Refactoring Your Code with Java 8

FP to the Rescue

Eder Ignatowicz
@ederign
Sr. Software Engineer
JBoss by Red Hat


SOU Java
sociedade de usuários java
Campinas

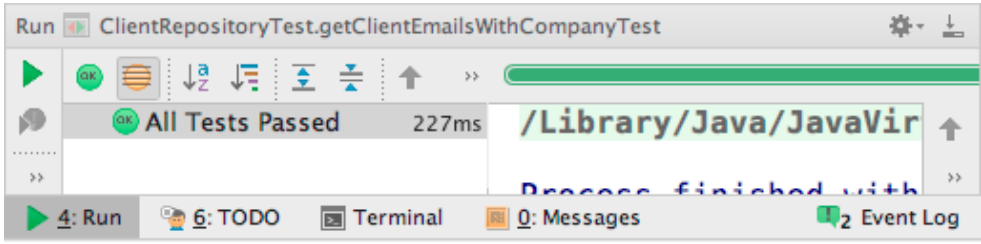
Design Patterns + Java + Functional Programming <3

Refactoring Loops to Collection Pipelines

```
public class Client {

    private String name;
    private String email;
    private Company company;

    public Client( String name, String email, Company company ) {
        this.name = name;
        this.email = email;
        this.company = company;
    }

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

    public Client( String name, String email ) {
        this.name = name;
        this.email = email;
    }


 …
}
```

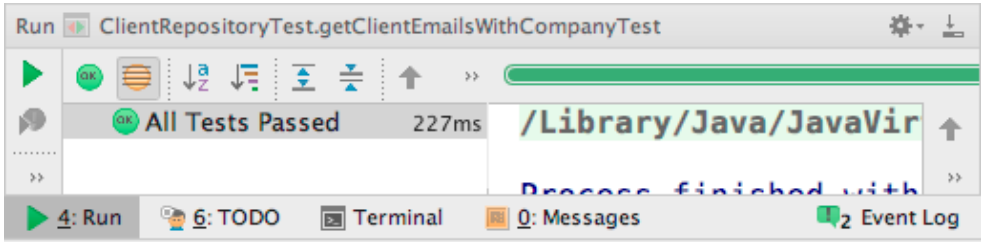
```java
public class ClientRepositoryTest {


    private ClientRepository repo;

    @Before
    public void setup() {
        Company rh = new Company( "RedHat" );
        Client full1 = new Client( "Full1", "full1@redhat.com", rh );
        Client full2 = new Client( "Full2", "full2@redhat.com", rh );
        Client noCompany = new Client( "noCompany", "noCompany@ederign.me" );
        Client onlyName = new Client( "onlyName" );
        repo = new ClientRepository(
                Arrays.asList( full1, noCompany, full2, onlyName ) );

    }

    @Test
    public void getClientEmailsWithCompanyTest() {
        List<String> clientMails = repo.getClientMails();
        assertEquals( 2, clientMails.size() );
        assertTrue( clientMