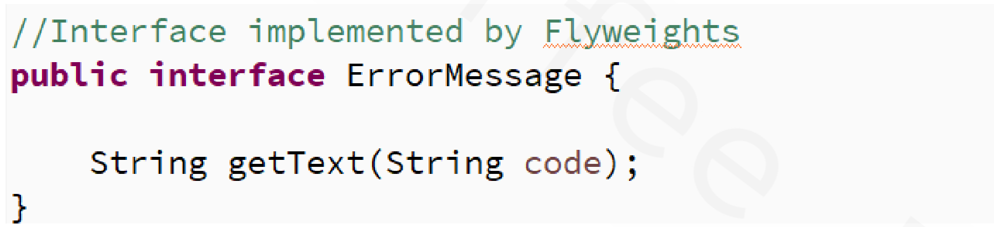
In-A-Hurry Summary



Role:- Flyweight Factory
- Provides instances of flyweights
- Takes care of sharing
Role:- Flyweight
- Interface for flyweight & method to receive extrinsic state
FlyweightFactory
+ getFlyweight(key): Flyweight
Flyweight
+ operation(extrinsicState): void
Role:- Unshared Flyweight
- Objects of these are NOT shared
ConcreteFlyweight
- intrinsicState
+ operation(extrinsicState): void
UnsharedConcreteFlyweight
- allState
+ operation(extrinsicState): void
Client
«use»
Role:- Client
- Computes or stores extrinsic state of used flyweights
Role:- Concrete Flyweight
- Has only intrinsic state
- Implements methods & uses provided extrinsic state

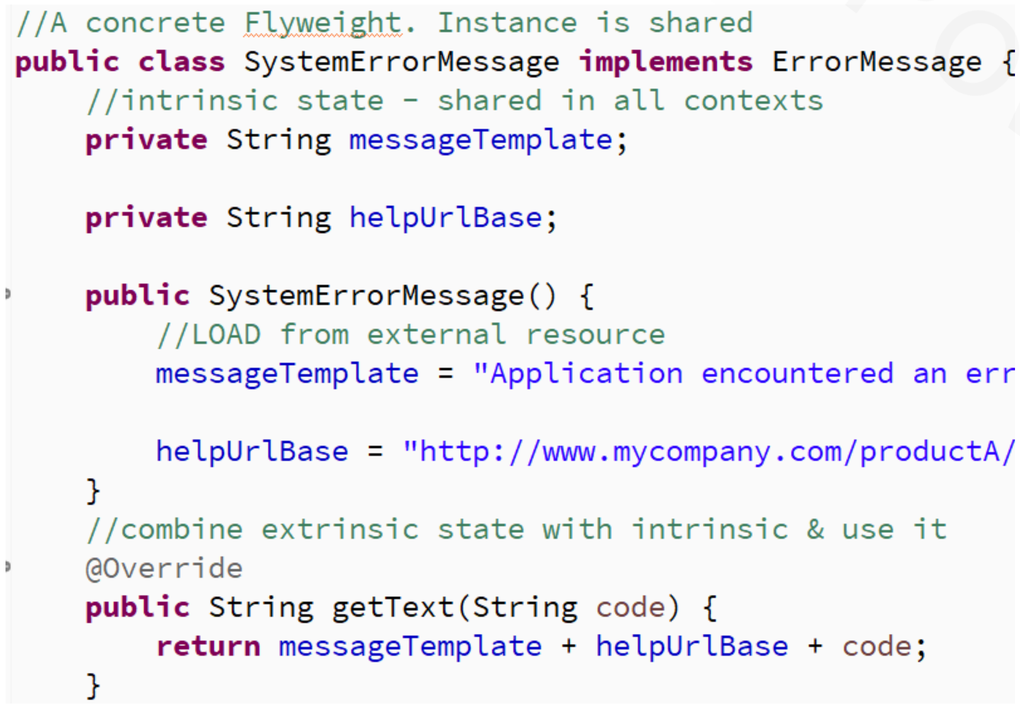
In-A-Hurry Summary

Flyweight



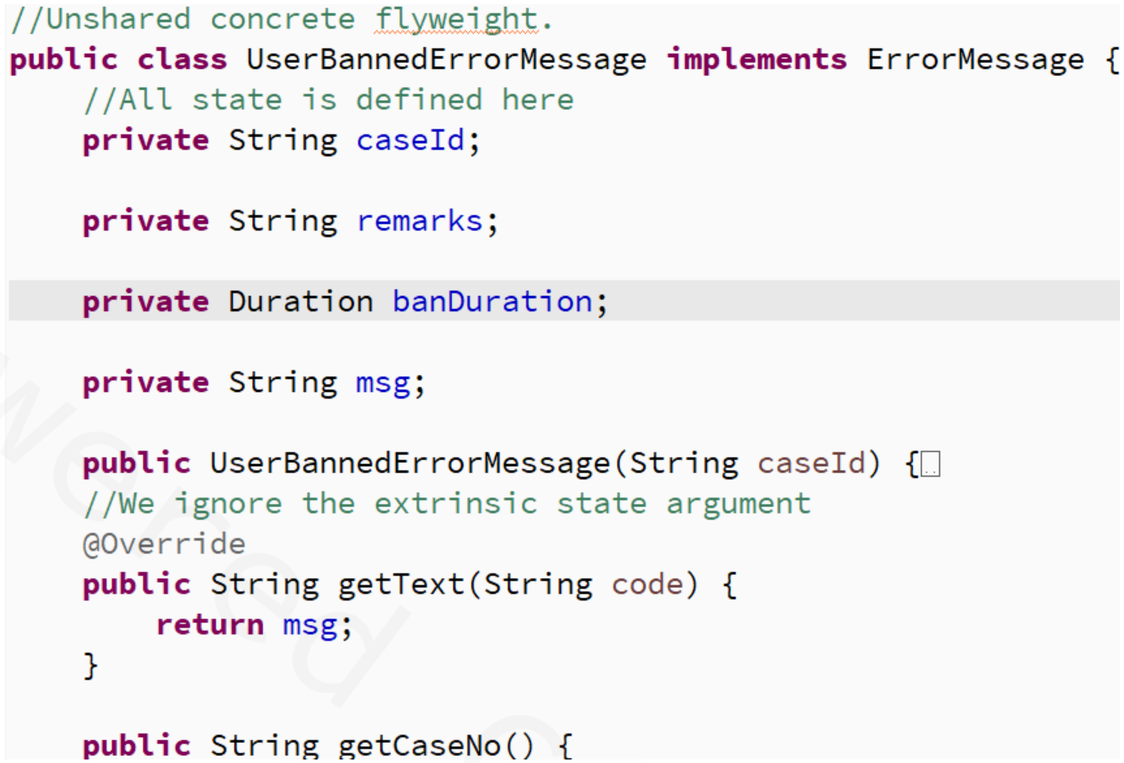
```
//Interface implemented by Flyweights
public interface ErrorMessage {

    String getText(String code);
}
```

Concrete Flyweight



```
//A concrete Flyweight. Instance is shared
public class SystemErrorMessage implements ErrorMessage {
    //intrinsic state - shared in all contexts
    private String messageTemplate;

    private String helpUrlBase;

    public SystemErrorMessage() {
        //LOAD from external resource
        messageTemplate = "Application encountered an err

        helpUrlBase = "http://www.mycompany.com/productA/
    }
    //combine extrinsic state with intrinsic & use it
    @Override
    public String getText(String code) {
        return messageTemplate + helpUrlBase + code;
    }
```

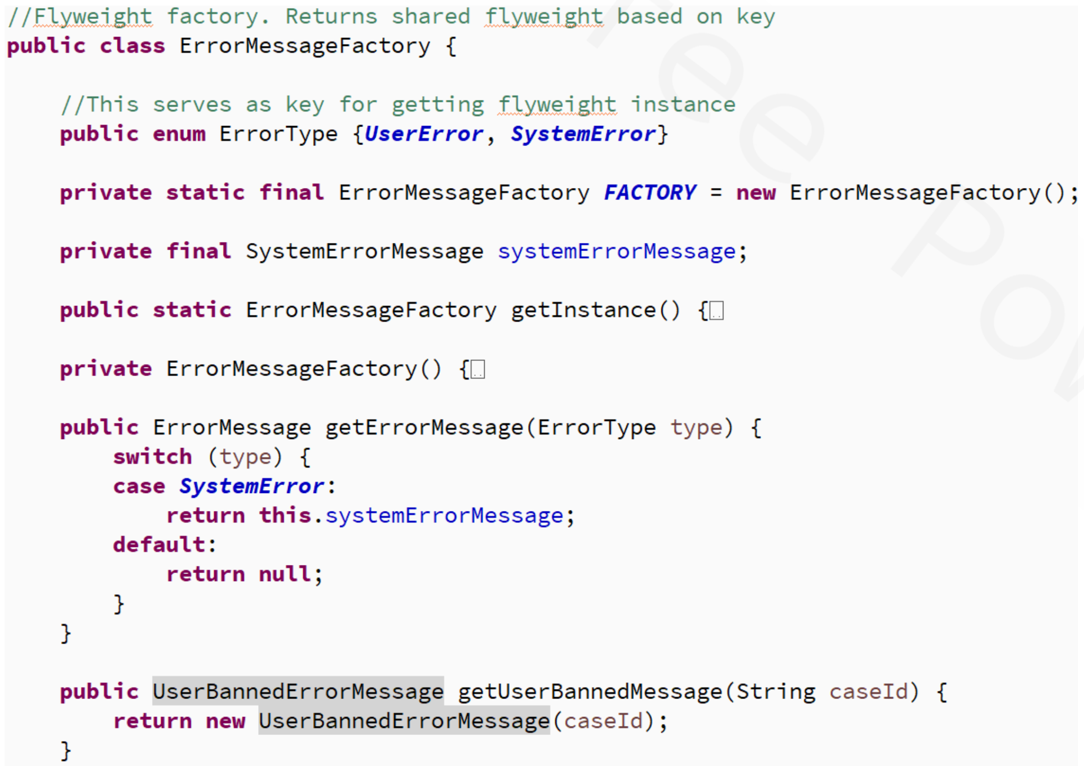
Unshared Flyweight



```
//Unshared concrete flyweight.
public class UserBannedErrorMessage implements ErrorMessage {
    //All state is defined here
    private String caseId;

    private String remarks;

    private Duration banDuration;

    private String msg;

    public UserBannedErrorMessage(String caseId) {
    //We ignore the extrinsic state argument
    @Override
    public String getText(String code) {
        return msg;
    }

    public String getCaseNo() {
```

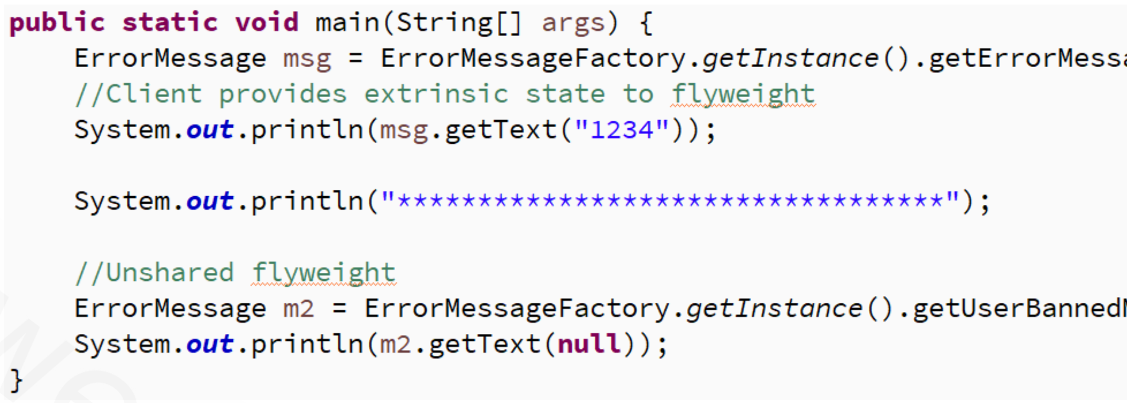
In-A-Hurry Summary

Flyweight Factory



```
//Flyweight factory. Returns shared flyweight based on key
public class ErrorMessageFactory {

    //This serves as key for getting flyweight instance
    public enum ErrorType {UserError, SystemError}

    private static final ErrorMessageFactory FACTORY = new ErrorMessageFactory();

    private final SystemErrorMessage systemErrorMessage;

    public static ErrorMessageFactory getInstance() {□

    private ErrorMessageFactory() {□

    public ErrorMessage getErrorMessage(ErrorType type) {
        switch (type) {
        case SystemError:
            return this.systemErrorMessage;
        default:
            return null;
        }
    }

    public UserBannedErrorMessage getUserBannedMessage(String caseId) {
        return new UserBannedErrorMessage(caseId);
    }
```

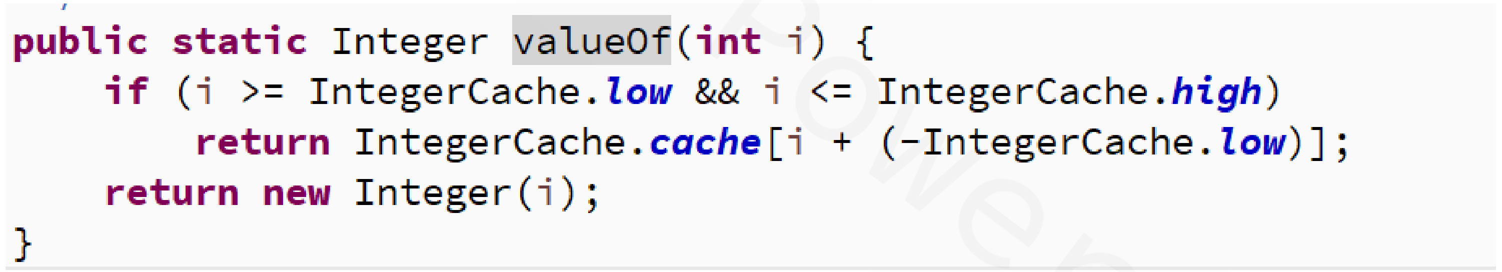
Client



```
public static void main(String[] args) {
    ErrorMessage msg = ErrorMessageFactory.getInstance().getErrorMess
    //Client provides extrinsic state to flyweight
    System.out.println(msg.getText("1234"));

    System.out.println("*********************************");

    //Unshared flyweight
    ErrorMessage m2 = ErrorMessageFactory.getInstance().getUserBannedM
    System.out.println(m2.getText(null));
}
```

Examples of a Flyweight

- Java uses flyweight pattern for Wrapper classes like java.lang.Integer, Short, Byte etc. Here the valueOf static method serves as the factory method.



```java
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

- String pool which is maintained by JVM is also an example of flyweight. We can call the intern() method on a String object to explicitly request this String object to be interned. This method will returned a reference to already cached object if present or else will create new String in cache if not present.

Note:- String.intern() is a native method.

Proxy

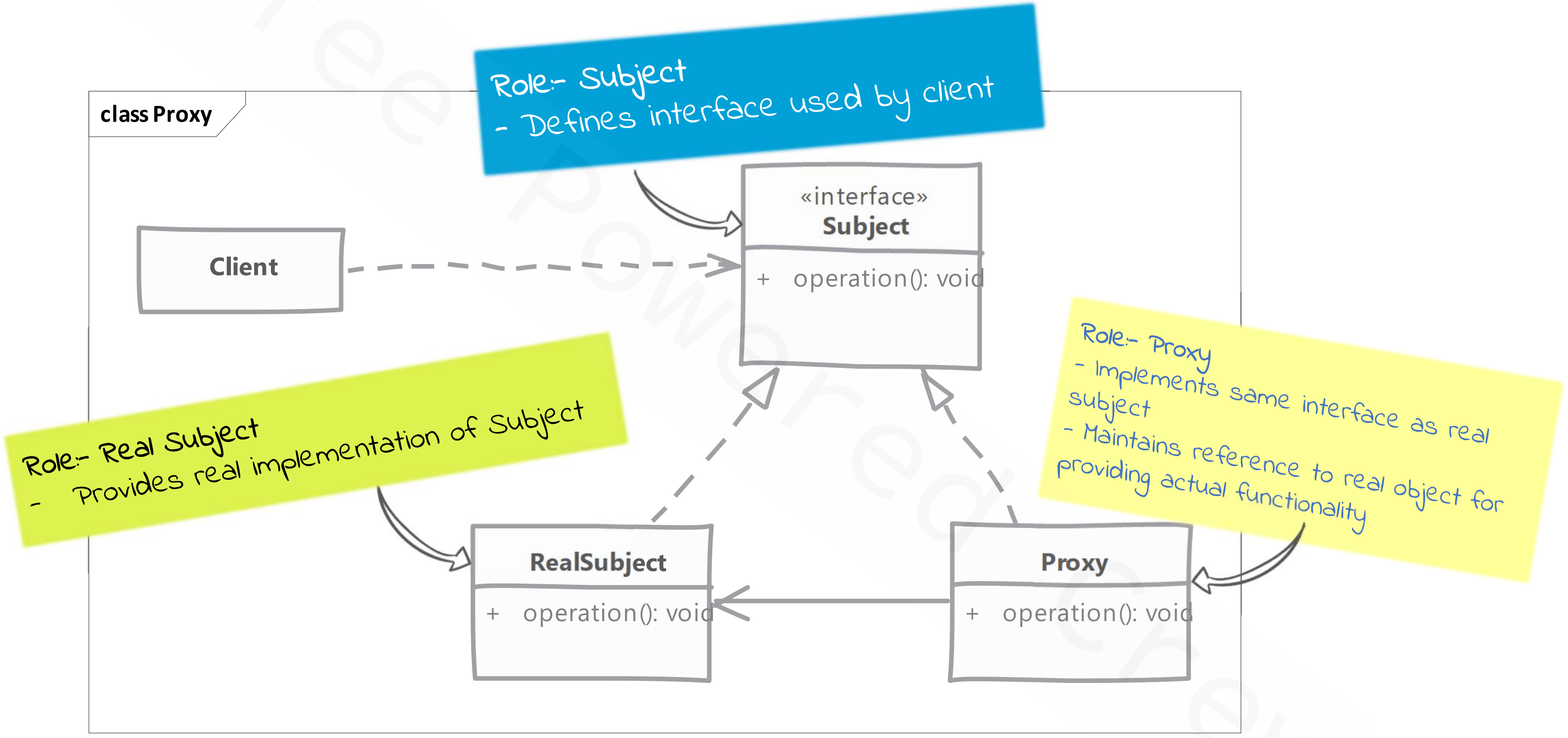
Structural Design Patterns

Design Patterns in Java

In-A-Hurry Summary

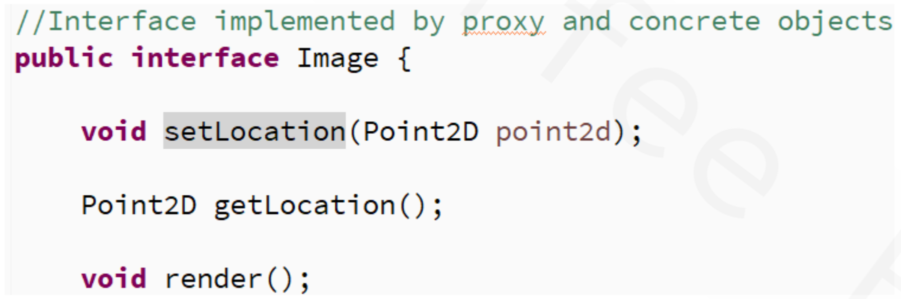
- We want a stand in or placeholder object or we want control access to our objects method, then we can use proxy pattern.
- Proxy implements same interface as expected of real object. It delegates actual functionality to real object. Proxies are either given real object or they create one when needed. Some proxies talk to remote service behind the scene.
- Main usage of proxies is for:
    - Protection Proxy - Control access to original object's operations
    - Remote Proxy – Provides a local representation of a remote object.
    - Virtual proxy – Delays construction of original object until absolutely necessary
- In java we can also use dynamic proxies. These are created on the fly at runtime.

In-A-Hurry Summary


class Proxy
Role:- Subject
- Defines interface used by client
«interface»
Subject
+ operation(): void
Client
Role:- Real Subject
- Provides real implementation of Subject
RealSubject
+ operation(): void
Proxy
+ operation(): void
Role:- Proxy
- Implements same interface as real subject
- Maintains reference to real object for providing actual functionality

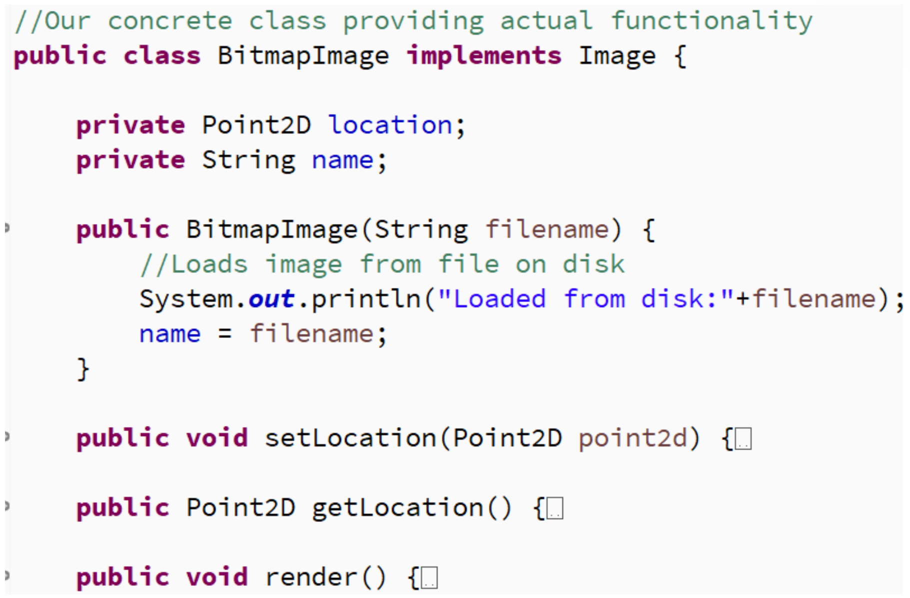
In-A-Hurry Summary

Subject Interface



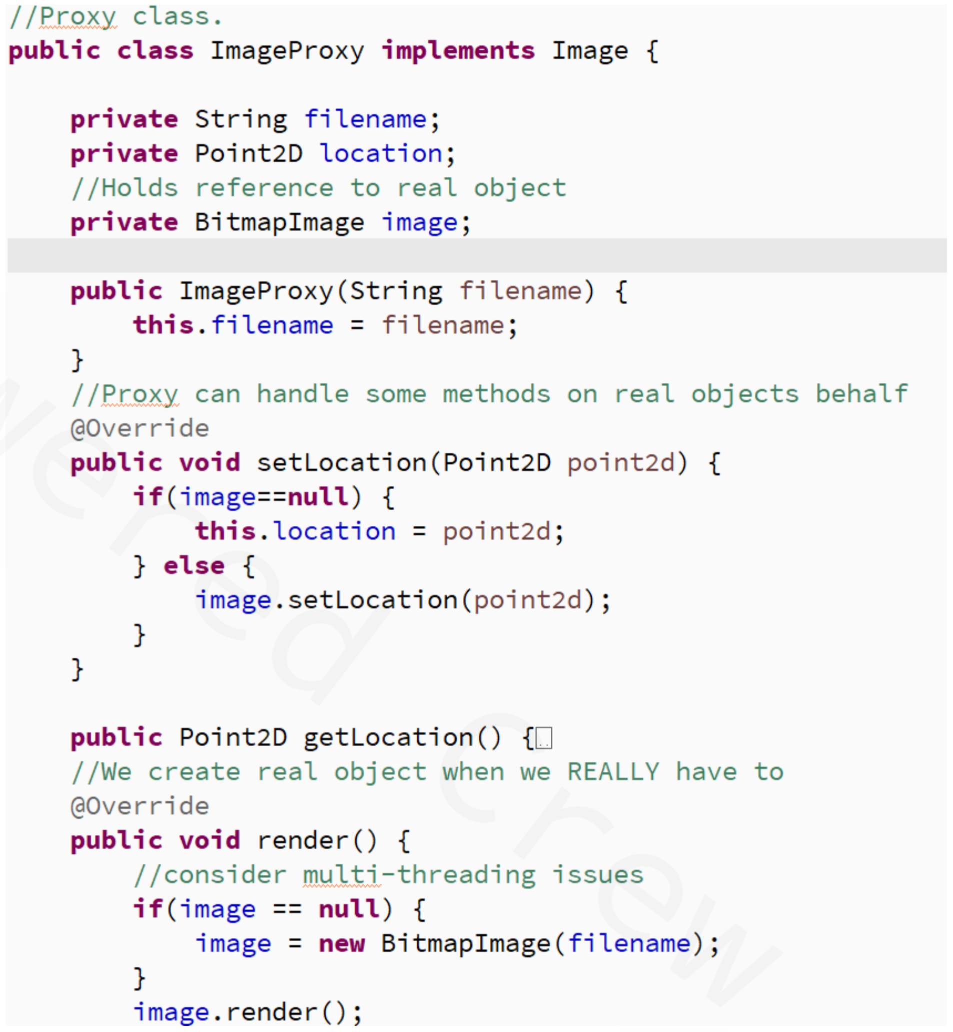
```java
//Interface implemented by proxy and concrete objects
public interface Image {

    void setLocation(Point2D point2d);

    Point2D getLocation();

    void render();
```

Concrete Subject



```java
//Our concrete class providing actual functionality
public class BitmapImage implements Image {

    private Point2D location;
    private String name;

    public BitmapImage(String filename) {
        //Loads image from file on disk
        System.out.println("Loaded from disk:"+filename);
        name = filename;
    }

    public void setLocation(Point2D point2d) {□

    public Point2D getLocation() {□

    public void render() {□
```

Proxy



```java
//Proxy class.
public class ImageProxy implements Image {

    private String filename;
    private Point2D location;
    //Holds reference to real object
    private BitmapImage image;

    public ImageProxy(String filename) {
        this.filename = filename;
    }
    //Proxy can handle some methods on real objects behalf
    @Override
    public void setLocation(Point2D point2d) {
        if(image==null) {
            this.location = point2d;
        } else {
            image.setLocation(point2d);
        }
    }

    public Point2D getLocation() {□
    //We create real object when we REALLY have to
    @Override
    public void render() {
        //consider multi-threading issues
        if(image == null) {
            image = new BitmapImage(filename);
        }
        image.render();
```

In-A-Hurry Summary

Image Factory

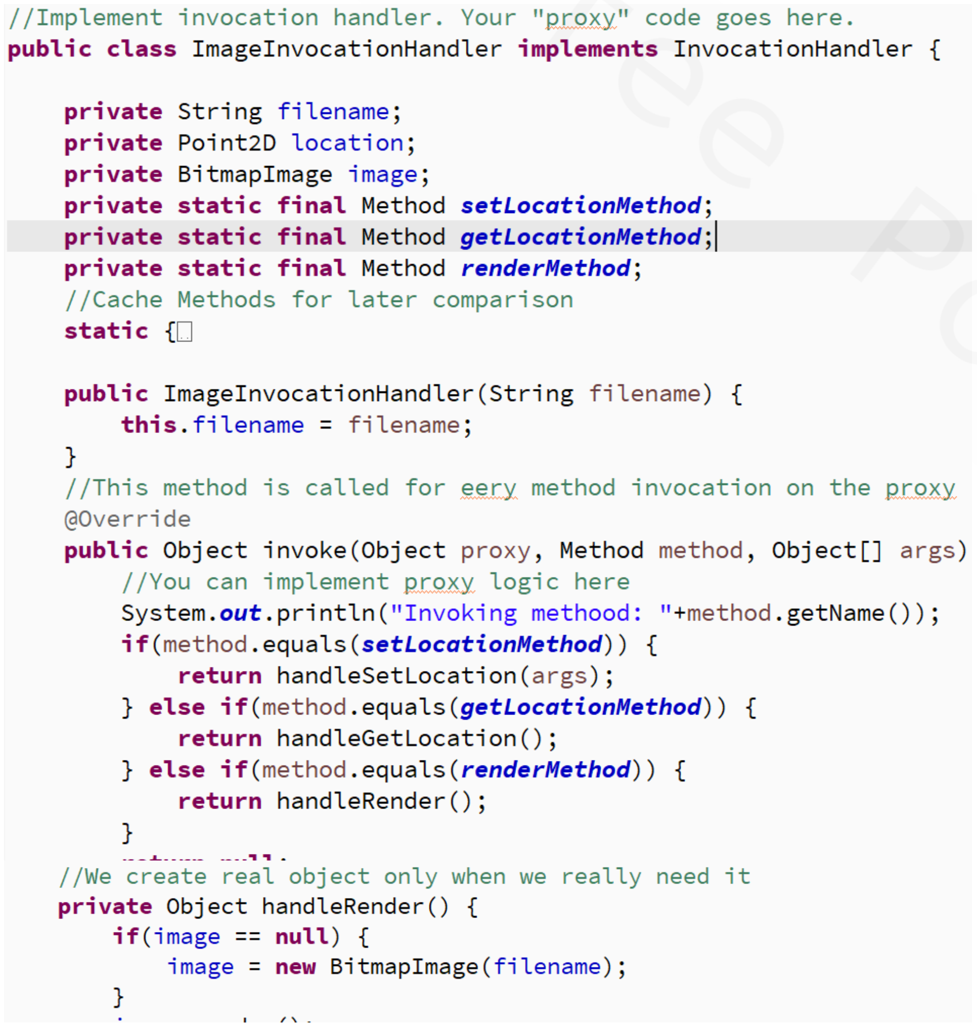
```
//Factory to get image objects.
public class ImageFactory {
    //We'll provide proxy to caller instead of real object
    public static Image getImage(String filename) {
        return new ImageProxy(filename);
    }
}
```

Client

```
public static void main(String[] args) {
    //Get image from factory
    Image img = ImageFactory.getImage("Image1.bmp");
    //Client is unaware of proxy returned by factory
    img.setLocation(new Point2D(10, 10));
    System.out.println("Image Location: "+img.getLocation());
    System.out.println("Starting render...");

    img.render();
}
```

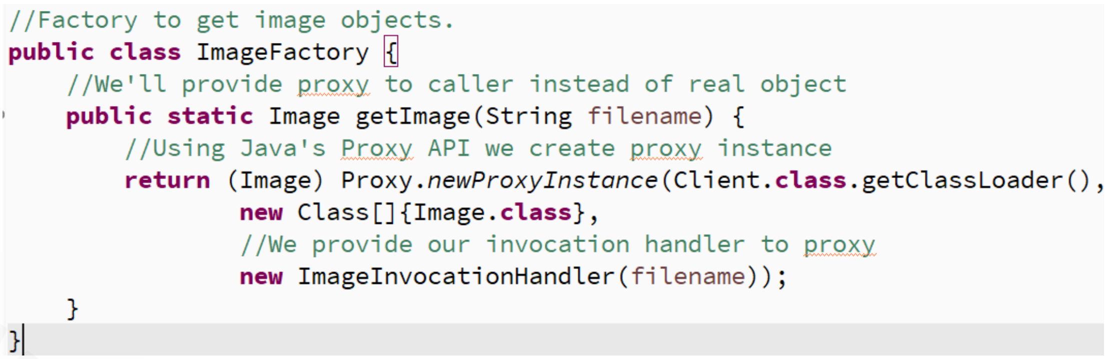
In-A-Hurry Summary

Image Invocation Handler



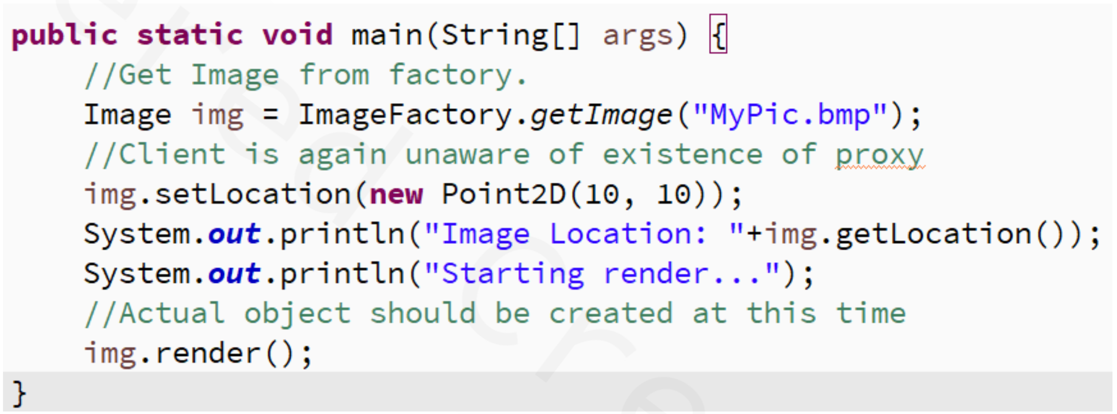
```
//Implement invocation handler. Your "proxy" code goes here.
public class ImageInvocationHandler implements InvocationHandler {

    private String filename;
    private Point2D location;
    private BitmapImage image;
    private static final Method setLocationMethod;
    private static final Method getLocationMethod;
    private static final Method renderMethod;
    //Cache Methods for later comparison
    static {

    public ImageInvocationHandler(String filename) {
        this.filename = filename;
    }
    //This method is called for eery method invocation on the proxy
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        //You can implement proxy logic here
        System.out.println("Invoking methood: "+method.getName());
        if(method.equals(setLocationMethod)) {
            return handleSetLocation(args);
        } else if(method.equals(getLocationMethod)) {
            return handleGetLocation();
        } else if(method.equals(renderMethod)) {
            return handleRender();
        }
    //We create real object only when we really need it
    private Object handleRender() {
        if(image == null) {
            image = new BitmapImage(filename);
        }
```

Image Factory



```
//Factory to get image objects.
public class ImageFactory {
    //We'll provide proxy to caller instead of real object
    public static Image getImage(String filename) {
        //Using Java's Proxy API we create proxy instance
        return (Image) Proxy.newProxyInstance(Client.class.getClassLoader(),
                new Class[]{Image.class},
                //We provide our invocation handler to proxy
                new ImageInvocationHandler(filename));
    }
}
```

Client



```
public static void main(String[] args) {
    //Get Image from factory.
    Image img = ImageFactory.getImage("MyPic.bmp");
    //Client is again unaware of existence of proxy
    img.setLocation(new Point2D(10, 10));
    System.out.println("Image Location: "+img.getLocation());
    System.out.println("Starting render...");
    //Actual object should be created at this time
    img.render();
}
```

Examples of a Proxy

- This is one pattern where you'll find numerous examples ☺
- Hibernate uses a proxy to load collections of value types. If you have a relationship in entity class mapped as a collection, marked as candidate for lazy loading then Hibernate will provide a virtual proxy in its place.
- Spring uses proxy pattern to provide support for features like transactions, caching and general AOP support.
- Hibernate & spring both can create proxies for classes which do not implement any interface. They use third party frameworks like cglib, aspectJ to create dynamic proxies (remember, Java's dynamic proxy needs interface) at runtime.

Chain of Responsibility

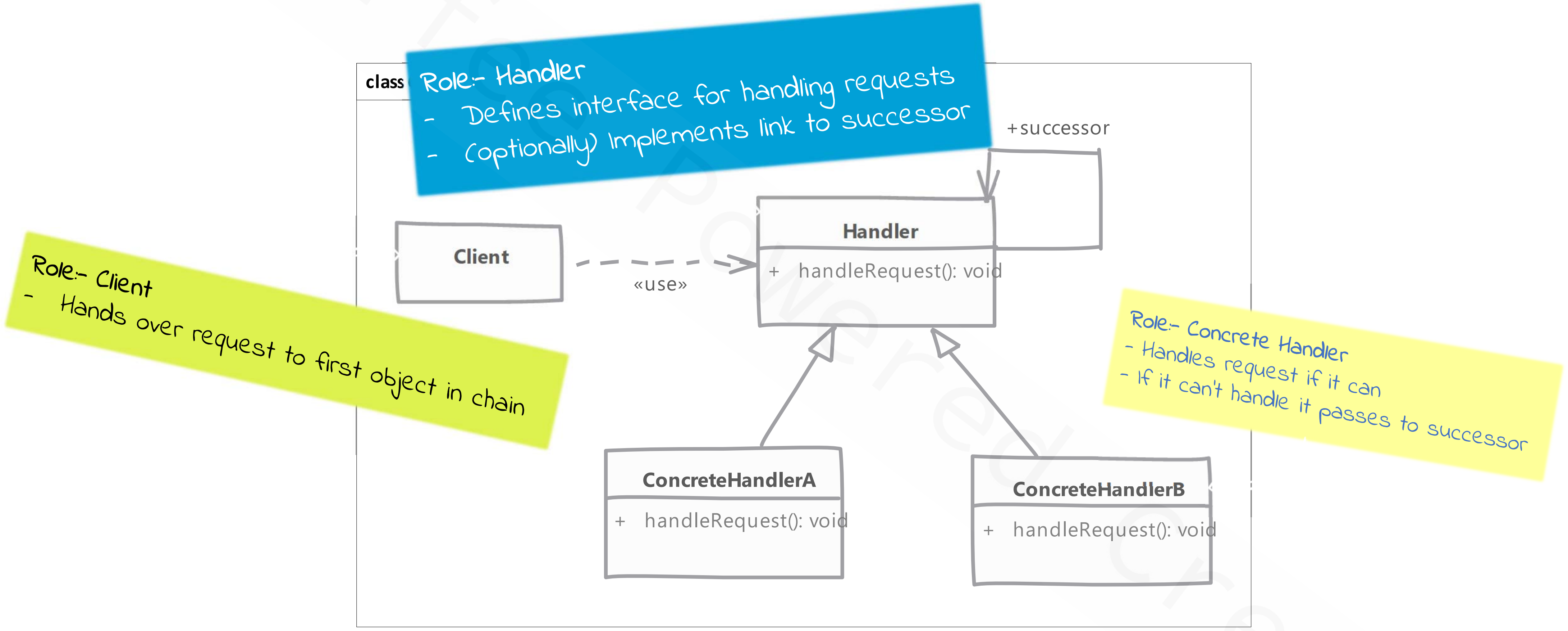
Behavioral Design Patterns

Design Patterns in Java

In-A-Hurry Summary

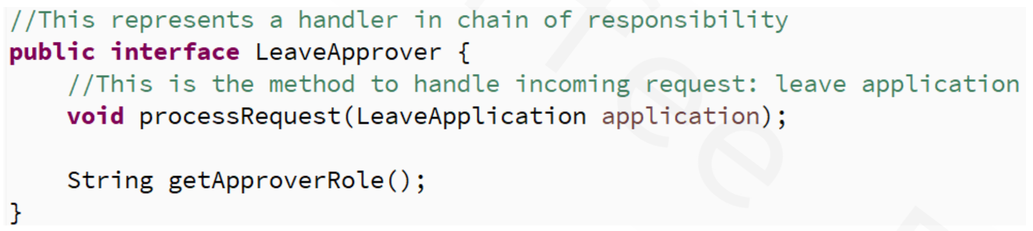
- When we want to decouple sender of request from the object which handles the request, we use chain of responsibility.
- We want this decoupling because we want to give multiple objects chance to handle the request & we don’t know all objects before hand.
- A handler checks if it can handle the request. If it can’t then it’ll pass the request on to next handler in chain.
- You can pass the request down the chain even if a handler handles the request. Design pattern doesn’t prevent that from happening.

In-A-Hurry Summary


class
Role:- Handler
- Defines interface for handling requests
- (optionally) Implements link to successor
+successor
Handler
+ handleRequest(): void
Client
«use»
Role:- Client
- Hands over request to first object in chain
Role:- Concrete Handler
- Handles request if it can
- If it can't handle it passes to successor
ConcreteHandlerA
+ handleRequest(): void
ConcreteHandlerB
+ handleRequest(): void

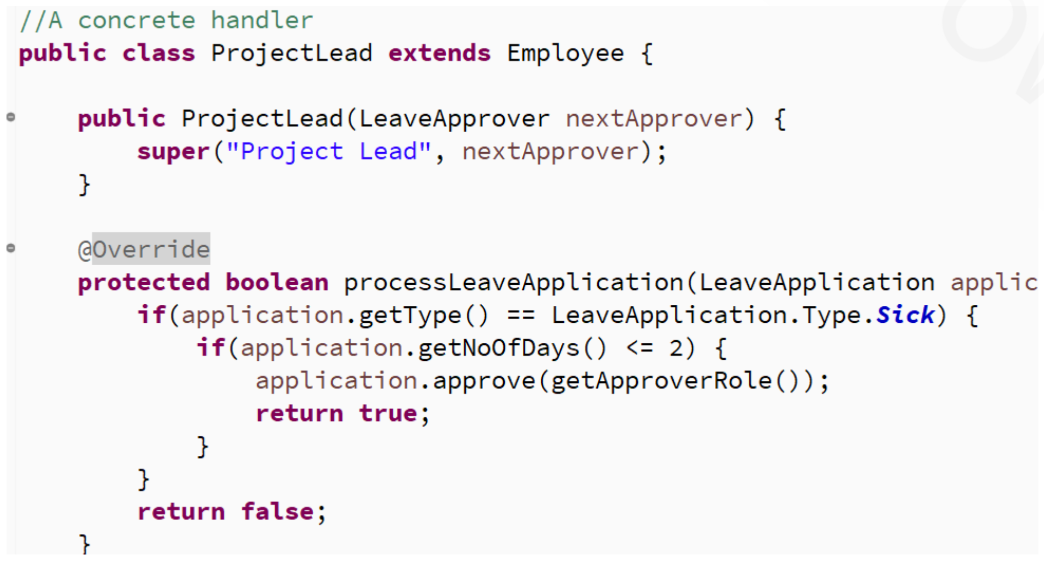
In-A-Hurry Summary

Handler



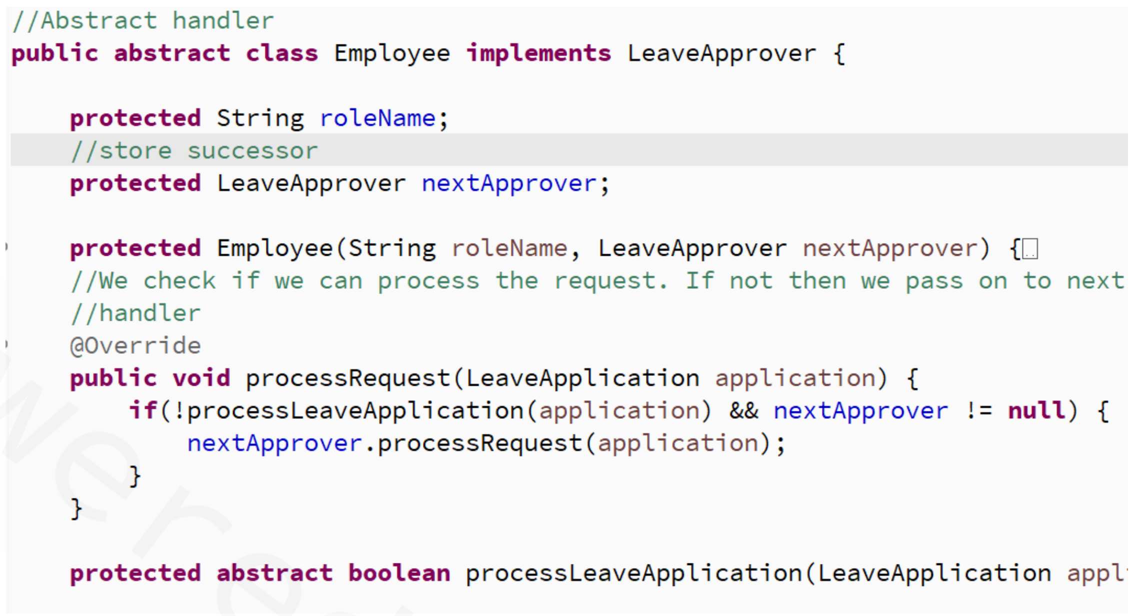
```java
//This represents a handler in chain of responsibility
public interface LeaveApprover {
    //This is the method to handle incoming request: leave application
    void processRequest(LeaveApplication application);

    String getApproverRole();
}
```

Concrete Handler



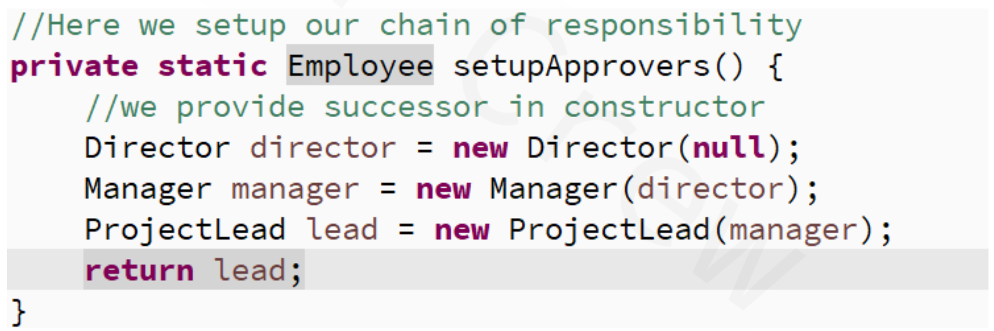
```java
//A concrete handler
public class ProjectLead extends Employee {

    public ProjectLead(LeaveApprover nextApprover) {
        super("Project Lead", nextApprover);
    }

    @Override
    protected boolean processLeaveApplication(LeaveApplication applic
        if(application.getType() == LeaveApplication.Type.Sick) {
            if(application.getNoOfDays() <= 2) {
                application.approve(getApproverRole());
                return true;
            }
        }
        return false;
    }
```

Abstract Handler



```java
//Abstract handler
public abstract class Employee implements LeaveApprover {

    protected String roleName;
    //store successor
    protected LeaveApprover nextApprover;

    protected Employee(String roleName, LeaveApprover nextApprover) {
    //We check if we can process the request. If not then we pass on to next
    //handler
    @Override
    public void processRequest(LeaveApplication application) {
        if(!processLeaveApplication(application) && nextApprover != null) {
            nextApprover.processRequest(application);
        }
    }

    protected abstract boolean processLeaveApplication(LeaveApplication appl
```

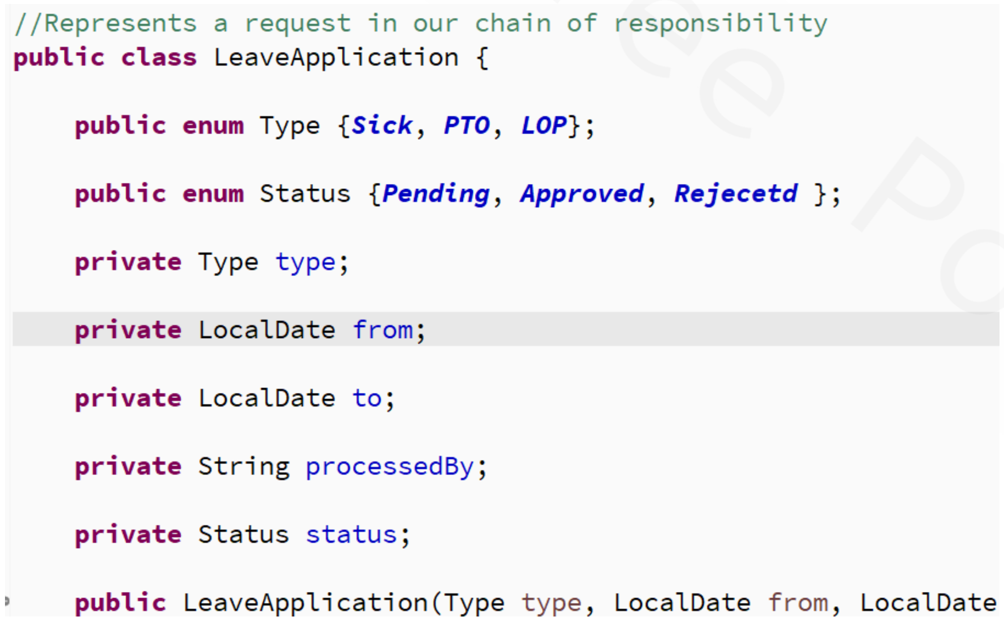
Setup chain of responsibility



```java
//Here we setup our chain of responsibility
private static Employee setupApprovers() {
    //we provide successor in constructor
    Director director = new Director(null);
    Manager manager = new Manager(director);
    ProjectLead lead = new ProjectLead(manager);
    return lead;
}
```

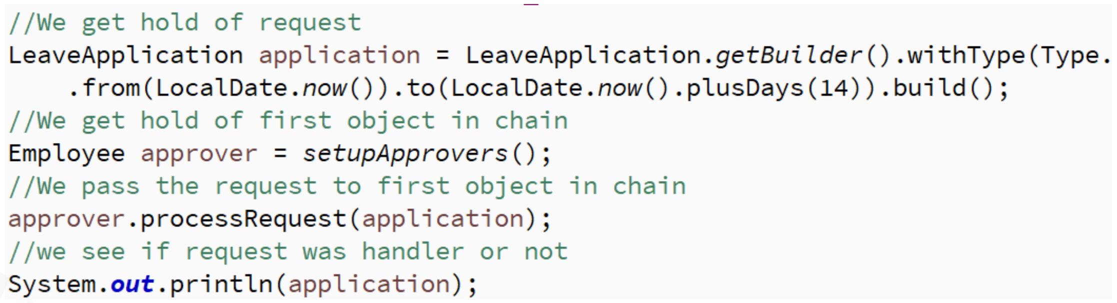
In-A-Hurry Summary

Request



```java
//Represents a request in our chain of responsibility
public class LeaveApplication {

    public enum Type {Sick, PTO, LOP};

    public enum Status {Pending, Approved, Rejecetd };

    private Type type;

    private LocalDate from;

    private LocalDate to;

    private String processedBy;

    private Status status;

    public LeaveApplication(Type type, LocalDate from, LocalDate
```

Client



```java
//We get hold of request
LeaveApplication application = LeaveApplication.getBuilder().withType(Type.
    .from(LocalDate.now()).to(LocalDate.now().plusDays(14)).build();
//We get hold of first object in chain
Employee approver = setupApprovers();
//We pass the request to first object in chain
approver.processRequest(application);
//we see if request was handler or not
System.out.println(application);
```

Example of a Chain of responsibility

- Probably the best example of chain of responsibility is servlet filters. Each filter gets a chance to handle incoming request and passes it down the chain once its work is done.
- All servlet filters implement the javax.servlet.Filter interface which defines following doFilter method

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
```

- Implementations will use FilterChain object to pass request to next handler in chain

  i.e. chain.doFilter(request, response);
- In servlet filters, it’s common practice to allow other filters to handle request even if current filter takes some action on the request.

Command

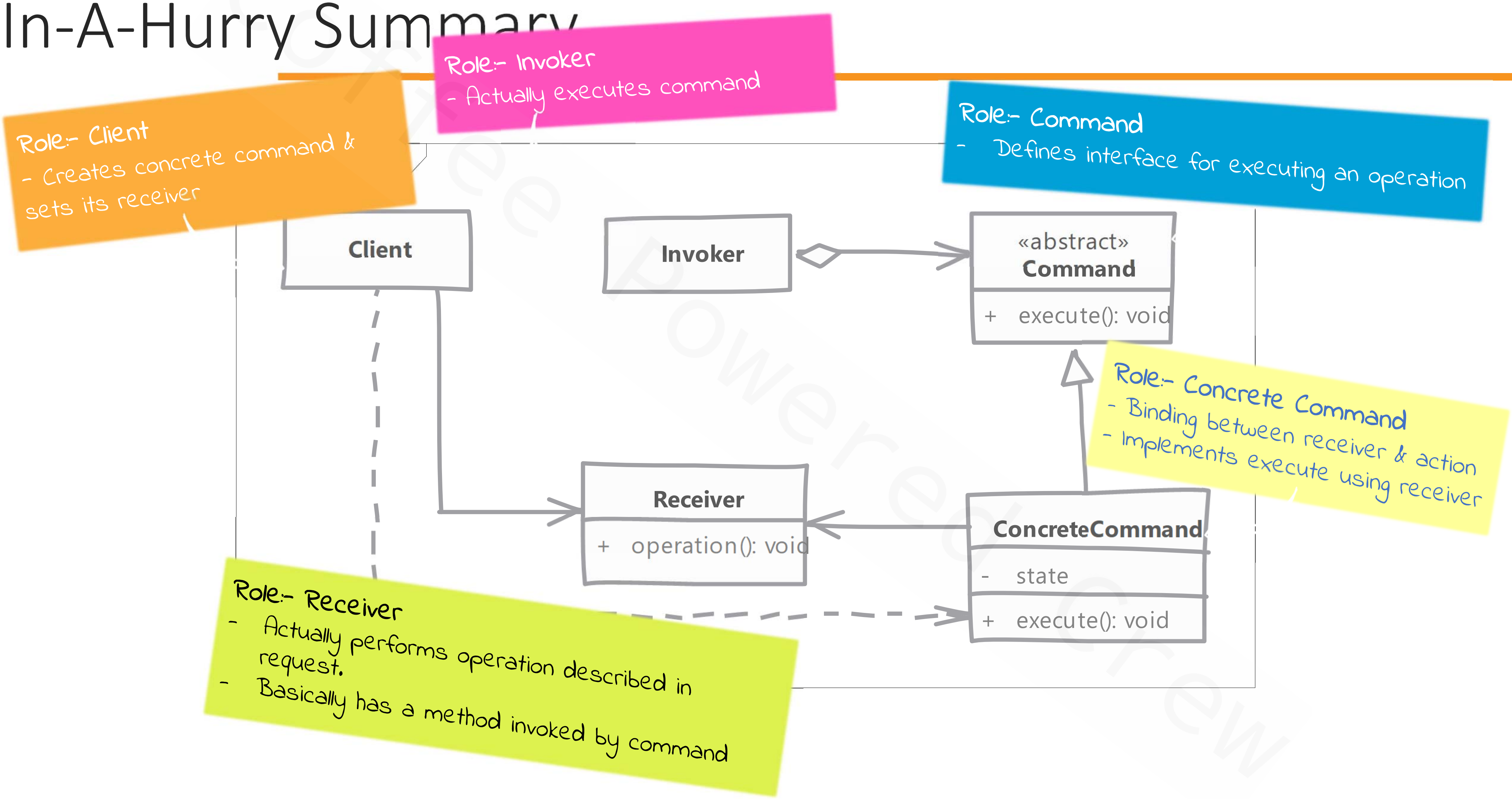
Behavioral Design Patterns

Design Patterns in Java

In-A-Hurry Summary

- Command pattern allows you to treat requests for operations as objects. This allows you to send these objects to different parts of code for later execution or to a different thread.
- Commands typically invoke the actual operation on a receiver but contain parameters or information needed for invocation.
- Client code is responsible for creating instances of command & providing it with receiver and request information.
- Commands can also implement an undo feature. Here command itself stores a snapshot of receiver.

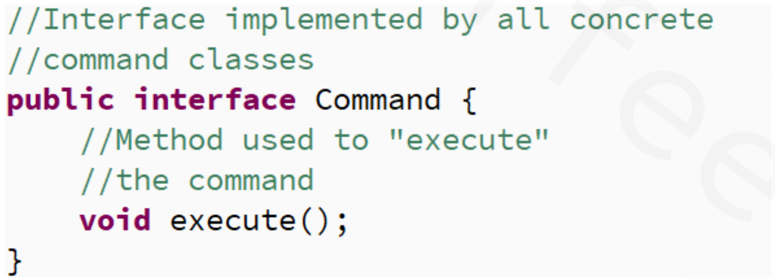
In-A-Hurry Summary



Role:- Client
- Creates concrete command & sets its receiver
Role:- Invoker
- Actually executes command
Role:- Command
- Defines interface for executing an operation
Client
Invoker
«abstract» Command
+ execute(): void
Role:- Concrete Command
- Binding between receiver & action
- Implements execute using receiver
Receiver
+ operation(): void
ConcreteCommand
- state
+ execute(): void
Role:- Receiver
- Actually performs operation described in request.
- Basically has a method invoked by command

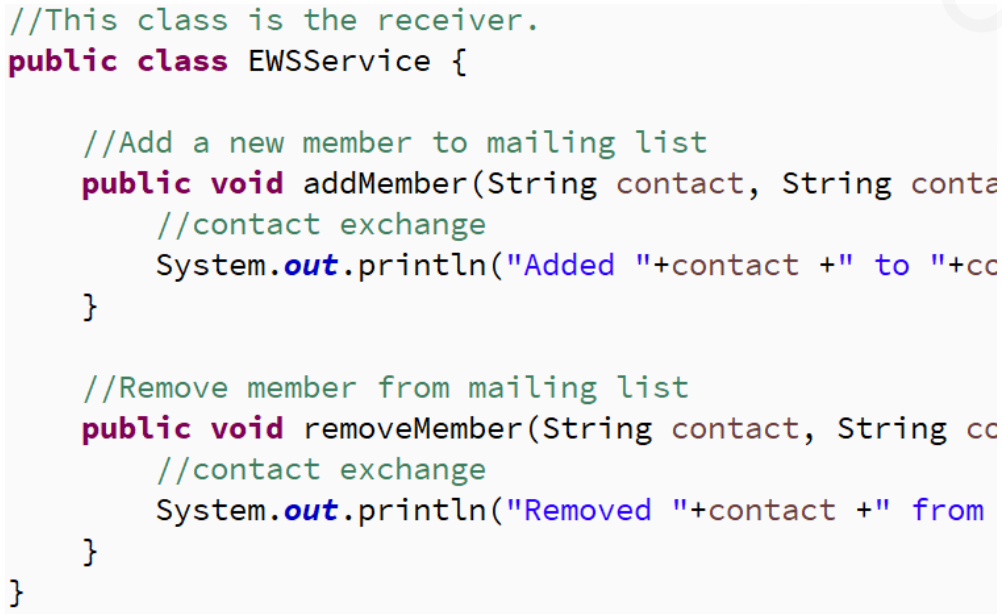
In-A-Hurry Summary

Command



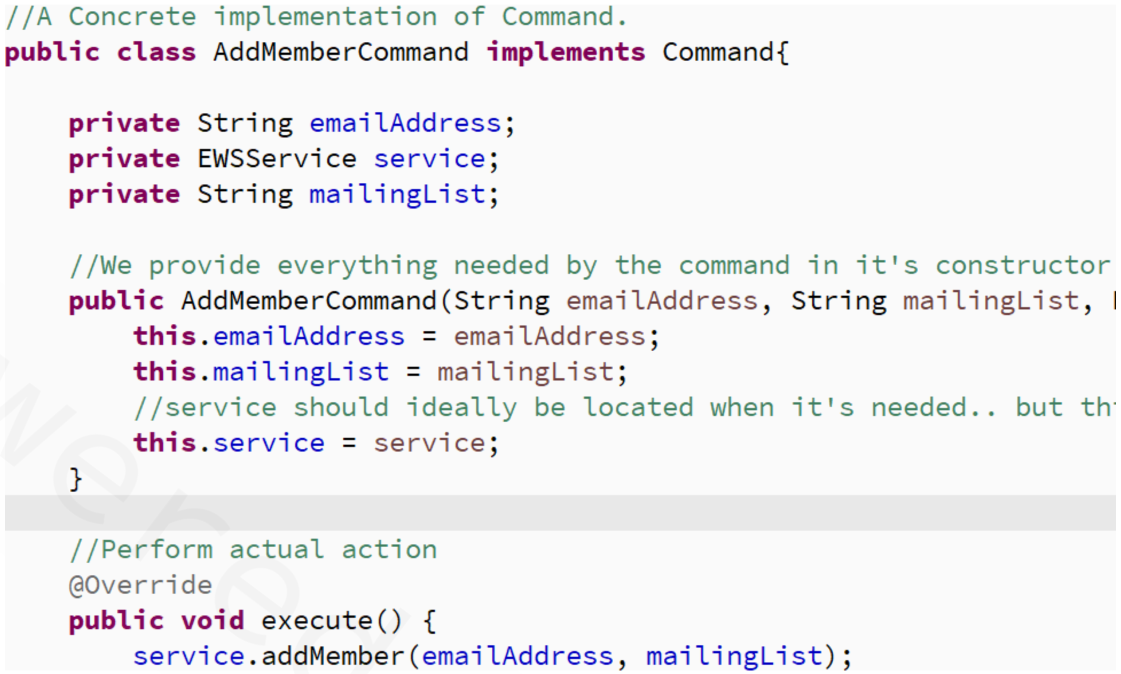
```
//Interface implemented by all concrete
//command classes
public interface Command {
    //Method used to "execute"
    //the command
    void execute();
}
```

Receiver



```
//This class is the receiver.
public class EWSService {

    //Add a new member to mailing list
    public void addMember(String contact, String conta
        //contact exchange
        System.out.println("Added "+contact +" to "+co
    }

    //Remove member from mailing list
    public void removeMember(String contact, String co
        //contact exchange
        System.out.println("Removed "+contact +" from
    }
}
```

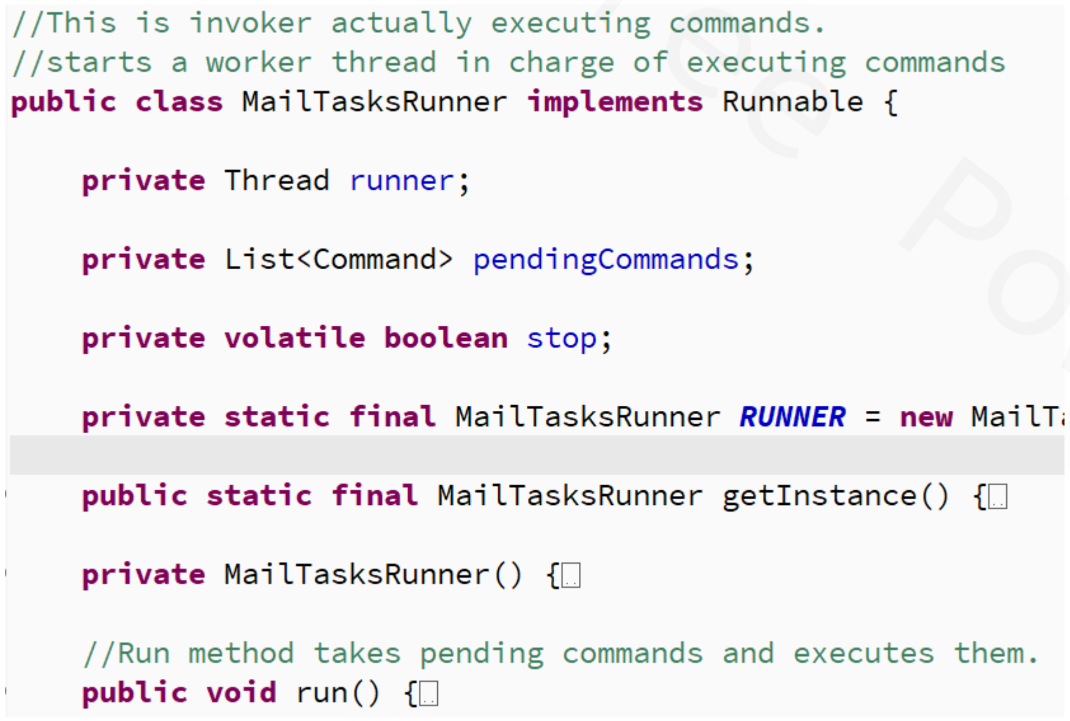
Concrete Command



```
//A Concrete implementation of Command.
public class AddMemberCommand implements Command{

    private String emailAddress;
    private EWSService service;
    private String mailingList;

    //We provide everything needed by the command in it's constructor
    public AddMemberCommand(String emailAddress, String mailingList,
        this.emailAddress = emailAddress;
        this.mailingList = mailingList;
        //service should ideally be located when it's needed.. but th
        this.service = service;
    }

    //Perform actual action
    @Override
    public void execute() {
        service.addMember(emailAddress, mailingList);
```

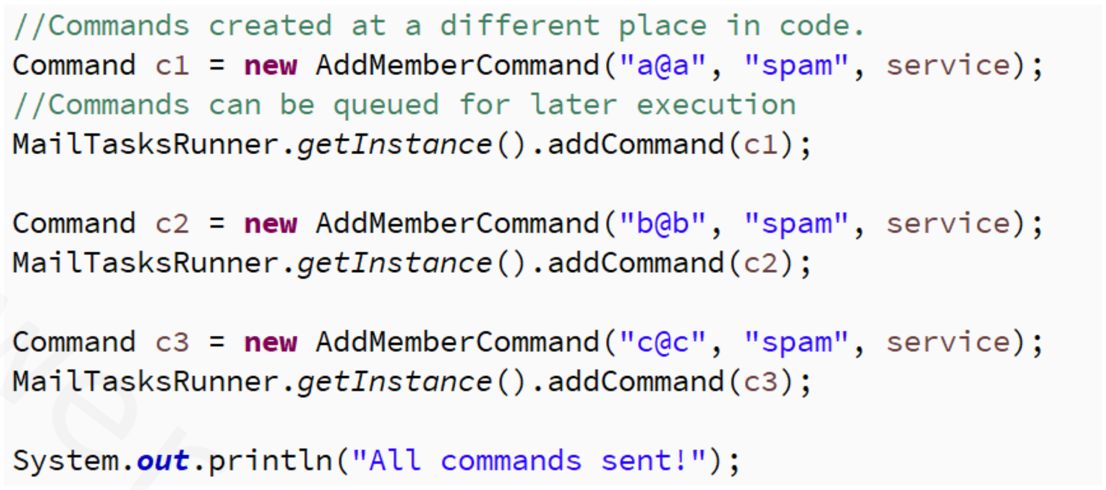
In-A-Hurry Summary

Invoker



```
//This is invoker actually executing commands.
//starts a worker thread in charge of executing commands
public class MailTasksRunner implements Runnable {

    private Thread runner;

    private List<Command> pendingCommands;

    private volatile boolean stop;

    private static final MailTasksRunner RUNNER = new MailT

    public static final MailTasksRunner getInstance() {

    private MailTasksRunner() {

    //Run method takes pending commands and executes them.
    public void run() {
```

Client



```
//Commands created at a different place in code.
Command c1 = new AddMemberCommand("a@a", "spam", service);
//Commands can be queued for later execution
MailTasksRunner.getInstance().addCommand(c1);

Command c2 = new AddMemberCommand("b@b", "spam", service);
MailTasksRunner.getInstance().addCommand(c2);

Command c3 = new AddMemberCommand("c@c", "spam", service);
MailTasksRunner.getInstance().addCommand(c3);

System.out.println("All commands sent!");
```

Example of Command Pattern

- The java.lang.Runnable interface represents the Command pattern.
  - We create the object of class implementing runnable, providing all information it needs.
  - In the run method we'll call an operation on the receiver.
  - We can send this object for later execution to other parts of our application.
- The Action class in struts framework is also an example of Command pattern. Here each URL is mapped to a action class. We also configure the exact no-arg method in that class which is called to process that request.

Interpreter

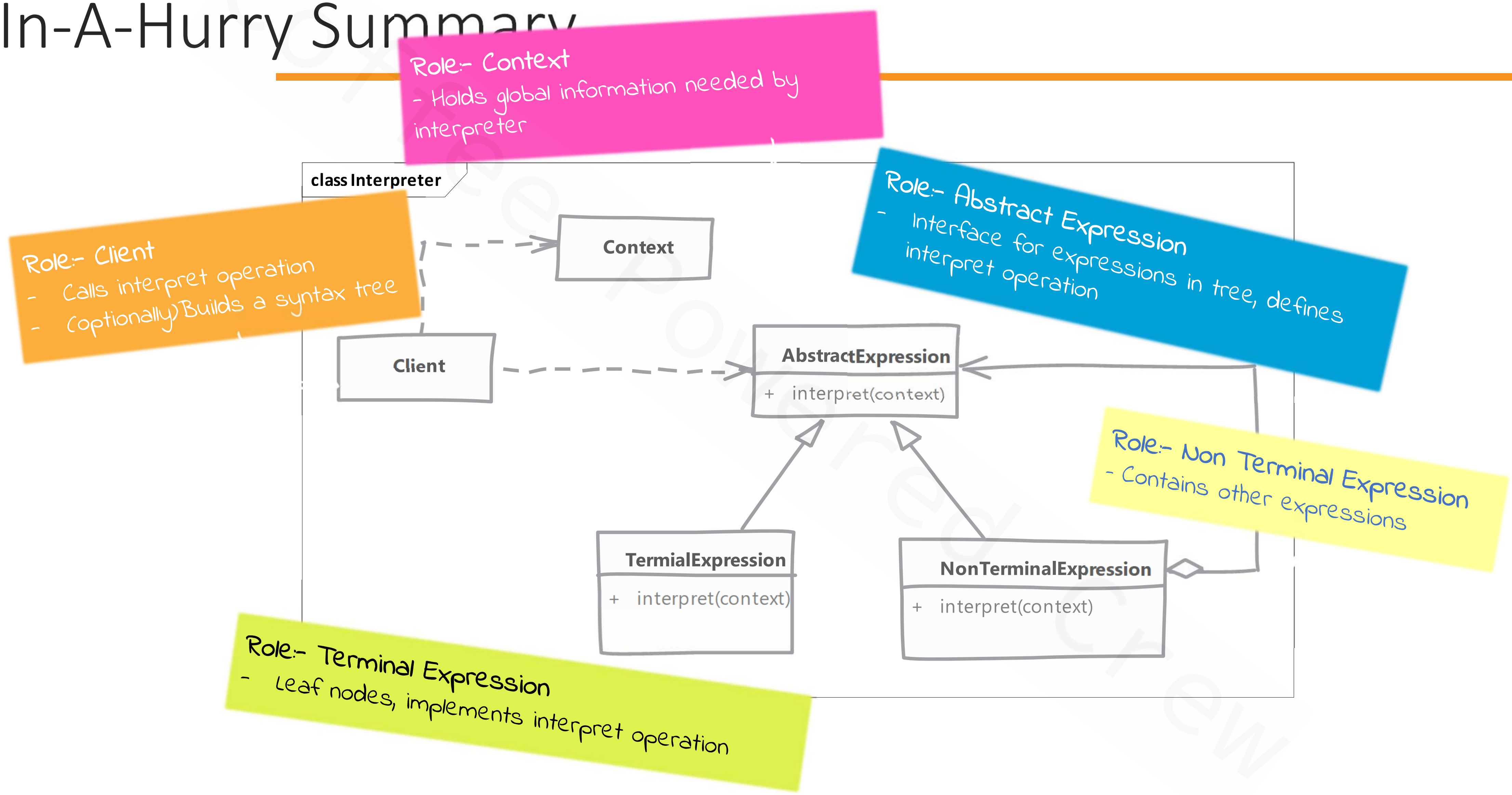
Behavioral Design Patterns

Design Patterns in Java

In-A-Hurry Summary

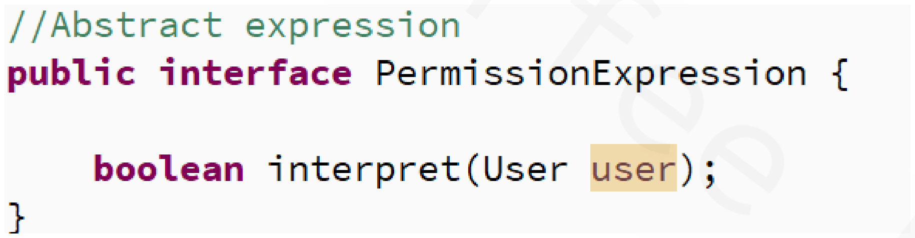
- When we want to parse a language with rules we can use the interpreter pattern.
- Each rule in the language becomes an expression class in the interpreter pattern. A terminal expression provides implementation of interpret method. A non-terminal expression holds other expressions and calls interpret on its children.
- This pattern doesn’t provide any solution for actual parsing and building of the abstract syntax tree. We have to do it outside this pattern.

In-A-Hurry Summary


Role:- Context
- Holds global information needed by interpreter
class Interpreter
Role:- Client
- Calls interpret operation
- (optionally) Builds a syntax tree
Context
Role:- Abstract Expression
- Interface for expressions in tree, defines interpret operation
Client
AbstractExpression
+ interpret(context)
Role:- Non Terminal Expression
- Contains other expressions
TermialExpression
+ interpret(context)
NonTerminalExpression
+ interpret(context)
Role:- Terminal Expression
- Leaf nodes, implements interpret operation

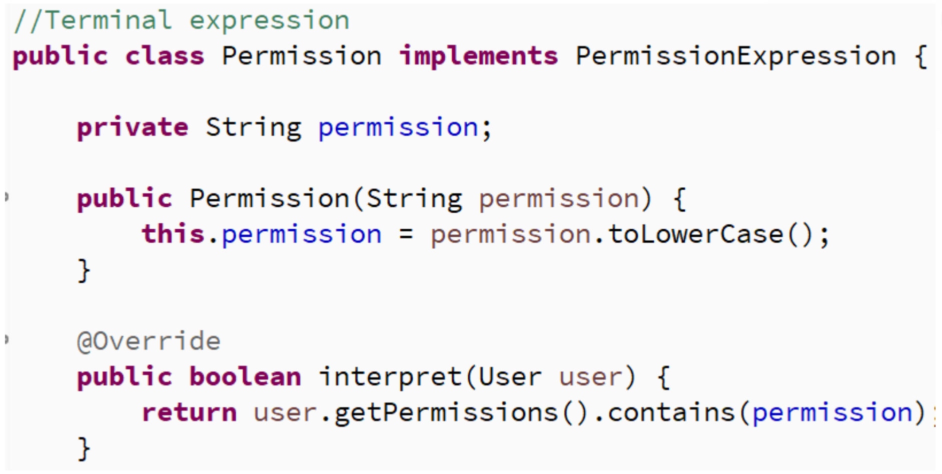
In-A-Hurry Summary

Abstract Expression



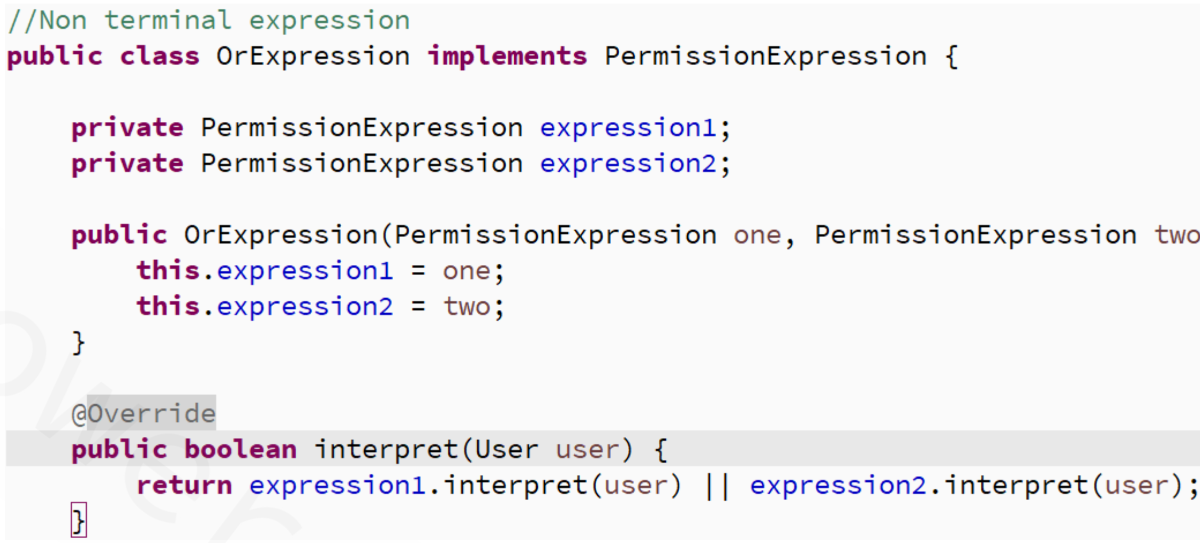
```
//Abstract expression
public interface PermissionExpression {

    boolean interpret(User user);
}
```

Terminal Expression



```
//Terminal expression
public class Permission implements PermissionExpression {

    private String permission;

    public Permission(String permission) {
        this.permission = permission.toLowerCase();
    }

    @Override
    public boolean interpret(User user) {
        return user.getPermissions().contains(permission);
    }
```

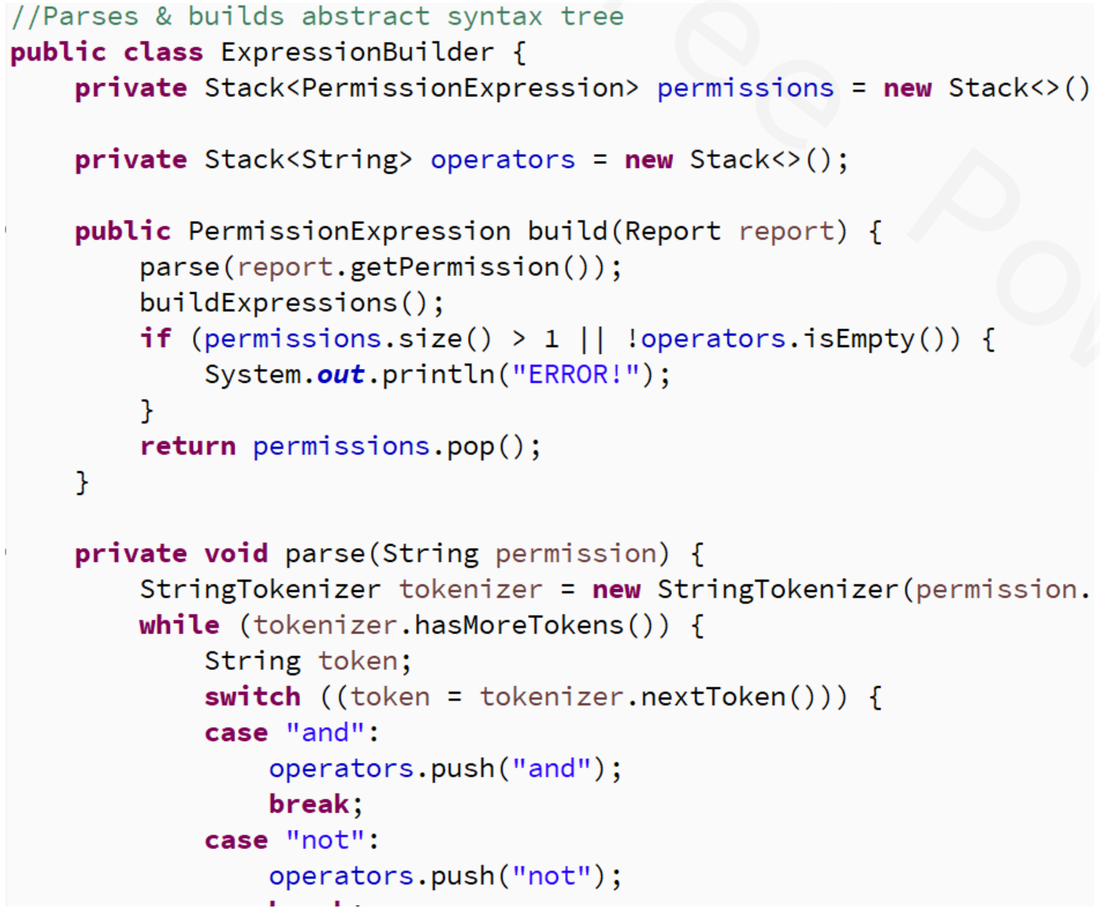
Non-terminal Expression



```
//Non terminal expression
public class OrExpression implements PermissionExpression {

    private PermissionExpression expression1;
    private PermissionExpression expression2;

    public OrExpression(PermissionExpression one, PermissionExpression two
        this.expression1 = one;
        this.expression2 = two;
    }

    @Override
    public boolean interpret(User user) {
        return expression1.interpret(user) || expression2.interpret(user);
    }
```

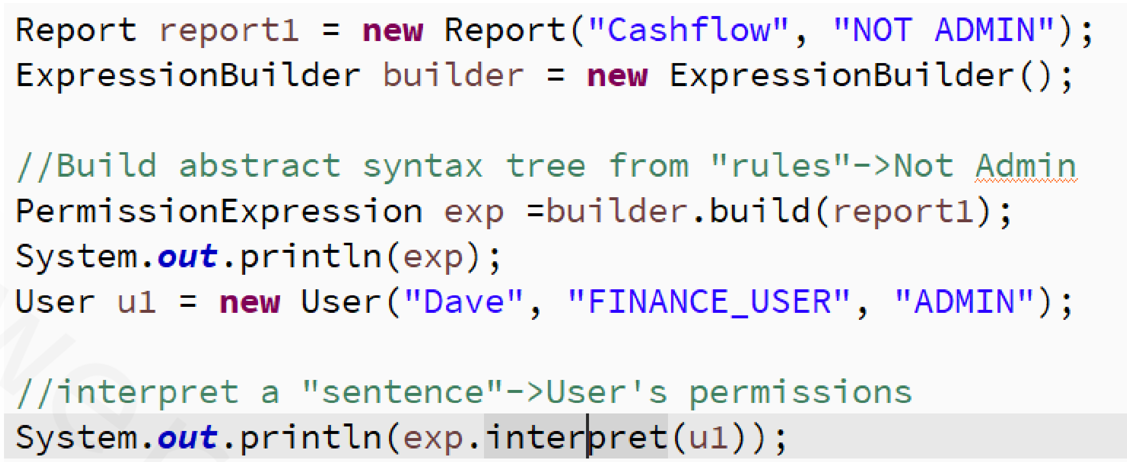
In-A-Hurry Summary

Expression Builder



```
//Parses & builds abstract syntax tree
public class ExpressionBuilder {
    private Stack<PermissionExpression> permissions = new Stack<>()

    private Stack<String> operators = new Stack<>();

    public PermissionExpression build(Report report) {
        parse(report.getPermission());
        buildExpressions();
        if (permissions.size() > 1 || !operators.isEmpty()) {
            System.out.println("ERROR!");
        }
        return permissions.pop();
    }

    private void parse(String permission) {
        StringTokenizer tokenizer = new StringTokenizer(permission.
        while (tokenizer.hasMoreTokens()) {
            String token;
            switch ((token = tokenizer.nextToken())) {
            case "and":
                operators.push("and");
                break;
            case "not":
                operators.push("not");
```

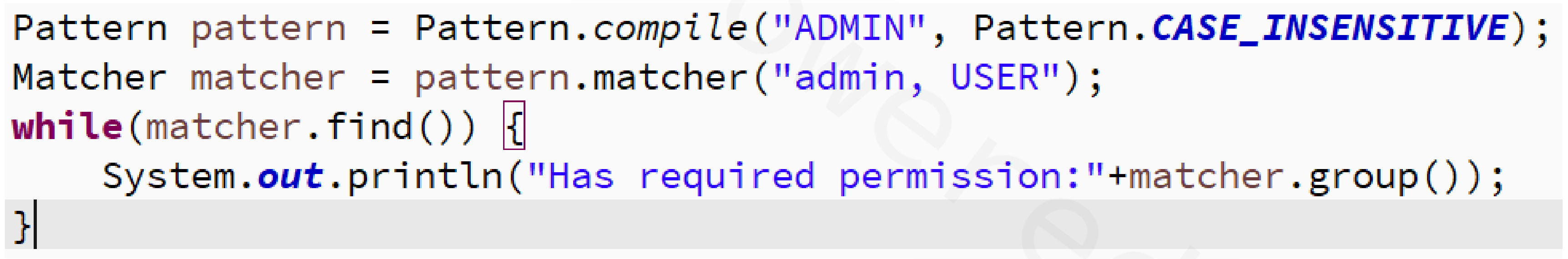
Client



```
Report report1 = new Report("Cashflow", "NOT ADMIN");
ExpressionBuilder builder = new ExpressionBuilder();

//Build abstract syntax tree from "rules"->Not Admin
PermissionExpression exp =builder.build(report1);
System.out.println(exp);
User u1 = new User("Dave", "FINANCE_USER", "ADMIN");

//interpret a "sentence"->User's permissions
System.out.println(exp.interpret(u1));
```

Examples of Interpreter

- The java.util.regex.Pattern class is an example of interpreter pattern in Java class library.
- Pattern instance is created with an internal abstract syntax tree, representing the grammar rules, during the static method call compile(). After that we check a sentence against this grammar using Matcher.



```
Pattern pattern = Pattern.compile("ADMIN", Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher("admin, USER");
while(matcher.find()) {
    System.out.println("Has required permission:"+matcher.group());
}
```

- Classes supporting the Unified Expression Language (EL) in JSP 2.1 JSF 1.2. These classes are in javax.el package. We have javax.el.Expression as a base class for value or method based expressions. We have javax.el.ExpressionFactory implementations to create the expressions. javax.el.ELResolver and its child classes complete the interpreter implementation.

Mediator

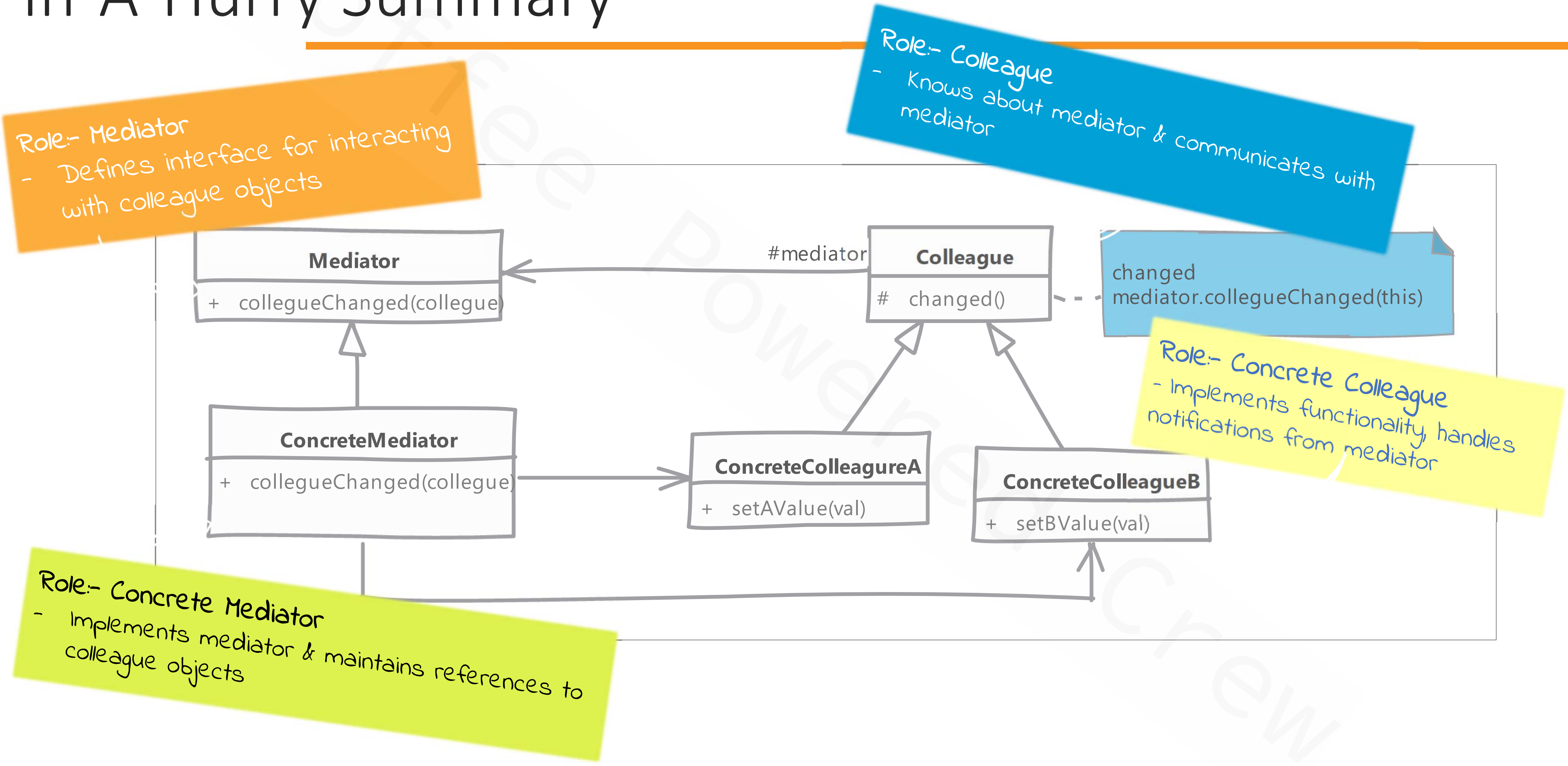
Behavioral Design Patterns

Design Patterns in Java

In-A-Hurry Summary

- When we want to decouple a group of objects which communicate with each other then we can use the mediator design pattern.
- Each object only knows about the mediator object and notifies it about change in it’s state. Mediator in turn will notify other objects on its behalf.
- Mediators are typically specific to a collaboration. It’s difficult to write a reusable mediator. Observer design pattern solves this problem. However mediators are easy to implement and extend.

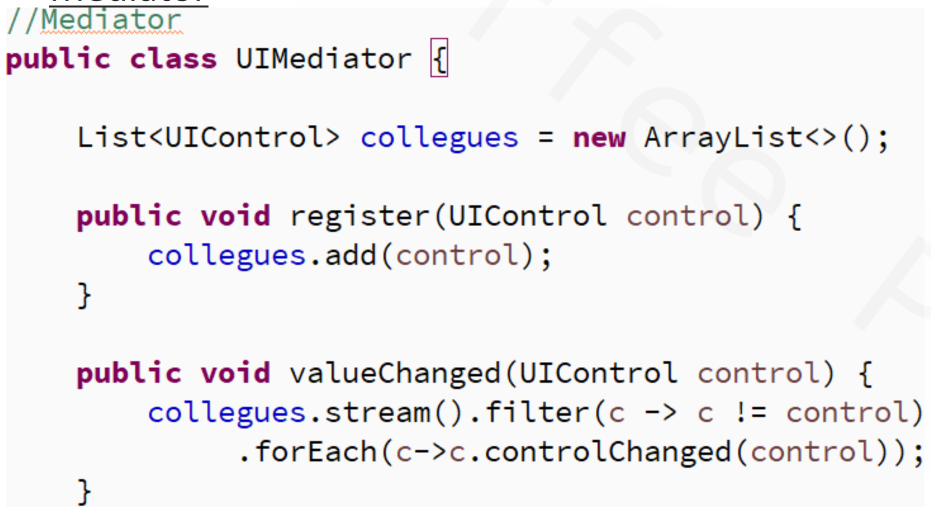
In-A-Hurry Summary



Role:- Mediator
- Defines interface for interacting with colleague objects
Role:- Colleague
- Knows about mediator & communicates with mediator
Mediator
+ collegueChanged(collegue)
#mediator
Colleague
# changed()
changed
mediator.collegueChanged(this)
Role:- Concrete Colleague
- Implements functionality, handles notifications from mediator
ConcreteMediator
+ collegueChanged(collegue)
ConcreteColleagureA
+ setAValue(val)
ConcreteColleagueB
+ setBValue(val)
Role:- Concrete Mediator
- Implements mediator & maintains references to colleague objects

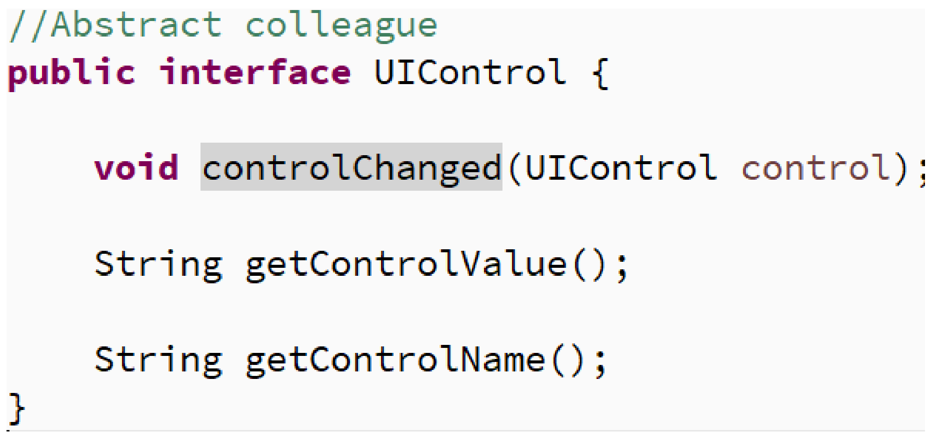
In-A-Hurry Summary

Mediator



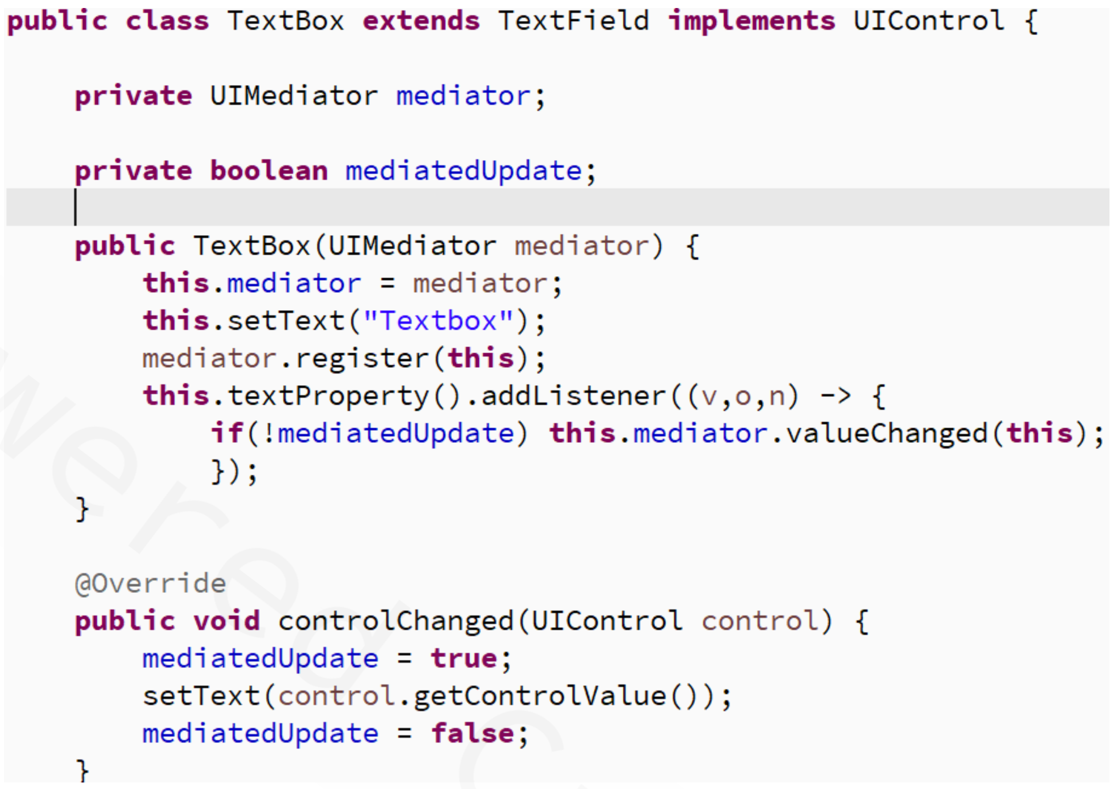
```
//Mediator
public class UIMediator {

    List<UIControl> collegues = new ArrayList<>();

    public void register(UIControl control) {
        collegues.add(control);
    }

    public void valueChanged(UIControl control) {
        collegues.stream().filter(c -> c != control)
             .forEach(c->c.controlChanged(control));
    }
```

Colleague



```
//Abstract colleague
public interface UIControl {

    void controlChanged(UIControl control);

    String getControlValue();

    String getControlName();
}
```

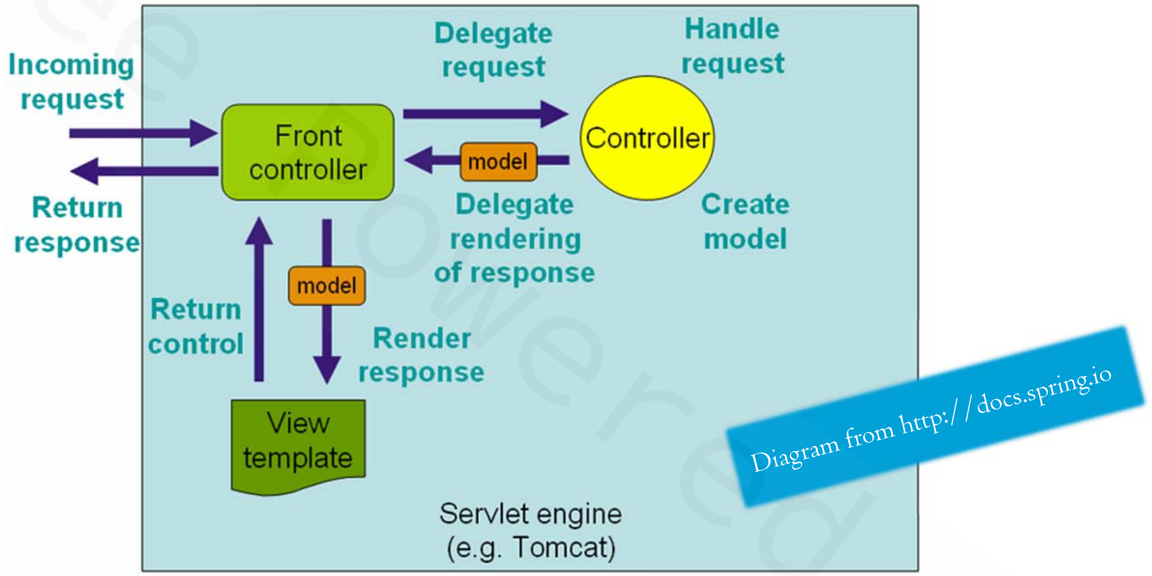
Concrete Colleague



```
public class TextBox extends TextField implements UIControl {

    private UIMediator mediator;

    private boolean mediatedUpdate;

    public TextBox(UIMediator mediator) {
        this.mediator = mediator;
        this.setText("Textbox");
        mediator.register(this);
        this.textProperty().addListener((v,o,n) -> {
            if(!mediatedUpdate) this.mediator.valueChanged(this);
            });
    }

    @Override
    public void controlChanged(UIControl control) {
        mediatedUpdate = true;
        setText(control.getControlValue());
        mediatedUpdate = false;
    }
```

Examples of Mediator

- Sometimes a front controller is given as an example of mediator pattern. E.g. The DispatcherServlet in Spring.



- Purpose of front controller is to act as a central point where requests from outside world can land and then they are forwarded to appropriate page controller, often by use of some form of URL to class mapping.
- Front controller pattern can be thought of as a specialized version of mediator pattern. Front controller satisfies mediator characteristics like acting as central hub for communication between objects. It is specialized since it also handles requests from outside system & performs lookup to find a specific controller which will handle the request. In mediator when one object changes *all* others are notified!

Iterator

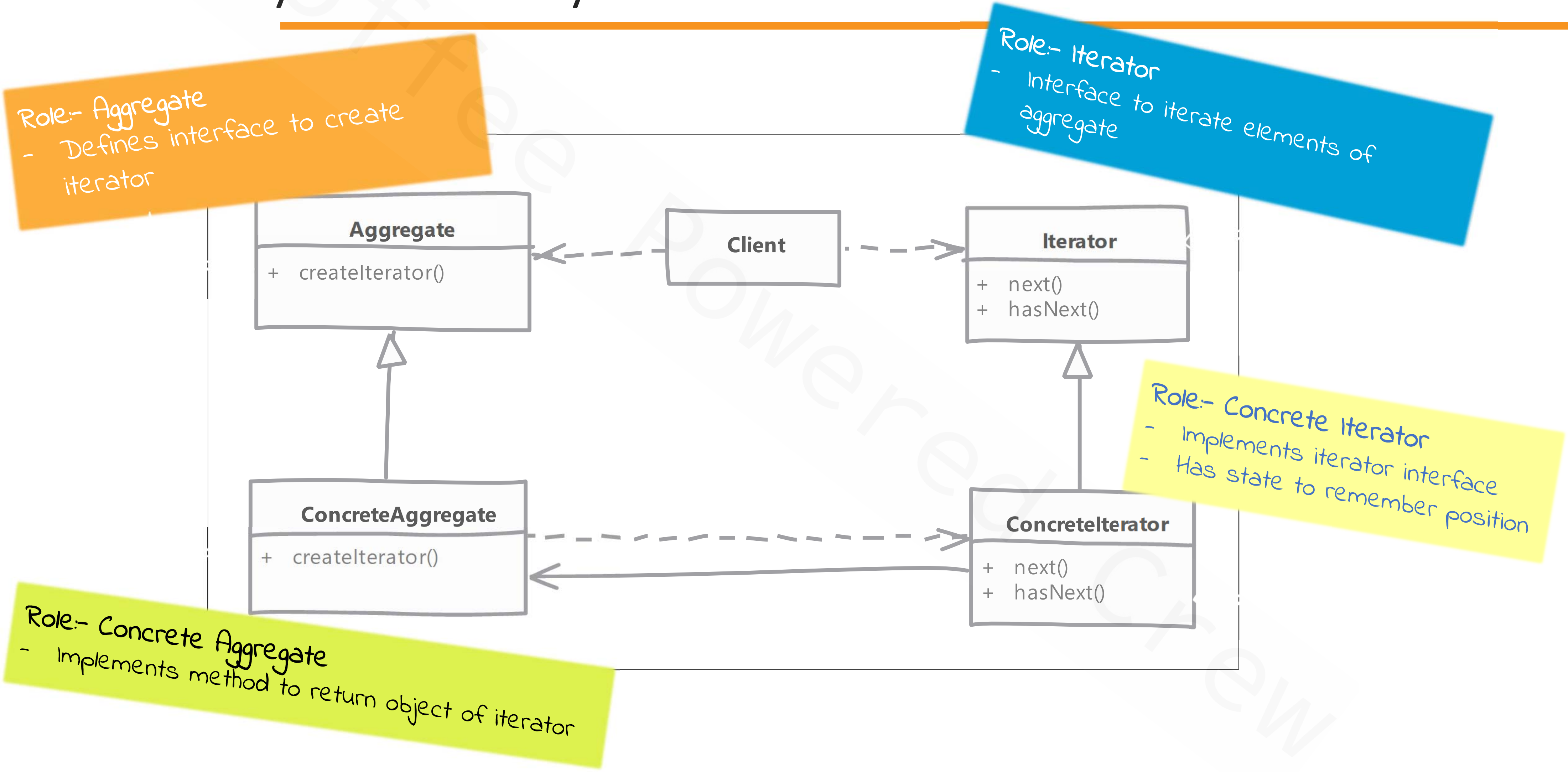
Behavioral Design Patterns

Design Patterns in Java

In-A-Hurry Summary

- When we want to iterate or give sequential access to elements of aggregate object we can use iterator design pattern.
- Iterator needs access to internal data structure of aggregator to provide its functionality. This usually means it’s quite common to have iterator implemented as inner class.
- Iterator allows the client code to check whether there is an element available to consume and give next available element.
- We can also provide reverse, or bi-directional (forward + backward) iterator depending on underlying data structure.

In-A-Hurry Summary


Role:- Aggregate
- Defines interface to create iterator
Role:- Iterator
- Interface to iterate elements of aggregate
Aggregate
+ createIterator()
Client
Iterator
+ next()
+ hasNext()
Role:- Concrete Iterator
- Implements iterator interface
- Has state to remember position
ConcreteAggregate
+ createIterator()
ConcreteIterator
+ next()
+ hasNext()
Role:- Concrete Aggregate
- Implements method to return object of iterator

In-A-Hurry Summary

Iterator

```java
//Iterator interface allowing to iterate over
//values of an aggregate
public interface Iterator<T> {

    boolean hasNext();

    T next();
}
```

Aggregate

```java
//This enum represents the aggregate from iterator pattern
public enum ThemeColor {

    RED,
    ORANGE,
    BLACK,
    WHITE;

    public static Iterator<ThemeColor> getIterator() {
        return new ThemeColorIterator();
    }
```

Concrete Iterator

```java
//This is the concrete iterator. Note that it has a state
private static class ThemeColorIterator implements Iterator<ThemeColor>
    //current position
    private int position;

    @Override
    public boolean hasNext() {
        return position < ThemeColor.values().length;
    }

    @Override
    public ThemeColor next() {
        return ThemeColor.values()[position++];
    }
```

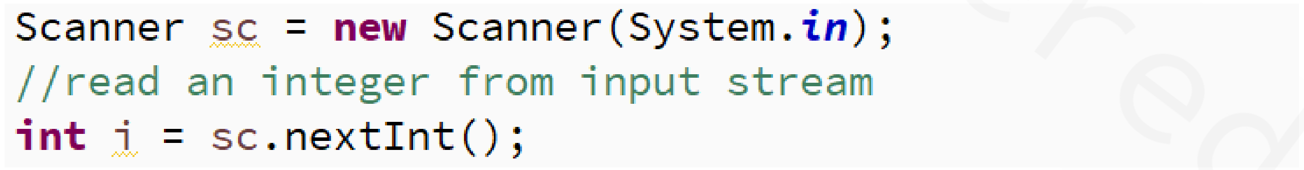
Client

```java
Iterator<ThemeColor> iter = ThemeColor.getIterator();

while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

Examples of Iterator

- Yup! The iterator classes in Java's collection framework are great examples of iterator. ☺ The concrete iterators are typically inner classes in each collection class implementing java.util.Iterator interface.

- The java.util.Scanner class is also an example of Iterator pattern. This class supports InputStream as well and allows to iterate over a stream.



```java
Scanner sc = new Scanner(System.in);
//read an integer from input stream
int i = sc.nextInt();
```

- The javax.xml.stream.XMLEventReader class is also an iterator. This class turns the XML into a stream of event objects.

Memento

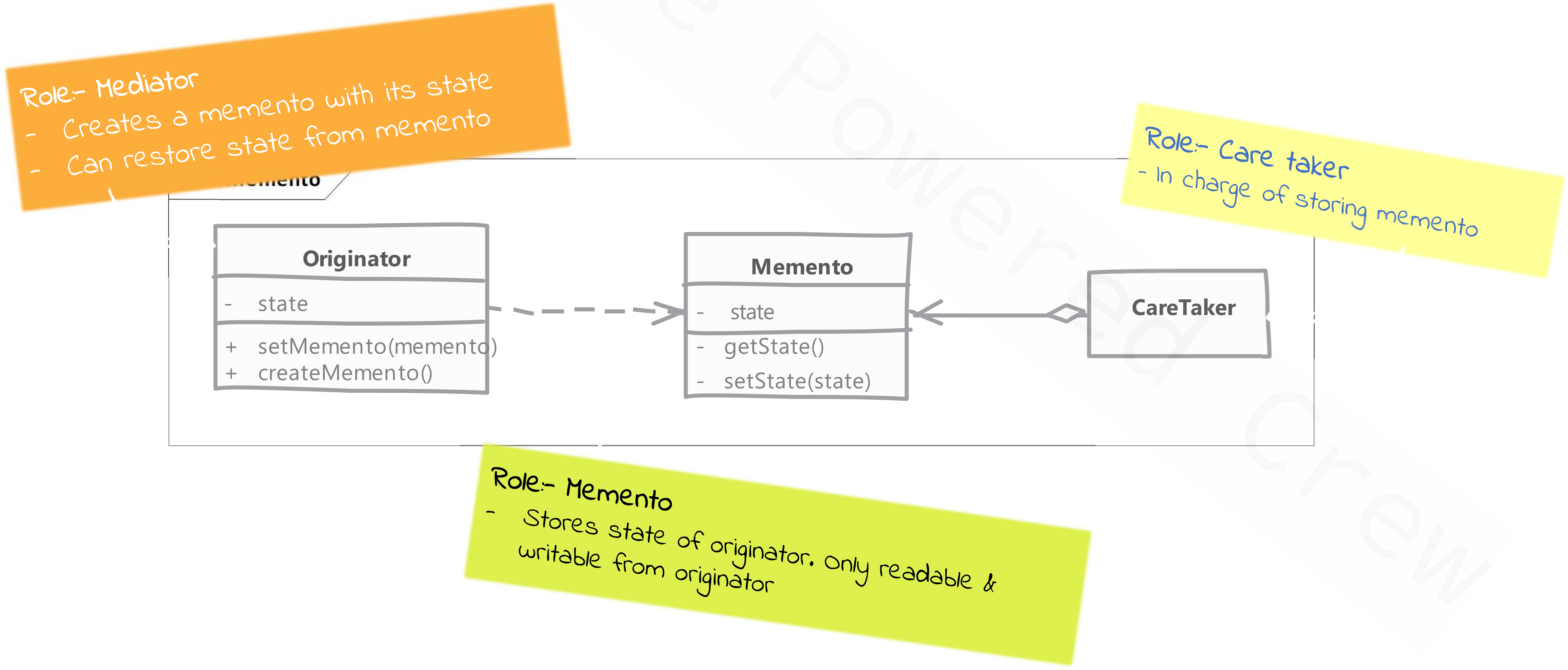
Behavioral Design Patterns

Design Patterns in Java

In-A-Hurry Summary

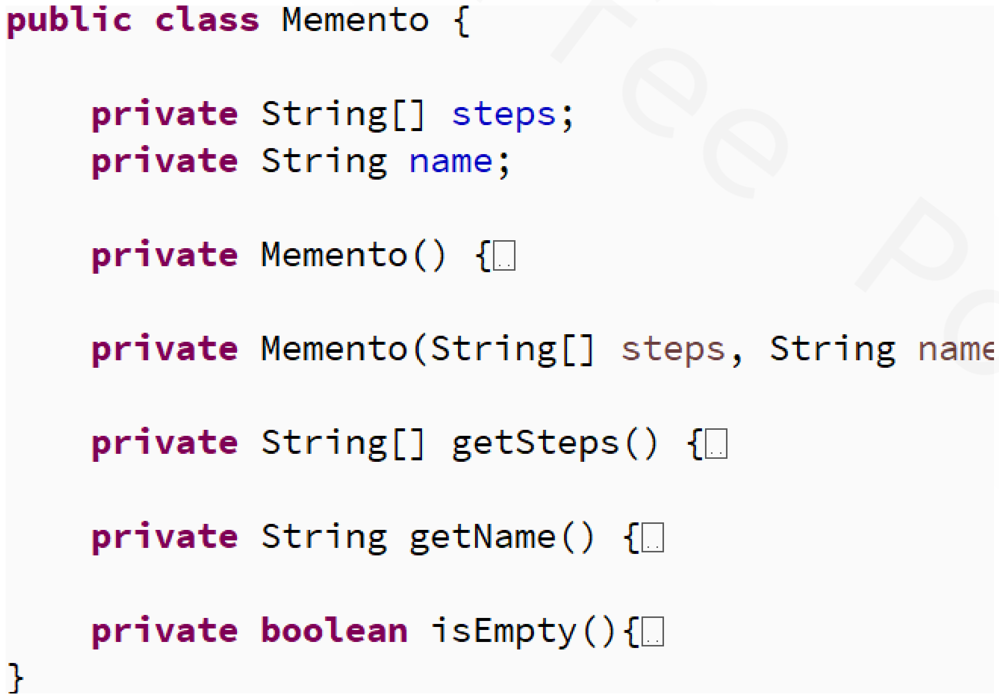
- We can use memento design pattern to take a snapshot of object’s state which can be then used to restore object to that particular state.
- Memento itself is created such that it doesn’t expose any state stored in it to any other class aside from the originator.
- Originator provides a method to get a memento out of it. And another method to assign it a memento, which results in getting the originator’s state reset to the one in memento.
- Mementos need to be saved for them to be of any use. Originator can save them but it adds complexity.
- Memento works well with command pattern. Each commands saves a memento as part of execution.

In-A-Hurry Summary


Role:- Mediator
- Creates a memento with its state
- Can restore state from memento
Originator
- state
+ setMemento(memento)
+ createMemento()
Memento
- state
- getState()
- setState(state)
CareTaker
Role:- Care taker
- In charge of storing memento
Role:- Memento
- Stores state of originator. only readable & writable from originator

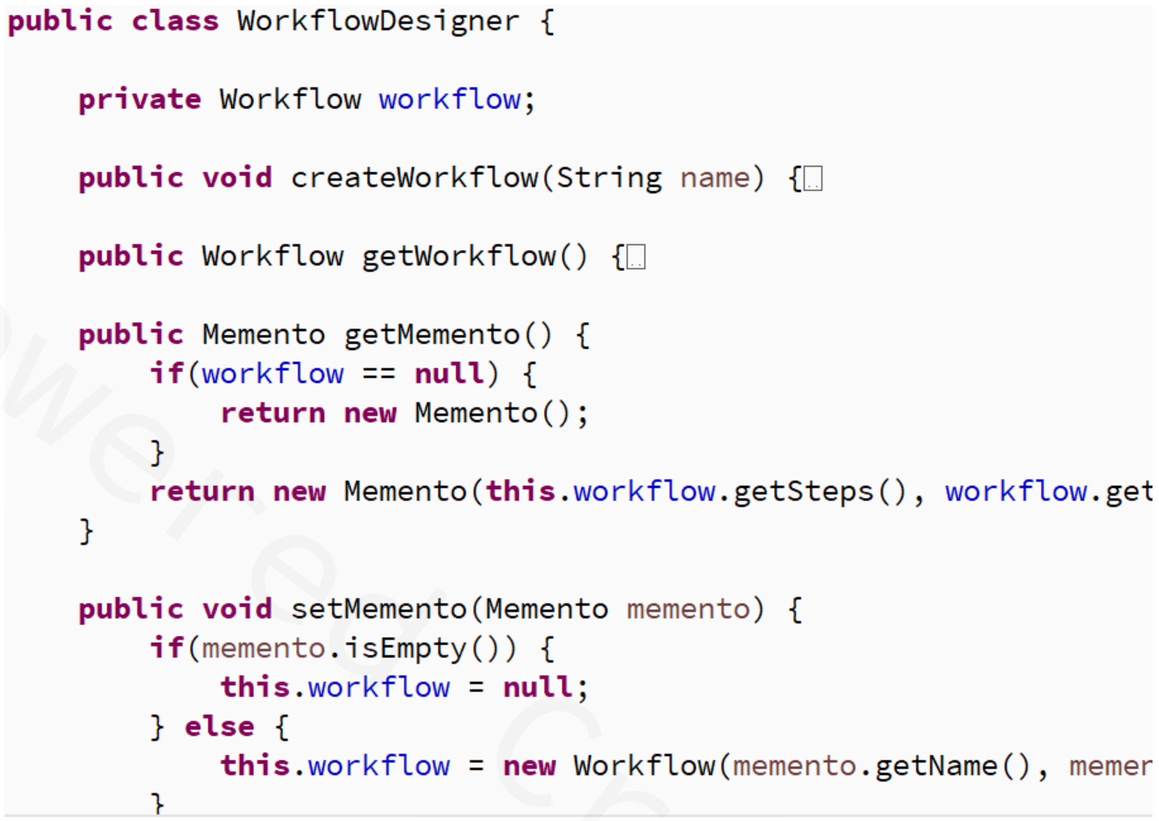
In-A-Hurry Summary

Memento



```
public class Memento {

    private String[] steps;
    private String name;

    private Memento() {□

    private Memento(String[] steps, String name

    private String[] getSteps() {□

    private String getName() {□

    private boolean isEmpty(){□
}
```

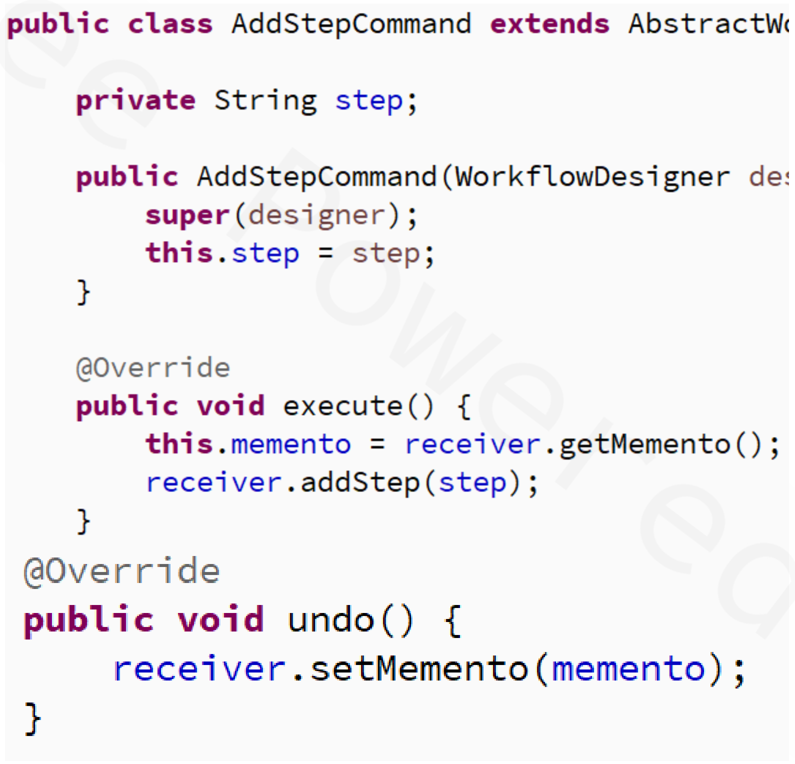
Originator



```
public class WorkflowDesigner {

    private Workflow workflow;

    public void createWorkflow(String name) {□

    public Workflow getWorkflow() {□

    public Memento getMemento() {
        if(workflow == null) {
            return new Memento();
        }
        return new Memento(this.workflow.getSteps(), workflow.get
    }

    public void setMemento(Memento memento) {
        if(memento.isEmpty()) {
            this.workflow = null;
        } else {
            this.workflow = new Workflow(memento.getName(), memer
        }
```

In-A-Hurry Summary

Caretaker



```
public class AddStepCommand extends AbstractW

    private String step;

    public AddStepCommand(WorkflowDesigner de
        super(designer);
        this.step = step;
    }

    @Override
    public void execute() {
        this.memento = receiver.getMemento();
        receiver.addStep(step);
    }
@Override
public void undo() {
    receiver.setMemento(memento);
}
```

Example of a Memento

- One great example of memento is the undo support provided by the javax.swing.text.JTextComponent and its child classes like JTextField, JTextArea etc.
- The javax.swing.undo.UndoManager acts as the caretaker & implementations of javax.swing.undo.UndoableEdit interface work as mementos. The javax.swing.text.Document implementation which is model for text components in swing is the originator.

```
//Create & attach UndoManager to a text field
UndoManager undoManager = new UndoManager();
JTextField text = new JTextField(10);
text.getDocument().addUndoableEditListener(undoManager);

//After some edits
if(undoManager.canUndo()) {
    undoManager.undo();
}
```

- The java.io.Serializable is often given as an example of Memento but it is NOT a memento & not even a design pattern. Memento object itself can be serialized but it is NOT mandatory requirement of the pattern. In fact mementos work most of the times with in-memory snapshots of state.

Observer

Behavioral Design Patterns

Design Patterns in Java

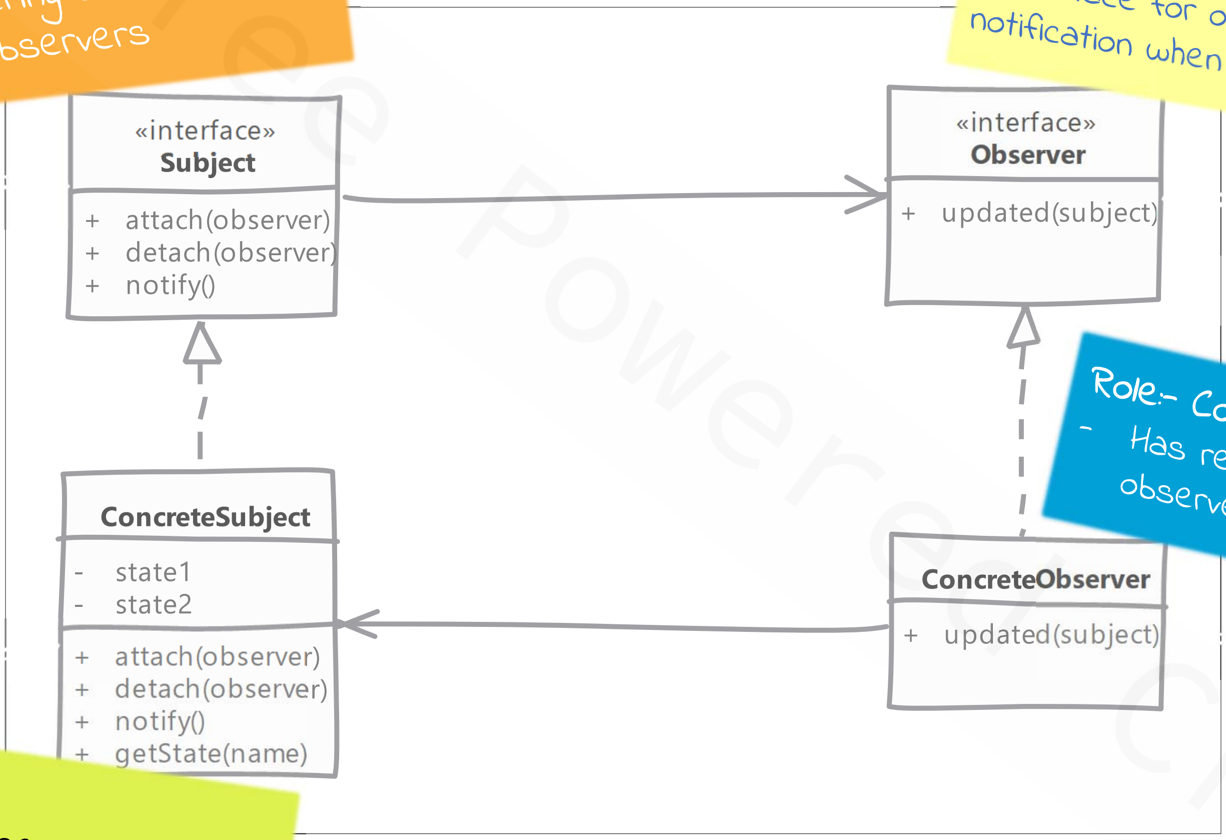
In-A-Hurry Summary

- Observer pattern allows to define one-to-many dependency between objects where many objects are interested in state change of a object.
- Observers register themselves with the subject which then notifies all registered observers if any state change occurs.
- In the notification sent to observers it is common to only send reference of subject instead of state values. Observers will call the subject back for more information if needed.
- We can also register observers for a specific event only, resulting in improved performance of sending notifications in the subject.
- This design pattern is also known as publisher-subscriber pattern. Java messaging uses this pattern but instead of registering with subject, listeners register with a JMS broker, which acts as a middleman.

In-A-Hurry Summary

Role:- Subject
- Interface for registering observers
- Supports multiple observers

Role:- observer
- Interface for objects that want notification when subject changes

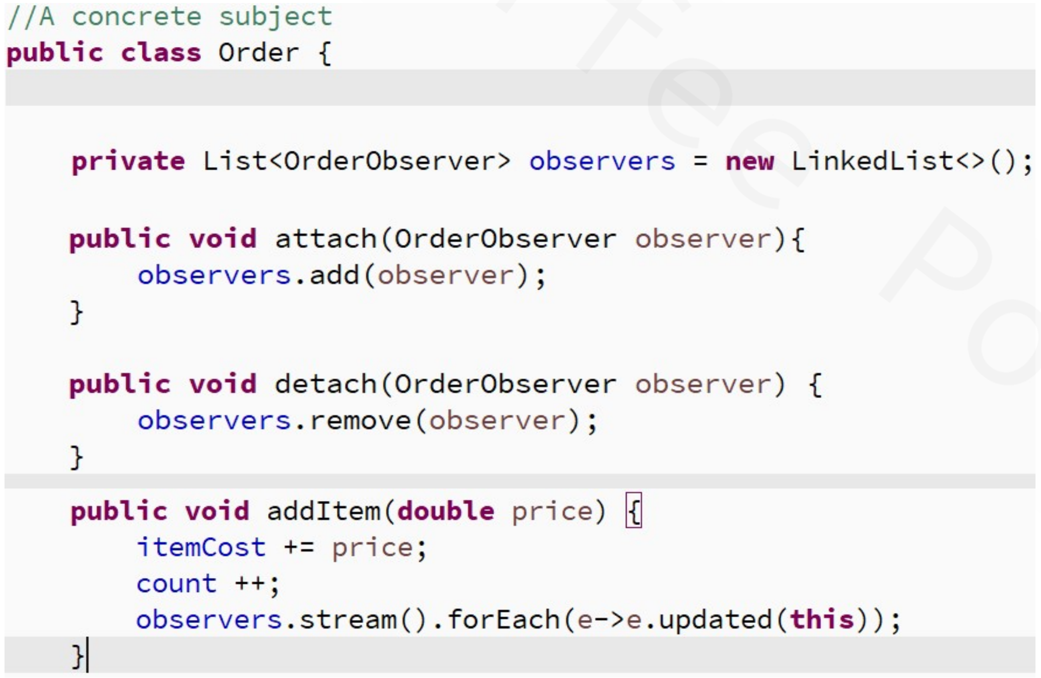


Role:- Concrete observer
- Has reference to concrete observer

Role:- Concrete Subject
- Sends notification to observers when its state changes

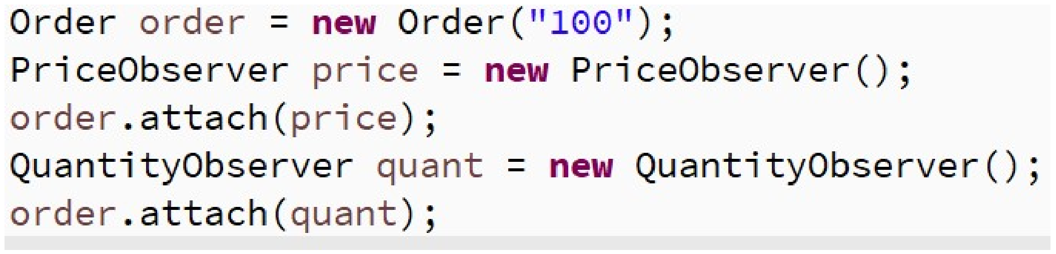
In-A-Hurry Summary

Subject



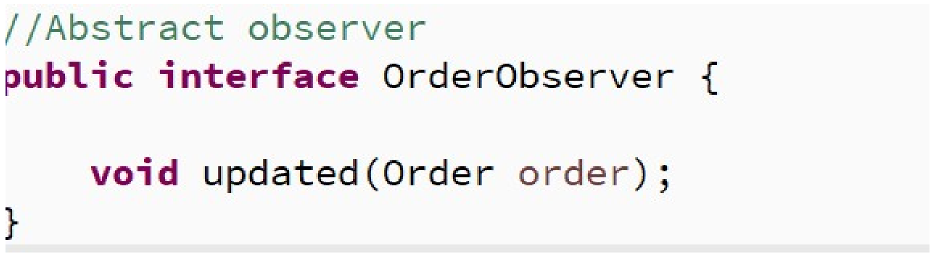
```java
//A concrete subject
public class Order {


    private List<OrderObserver> observers = new LinkedList<>();

    public void attach(OrderObserver observer){
        observers.add(observer);
    }

    public void detach(OrderObserver observer) {
        observers.remove(observer);
    }
    public void addItem(double price) {
        itemCost += price;
        count ++;
        observers.stream().forEach(e->e.updated(this));
    }
```

Attaching Observers to Subject



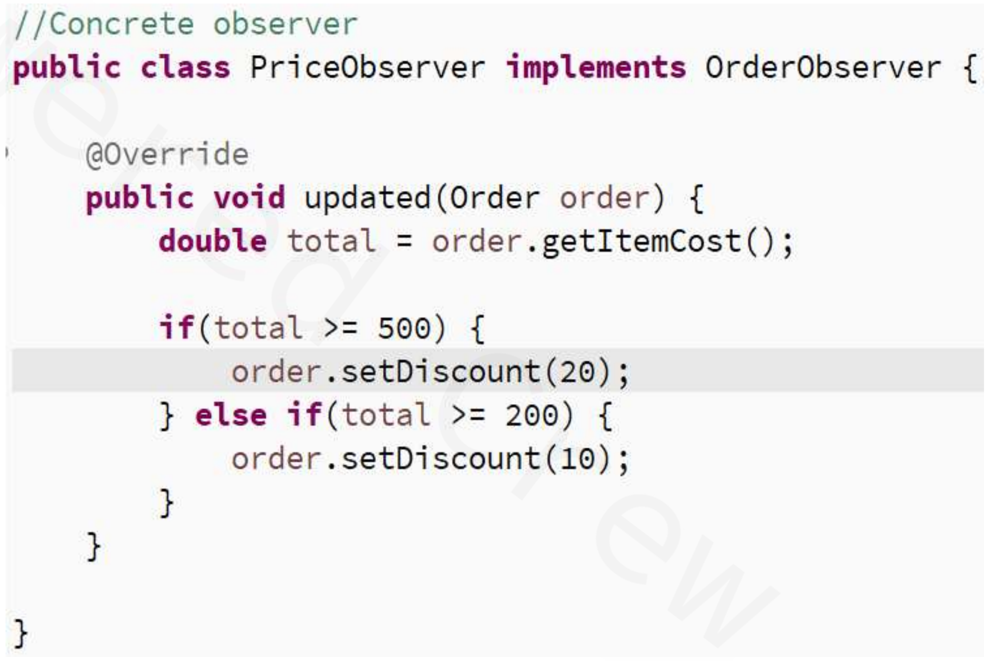
```java
Order order = new Order("100");
PriceObserver price = new PriceObserver();
order.attach(price);
QuantityObserver quant = new QuantityObserver();
order.attach(quant);
```

Observer



```java
//Abstract observer
public interface OrderObserver {

    void updated(Order order);
}
```

Concrete Observer



```java
//Concrete observer
public class PriceObserver implements OrderObserver {

    @Override
    public void updated(Order order) {
        double total = order.getItemCost();

        if(total >= 500) {
            order.setDiscount(20);
        } else if(total >= 200) {
            order.setDiscount(10);
        }
    }

}
```

Examples of Observer

- Observer is such a useful pattern that Java comes with support for this support in Java Class Library! We have java.util.Observer interface & java.util.Observable class shipped with JDK.

Although for some reason I haven't seen developers using these outside of Swing. ☺

- Another commonly used example is various listeners in Java Servlet application. We can create various listeners by implementing interfaces like HttpSessionListener, ServletRequestListener.
- We then register these listeners with ServletContext's addListener method. These listeners are notified when certain events occur like, creation of a request or addition of a value to the session.
- The notification will sent to observers based on the event that has taken place and the interface(s) implemented by registered observers.
- Spring also supports Observer through the org.springframework.context.ApplicationListener interface.

State

Behavioral Design Patterns

Design Patterns in Java

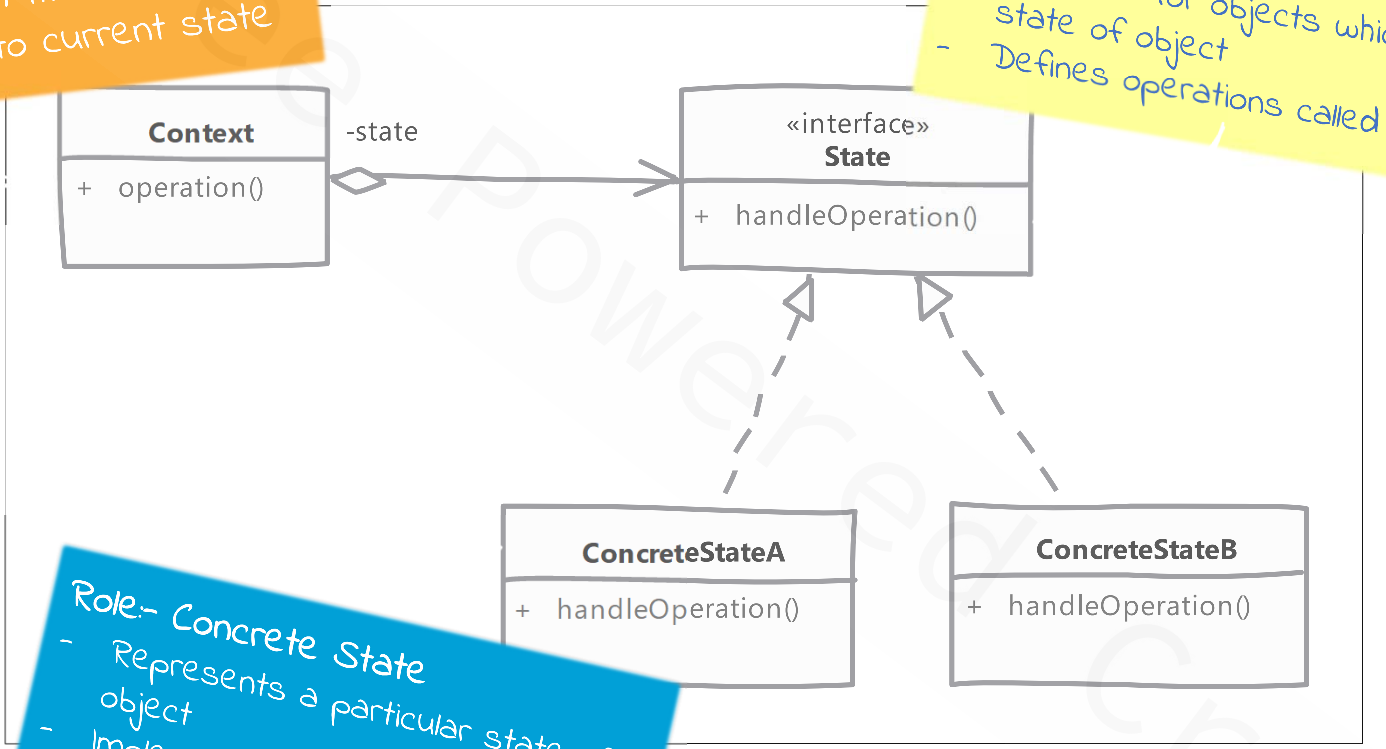
In-A-Hurry Summary

- If we have an object whose behavior is completely tied to its internal state which can be expressed as an object we can use the state pattern.
- Each possible state ***value*** now becomes a class providing behavior specific to a state value.
- Our main object (aka context) delegates the actual operation to its current state. States will implement behavior which is specific to a particular state value.
- Context object's state change is explicit now, since we change the entire state object.
- State transitions are handled either by states themselves or context can trigger them.
- We can reuse state objects if they don't have any instance variables and only provide behavior.

In-A-Hurry Summary

Role:- Context
- Class whose state is now an object!
- Client code works with this class
- Delegates operation to current state

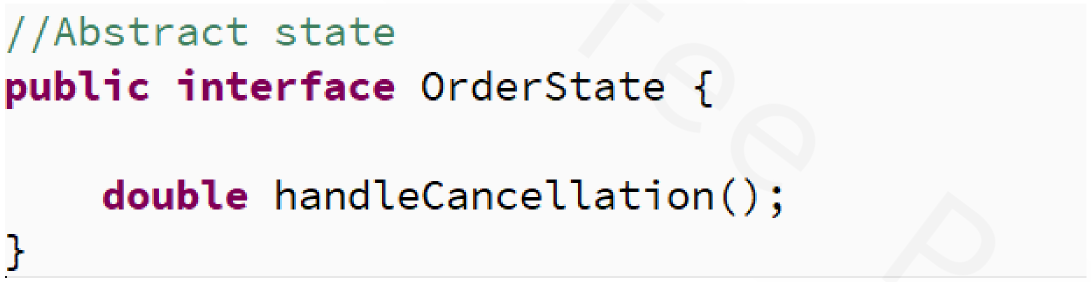
Role:- State
- Interface for objects which represents state of object
- Defines operations called by owning object



Role:- Concrete State
- Represents a particular state of object
- Implements behavior specific to this state value

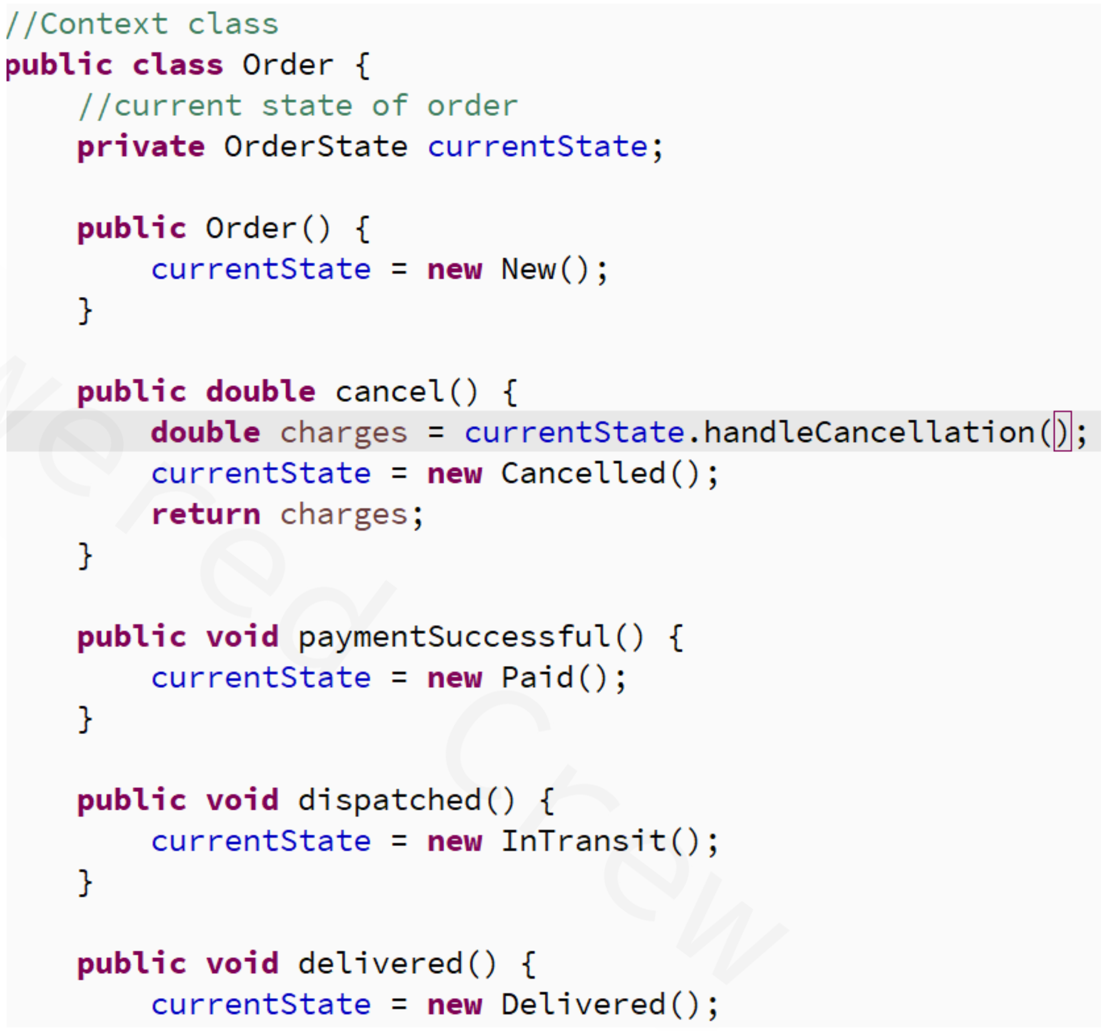
In-A-Hurry Summary

State



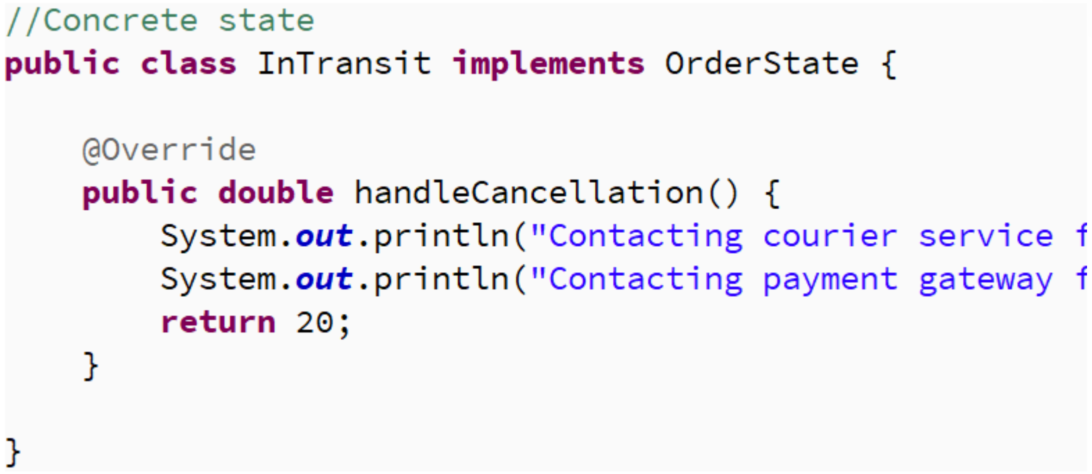
```
//Abstract state
public interface OrderState {

    double handleCancellation();
}
```

Context



```
//Context class
public class Order {
    //current state of order
    private OrderState currentState;

    public Order() {
        currentState = new New();
    }

    public double cancel() {
        double charges = currentState.handleCancellation();
        currentState = new Cancelled();
        return charges;
    }

    public void paymentSuccessful() {
        currentState = new Paid();
    }

    public void dispatched() {
        currentState = new InTransit();
    }

    public void delivered() {
        currentState = new Delivered();
```

Concrete State



```
//Concrete state
public class InTransit implements OrderState {

    @Override
    public double handleCancellation() {
        System.out.println("Contacting courier service f
        System.out.println("Contacting payment gateway f
        return 20;
    }

}
```

Examples of State pattern

- One of the examples of state pattern can be found in JSF(Java Server Faces) framework’s LifeCycle implementation.
- FacesServlet will invoke execute & render methods of LifeCycle. LifeCycle instance in turn collaborates with multiple “phases” to execute a JSF request. Here each phase represents a state in state pattern.
- JSF has six phases: RestoreViewPhase, ApplyRequestValues, ProcessValidationsPhase, UpdateModelValuesPhase, InvokeApplicationPhase, and RenderResponsePhase
- The LifeCycle also provides an object of FacesContext to these phases to help in providing state specific behavior.
- In case you are not familiar with JSF, then you can always fall back on the order processing example that we studied.

Strategy

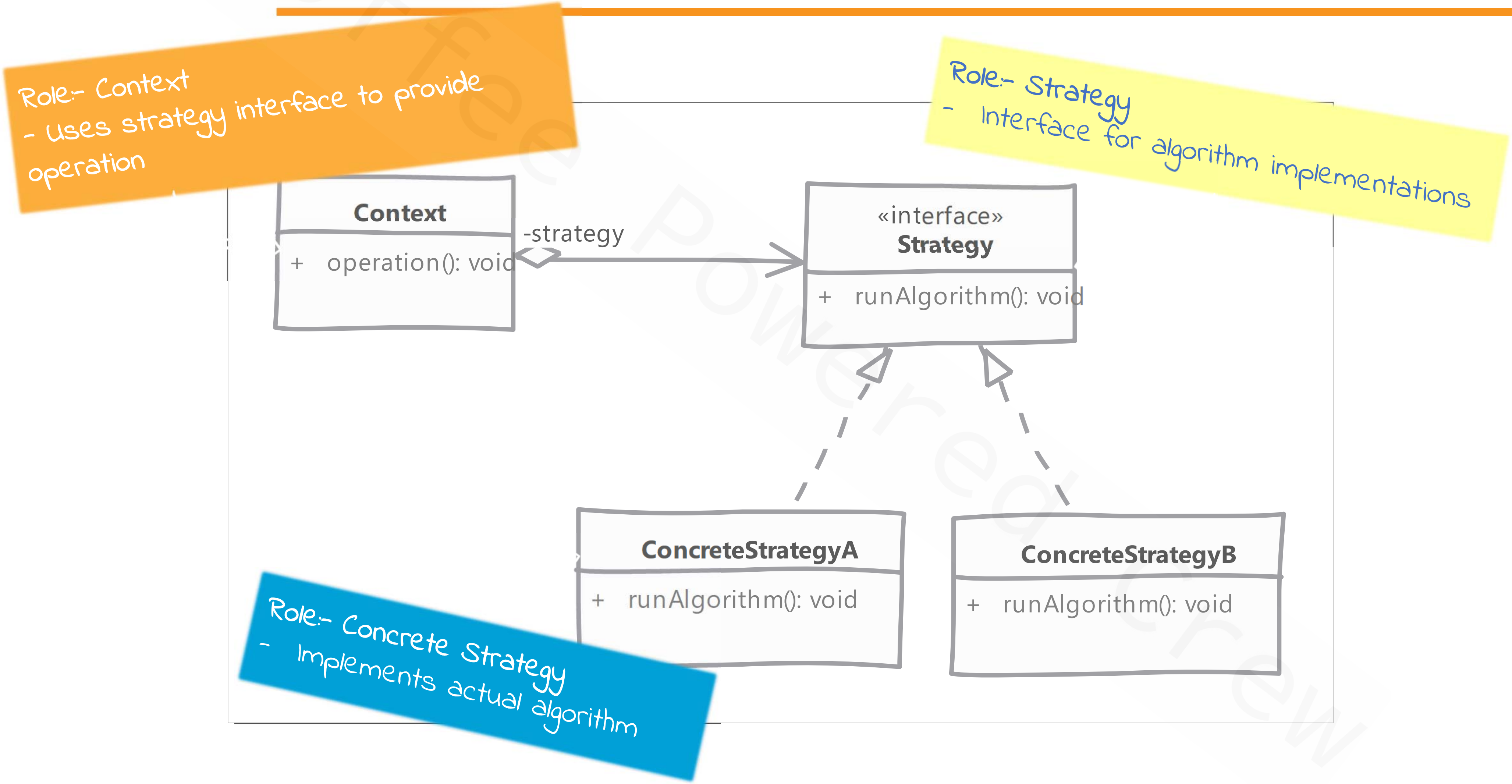
Behavioral Design Patterns

Design Patterns in Java

In-A-Hurry Summary

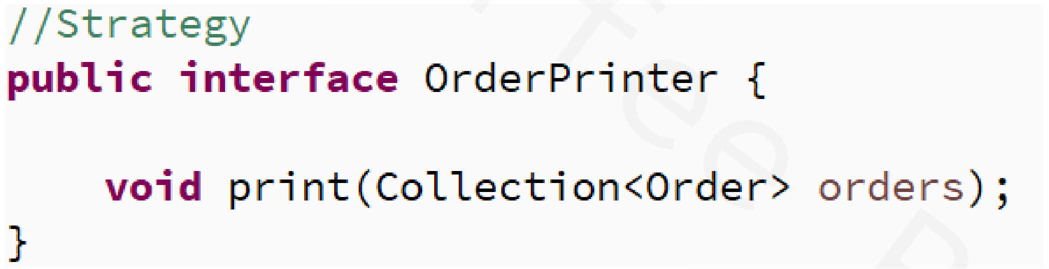
- Strategy pattern allows us to encapsulate algorithms in separate classes. The class using these algorithms (called context) can now be configured with desired implementation of an algorithm.
- It is typically the responsibility of client code which is using our context object to configure it.
- Strategy objects are given all data they need by the context object. We can pass data either in form of arguments or pass on context object itself.
- Strategy objects typically end up being stateless making them great candidates for flyweight pattern.
- Client code ends up knowing about all implementations of strategy since it has to create their objects.

In-A-Hurry Summary



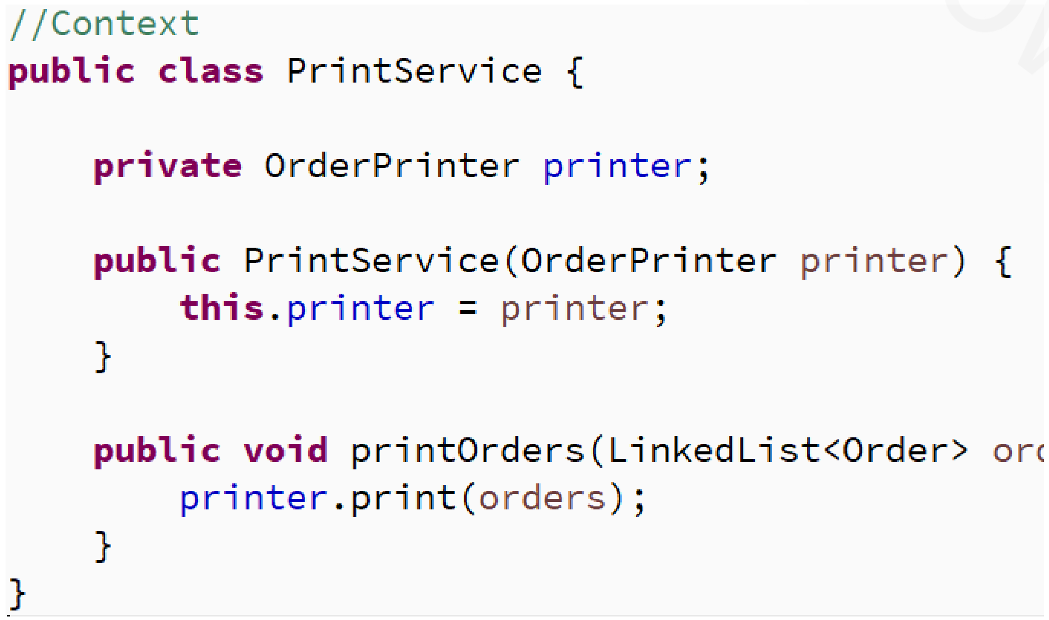
In-A-Hurry Summary

Strategy



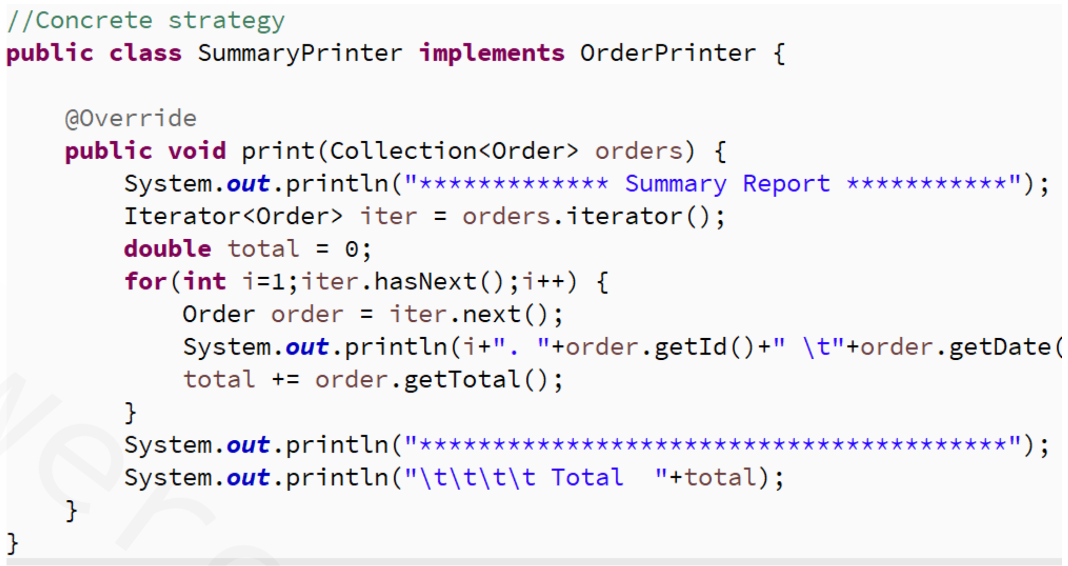
```
//Strategy
public interface OrderPrinter {

    void print(Collection<Order> orders);
}
```

Context



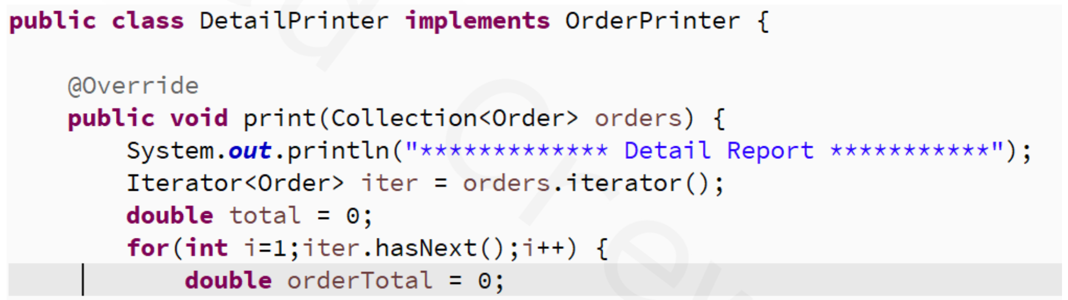
```
//Context
public class PrintService {

    private OrderPrinter printer;

    public PrintService(OrderPrinter printer) {
        this.printer = printer;
    }

    public void printOrders(LinkedList<Order> orc
        printer.print(orders);
    }
}
```

Concrete Strategy



```
//Concrete strategy
public class SummaryPrinter implements OrderPrinter {

    @Override
    public void print(Collection<Order> orders) {
        System.out.println("************* Summary Report ***********");
        Iterator<Order> iter = orders.iterator();
        double total = 0;
        for(int i=1;iter.hasNext();i++) {
            Order order = iter.next();
            System.out.println(i+". "+order.getId()+" \t"+order.getDate(
            total += order.getTotal();
        }
        System.out.println("****************************************");
        System.out.println("\t\t\t\t Total  "+total);
    }
}
```

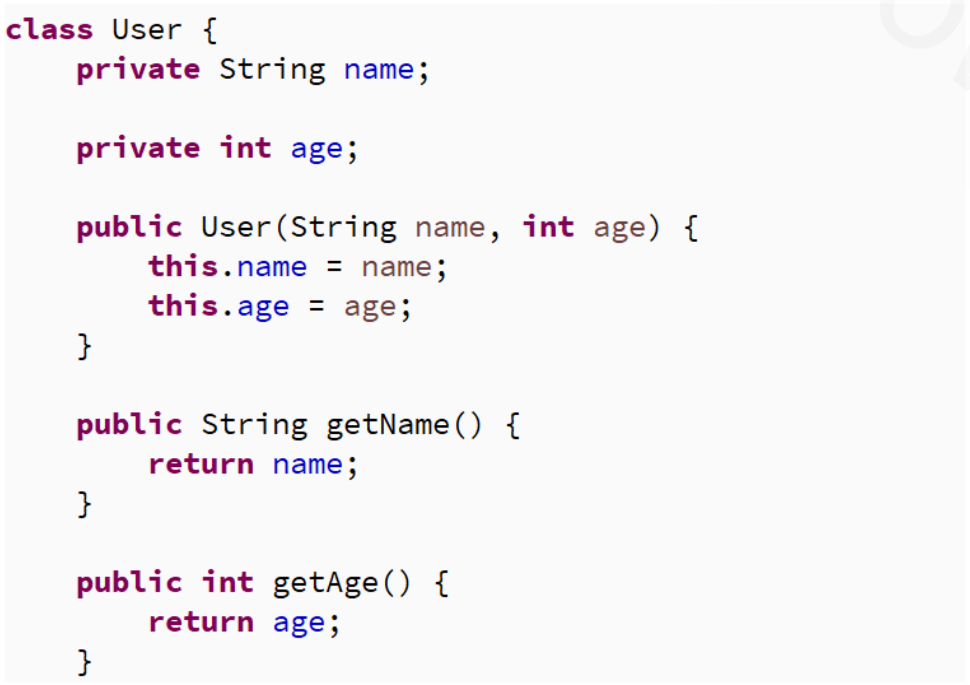
Concrete Strategy



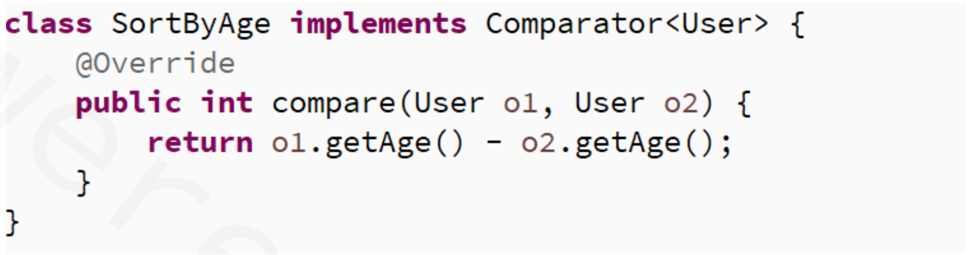
```
public class DetailPrinter implements OrderPrinter {

    @Override
    public void print(Collection<Order> orders) {
        System.out.println("************* Detail Report ***********");
        Iterator<Order> iter = orders.iterator();
        double total = 0;
        for(int i=1;iter.hasNext();i++) {
            double orderTotal = 0;
```

Examples of Strategy pattern

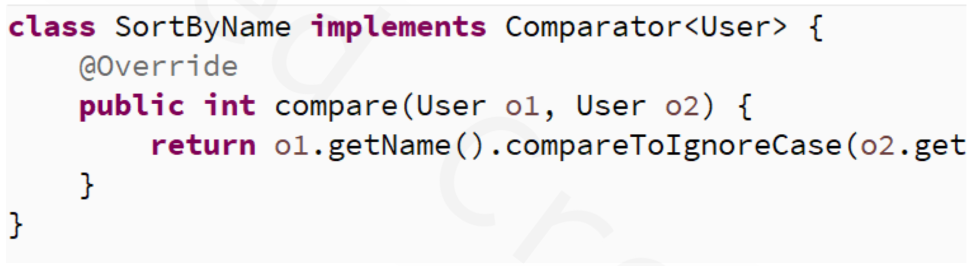
- The java.util.Comparator is a great example of strategy pattern. We can create multiple implementations of comparator, each using a different algorithm to perform comparison and supply those to various sort methods.



```java
class User {
    private String name;

    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
```

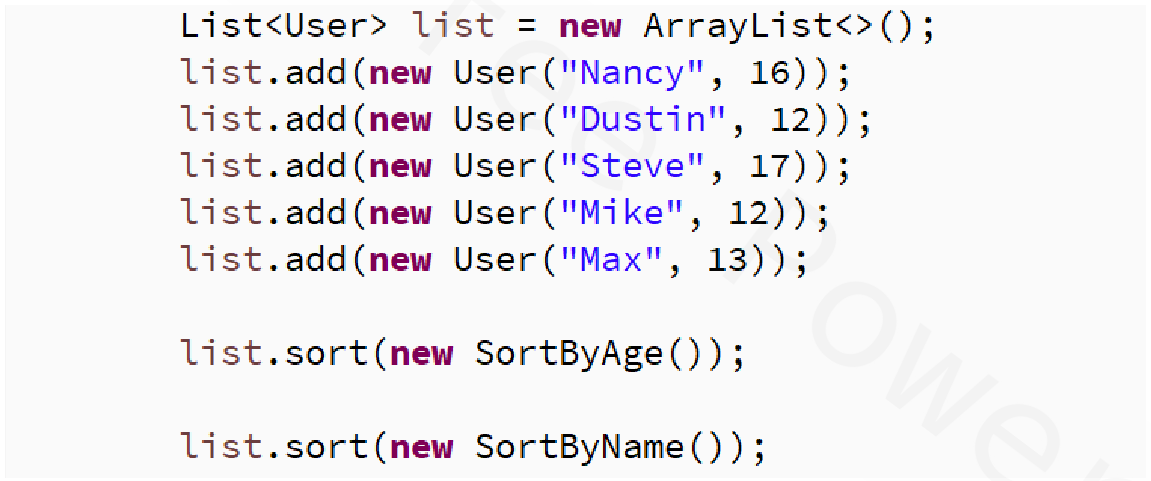


```java
class SortByAge implements Comparator<User> {
    @Override
    public int compare(User o1, User o2) {
        return o1.getAge() - o2.getAge();
    }
}
```



```java
class SortByName implements Comparator<User> {
    @Override
    public int compare(User o1, User o2) {
        return o1.getName().compareToIgnoreCase(o2.get
    }
}
```

Examples of Strategy pattern



```
List<User> list = new ArrayList<>();
list.add(new User("Nancy", 16));
list.add(new User("Dustin", 12));
list.add(new User("Steve", 17));
list.add(new User("Mike", 12));
list.add(new User("Max", 13));

list.sort(new SortByAge());

list.sort(new SortByName());
```

- Another example of strategy pattern is the ImplicitNamingStrategy & PhysicalNamingStrategy contracts in Hibernate. Implementations of these classes are used when mapping an Entity to database tables. These classes tell hibernate which table to use & which columns to use.
- We can use different algorithms to perform mapping from domain model mapping to a logical name via ImplicitNamingStrategy & then from logical name to physical name via PhysicalNamingStrategy.

Template Method

Behavioral Design Patterns

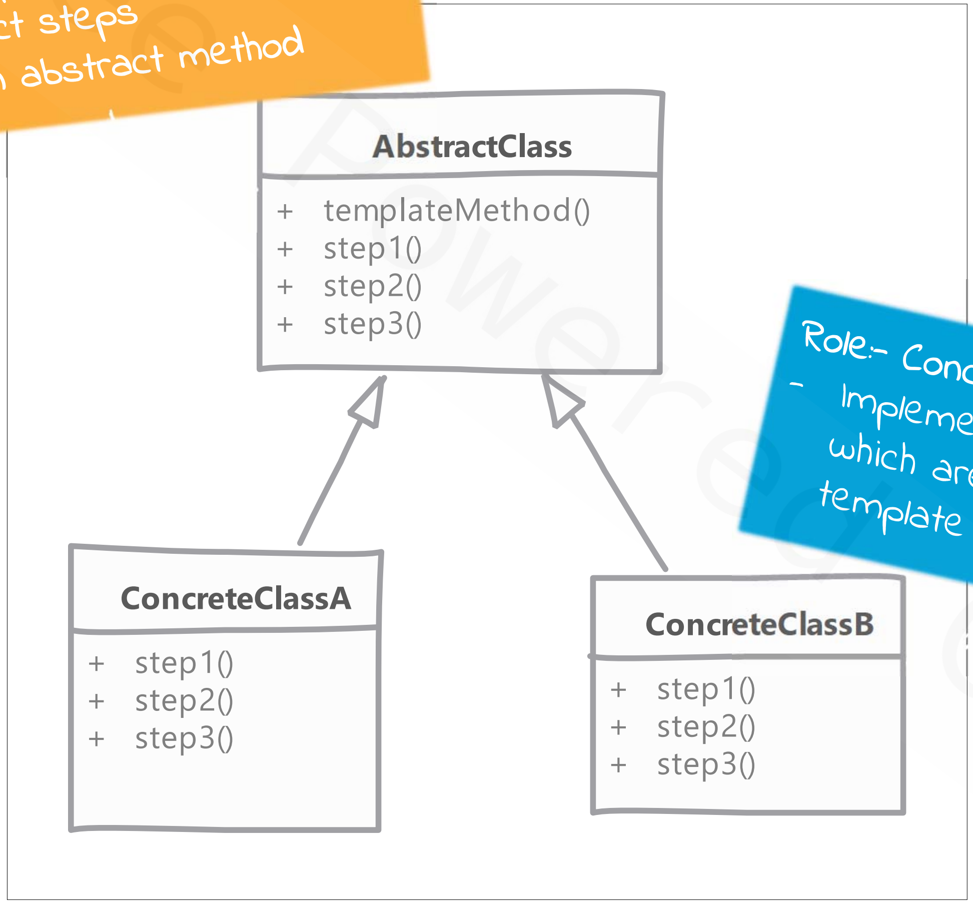
Design Patterns in Java

In-A-Hurry Summary

- Template method allow us to define a skeleton of an algorithm in base class. Steps of algorithm are defined as abstract methods in base class.
- Subclasses of our abstract class will provide implementation of steps. This way we can have different implementations for same algorithm.
- Client will create object of any of the concrete subclasses and use the algorithm.
- Factory method design pattern is often implemented as part of template method design pattern.
- One drawback of template method is algorithm implementation is now spread across multiple classes so it makes it slightly difficult to understand.

In-A-Hurry Summary

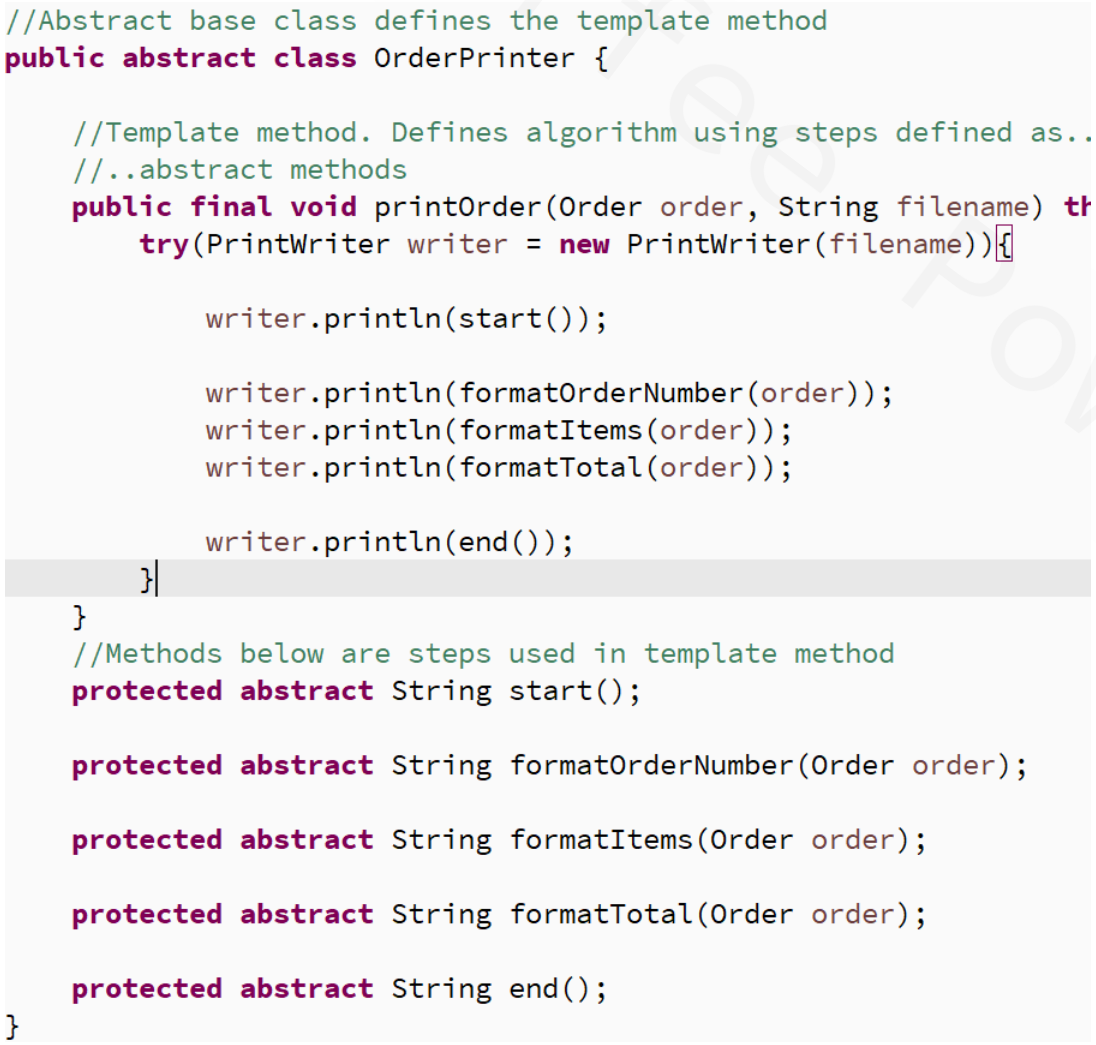
Role:- Abstract class
- Implements template method using one or more abstract steps
- Each step is an abstract method



Role:- Concrete class
- Implements individual steps which are then called by template method

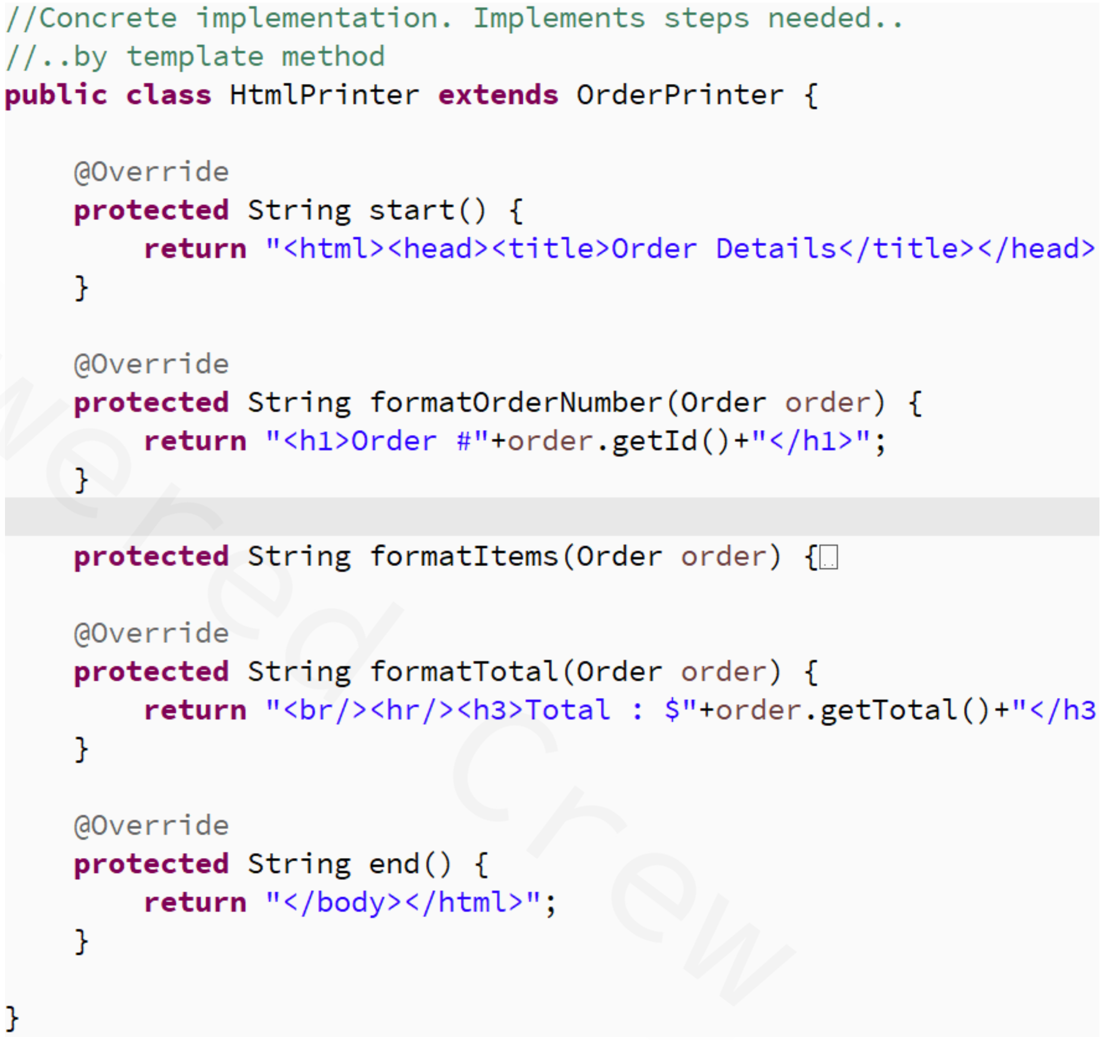
In-A-Hurry Summary

Abstract Class



```
//Abstract base class defines the template method
public abstract class OrderPrinter {

    //Template method. Defines algorithm using steps defined as..
    //..abstract methods
    public final void printOrder(Order order, String filename) th
        try(PrintWriter writer = new PrintWriter(filename)){

            writer.println(start());

            writer.println(formatOrderNumber(order));
            writer.println(formatItems(order));
            writer.println(formatTotal(order));

            writer.println(end());
        }
    }
    //Methods below are steps used in template method
    protected abstract String start();

    protected abstract String formatOrderNumber(Order order);

    protected abstract String formatItems(Order order);

    protected abstract String formatTotal(Order order);

    protected abstract String end();
}
```

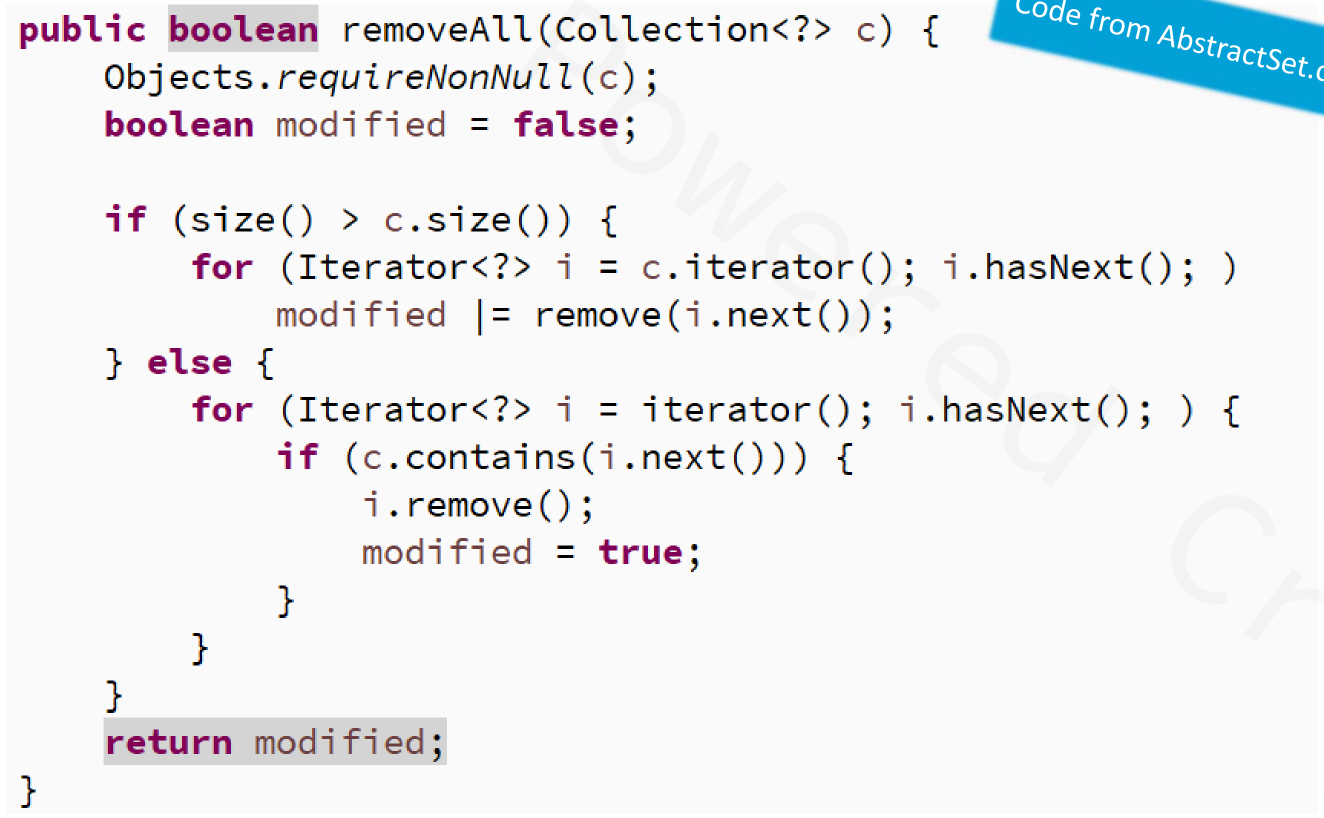
Concrete Class



```
//Concrete implementation. Implements steps needed..
//..by template method
public class HtmlPrinter extends OrderPrinter {

    @Override
    protected String start() {
        return "<html><head><title>Order Details</title></head>
    }

    @Override
    protected String formatOrderNumber(Order order) {
        return "<h1>Order #"+order.getId()+"</h1>";
    }

    protected String formatItems(Order order) {

    @Override
    protected String formatTotal(Order order) {
        return "<br/><hr/><h3>Total : $"+order.getTotal()+"</h3
    }

    @Override
    protected String end() {
        return "</body></html>";
    }

}
```

Examples of Template Method

- The java.util.AbstractMap, java.util.AbstractSet classes have many non-abstract methods that are good exampels of template method pattern.



```
public boolean removeAll(Collection<?> c) {
    Objects.requireNonNull(c);
    boolean modified = false;

    if (size() > c.size()) {
        for (Iterator<?> i = c.iterator(); i.hasNext(); )
            modified |= remove(i.next());
    } else {
        for (Iterator<?> i = iterator(); i.hasNext(); ) {
            if (c.contains(i.next())) {
                i.remove();
                modified = true;
            }
        }
    }
    return modified;
}
```

Code from AbstractSet.class

Visitor

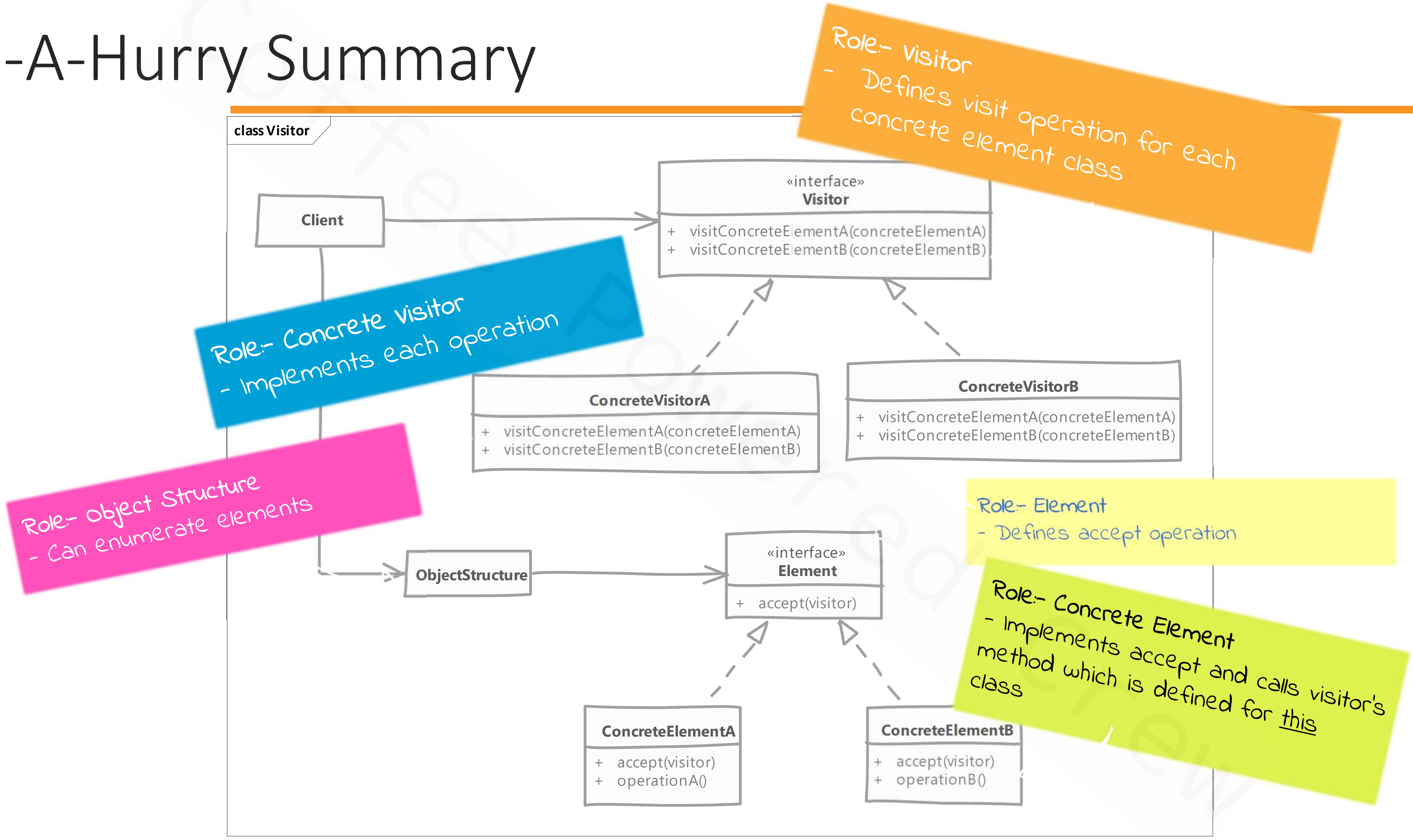
Behavioral Design Patterns

Design Patterns in Java

In-A-Hurry Summary

- Visitor pattern allows to add new operations that work on objects without modifying class definitions of these objects.
- Visitors define class specific methods which work with an object of that class to provide new functionality.
- To use this pattern classes define a simple accept method which gets a reference to a visitor and inside this method, objects class method on visitor which is defined for that specific class.
- Adding a new functionality means creating a new visitor and implementing new functionality in that class instead of modifying each class where this functionality is needed.
- This pattern is often used where we have an object structure and then another class or visitor itself iterates over this structure passing our visitor object to each object.

In-A-Hurry Summary



class Visitor
Client
«interface»
Visitor
+ visitConcreteElementA(concreteElementA)
+ visitConcreteElementB(concreteElementB)
ConcreteVisitorA
+ visitConcreteElementA(concreteElementA)
+ visitConcreteElementB(concreteElementB)
ConcreteVisitorB
+ visitConcreteElementA(concreteElementA)
+ visitConcreteElementB(concreteElementB)
ObjectStructure
«interface»
Element
+ accept(visitor)
ConcreteElementA
+ accept(visitor)
+ operationA()
ConcreteElementB
+ accept(visitor)
+ operationB()
Role:- Visitor
- Defines visit operation for each concrete element class
Role:- Concrete Visitor
- Implements each operation
Role:- Object Structure
- Can enumerate elements
Role:- Element
- Defines accept operation
Role:- Concrete Element
- Implements accept and calls visitor's method which is defined for this class

In-A-Hurry Summary

Element Interface

```java
public interface Employee {

    int getPerformanceRating();

    void setPerformanceRating(int rating);

    Collection<Employee> getDirectReports();

    void accept(Visitor visitor);

    int getEmployeeId();
}
```

Abstract Element

```java
public abstract class AbstractEmployee implements Employee {

    private int performanceRating;

    private String name;

    private static int employeeIdCounter = 101;

    private int employeeId;

    public AbstractEmployee(String name) {□
```

Concrete Elements

```java
public class Programmer extends AbstractEmployee {
    private String skill;

    public Programmer(String name, String skill) {
        super(name);
        this.skill = skill;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
```

```java
public class ProjectLead extends AbstractEmployee {

    private List<Employee> directReports = new ArrayList<>();

    public ProjectLead(String name, Employee...employees) {
        super(name);
        Arrays.stream(employees).forEach(directReports::add);
    }

    @Override
    public Collection<Employee> getDirectReports() {
        return directReports;
    }
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
```

In-A-Hurry Summary

Visitor

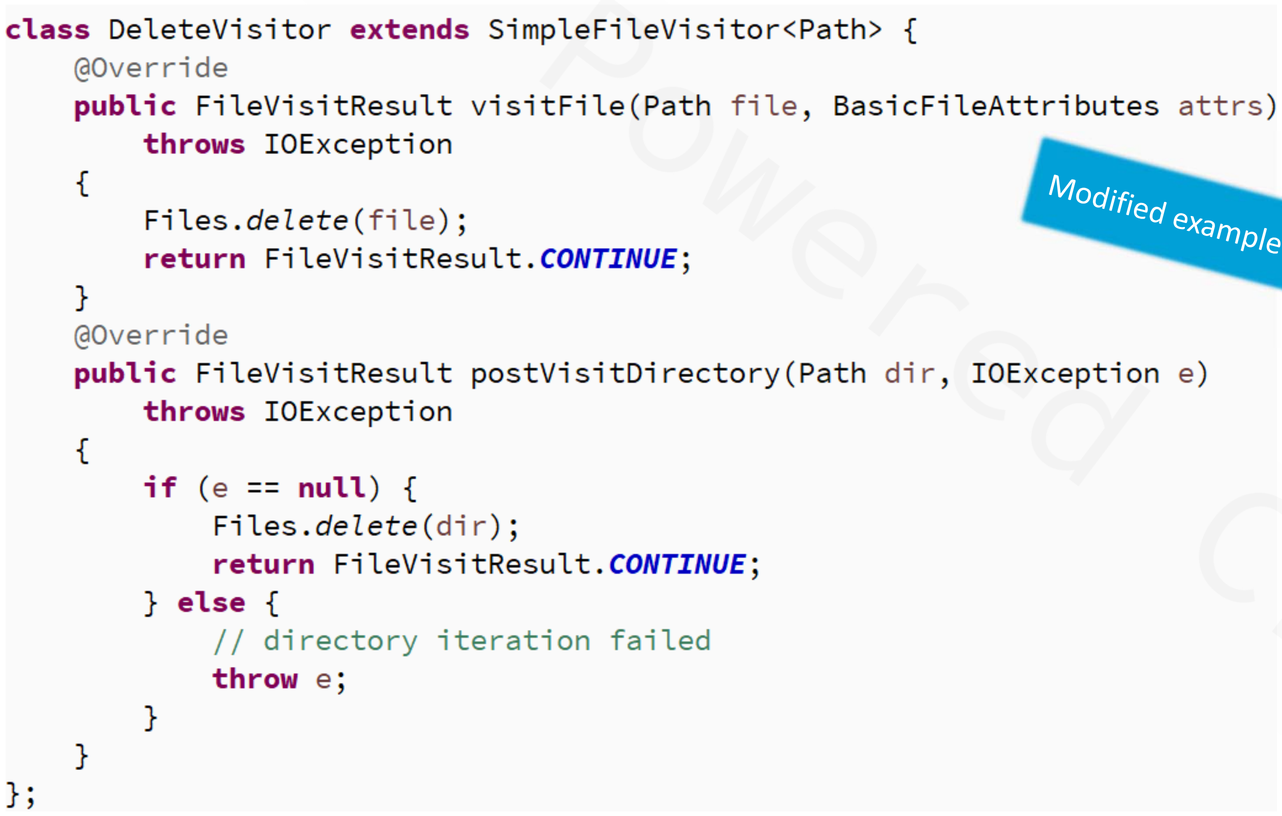
```java
public interface Visitor {

    void visit(Programmer programmer);

    void visit(ProjectLead lead);

    void visit(Manager manager);

    void visit(VicePresident vp);
}
```

Concrete Visitor

```java
public class PrintVisitor implements Visitor {

    private Ratings ratings;

    public PrintVisitor() {}

    public PrintVisitor(Ratings ratings) {

    @Override
    public void visit(Programmer programmer) {
        String msg = programmer.getName() + " is a "+programmer.getSkill()
        +" Programmer";
        msg += getRatings(programmer);
        System.out.println(msg);
    }

    @Override
    public void visit(ProjectLead lead) {
        String msg = lead.getName()+" is a lead with "+lead.getDirectRepor
        msg += getRatings(lead);
        System.out.println(msg);
    }
```

Examples of Visitor Pattern

- The dom4j library used for parsing XML has interface org.dom4j.Visitor & implementation org.dom4j.VisitorSupport which are examples of visitor. By implementing this visitor we can process each node in an XML tree.
- Another example of visitor pattern is the java.nio.file.FileVisitor & its implementation SimpleFileVisitor.



```java
class DeleteVisitor extends SimpleFileVisitor<Path> {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException
    {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }
    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException e)
        throws IOException
    {
        if (e == null) {
            Files.delete(dir);
            return FileVisitResult.CONTINUE;
        } else {
            // directory iteration failed
            throw e;
        }
    }
};
```

Modified example from Javadoc

Null Object

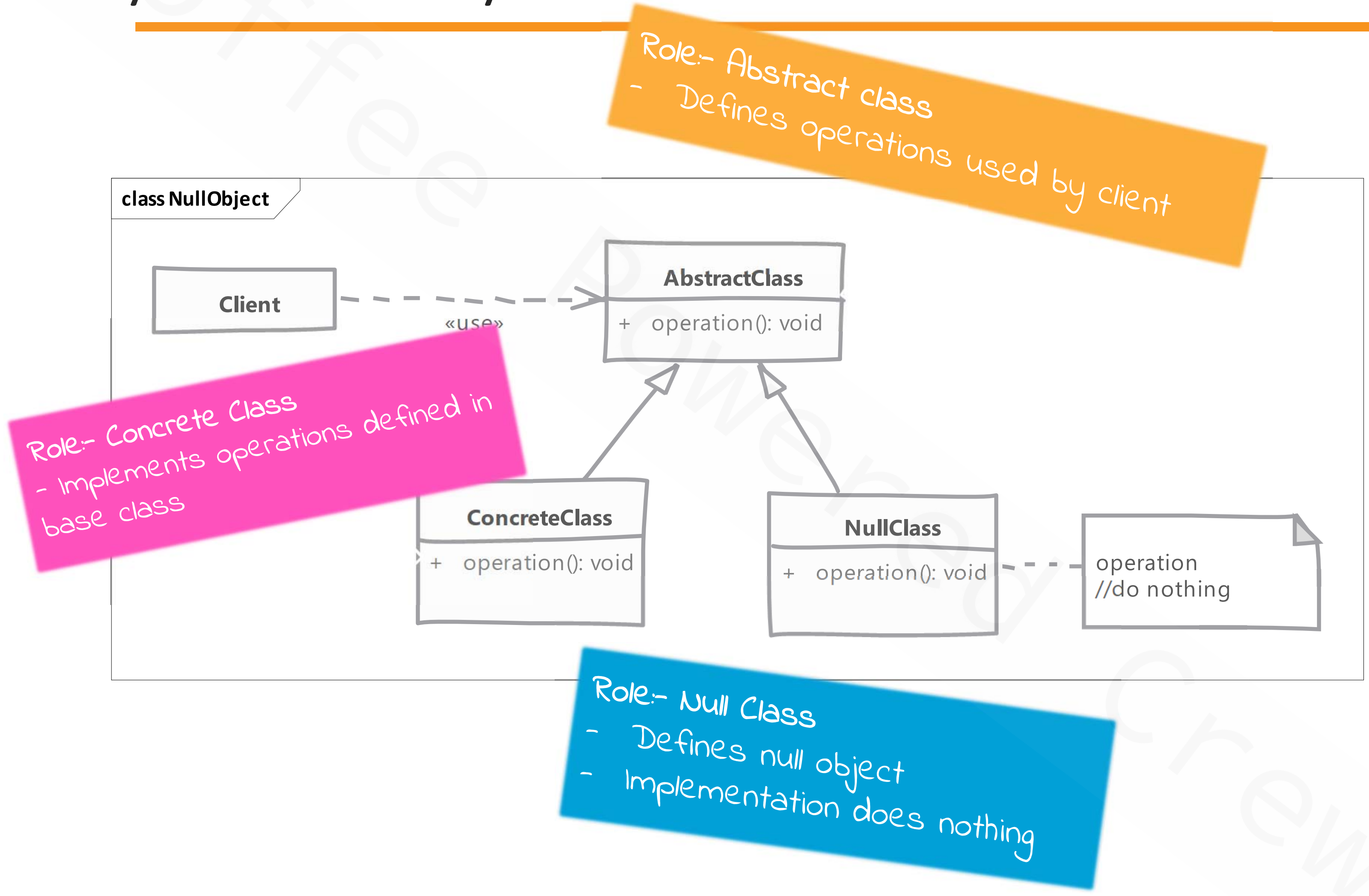
Behavioral Design Patterns

Design Patterns in Java

In-A-Hurry Summary

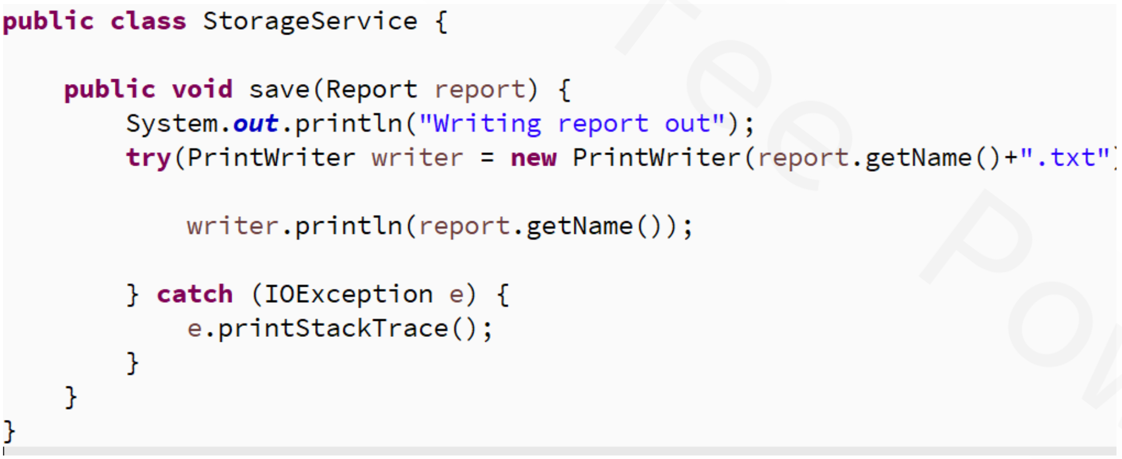
- Null object pattern allows us to represent absence of real object as a do nothing object.
- Method implementations in a Null object will not do anything. In case a return value is expected, these methods will return a sensible, hard-coded default value.
- Classes which use Null object won’t be aware of presence of this special implementation. Whole purpose of the pattern is to avoid null checks in other classes.
- Null objects do not transform into real objects, nor do they use indirection to real objects.

In-A-Hurry Summary



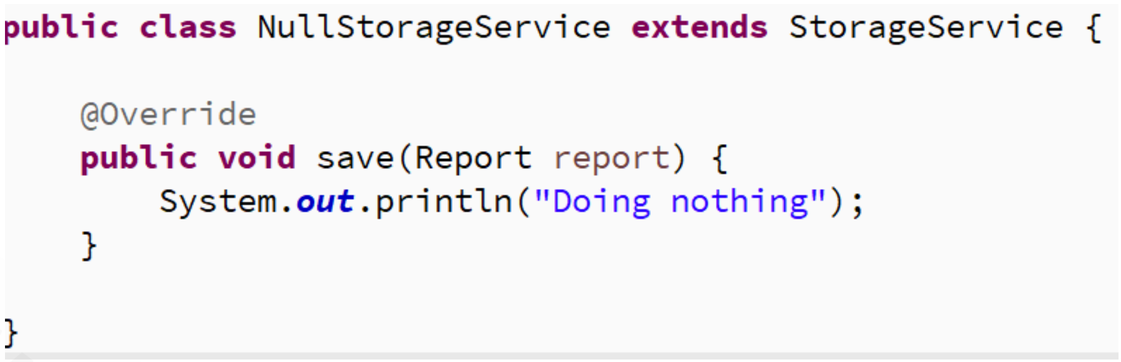
In-A-Hurry Summary

Base Class



```java
public class StorageService {

    public void save(Report report) {
        System.out.println("Writing report out");
        try(PrintWriter writer = new PrintWriter(report.getName()+".txt"

            writer.println(report.getName());

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

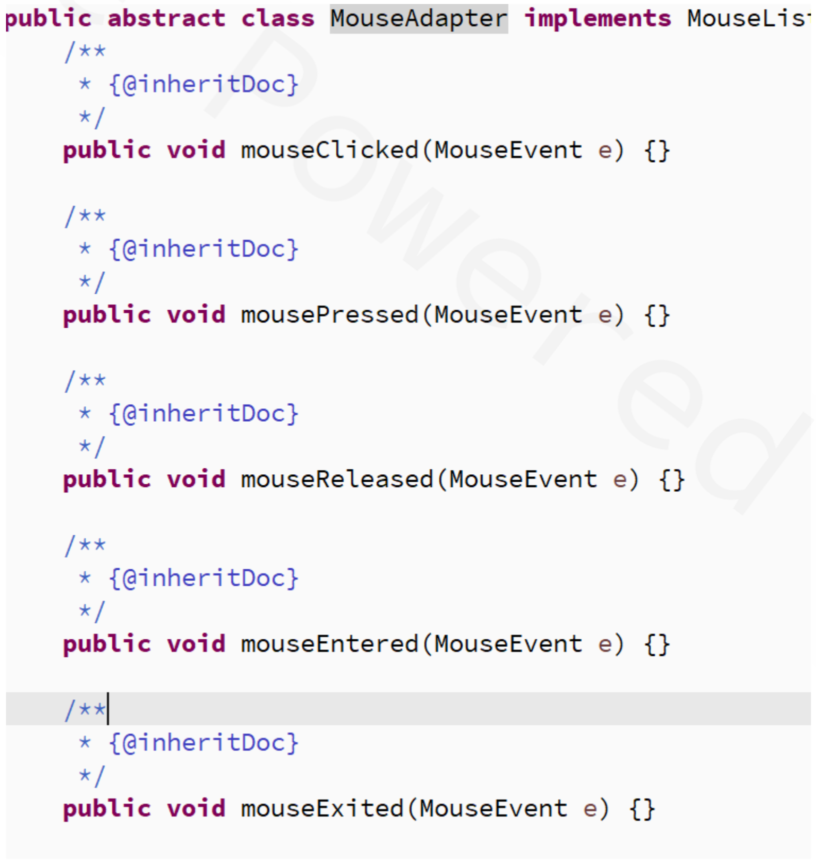
Null Object



```java
public class NullStorageService extends StorageService {

    @Override
    public void save(Report report) {
        System.out.println("Doing nothing");
    }

}
```

Example of Null Object

- The various adapter classes from java.awt.event package can be thought of as examples null object. Only reason they are not *the* examples of this pattern is that they are abstract classes but without any abstract method.



```
public abstract class MouseAdapter implements MouseLis
    /**
     * {@inheritDoc}
     */
    public void mouseClicked(MouseEvent e) {}

    /**
     * {@inheritDoc}
     */
    public void mousePressed(MouseEvent e) {}

    /**
     * {@inheritDoc}
     */
    public void mouseReleased(MouseEvent e) {}

    /**
     * {@inheritDoc}
     */
    public void mouseEntered(MouseEvent e) {}

    /**
     * {@inheritDoc}
     */
    public void mouseExited(MouseEvent e) {}
```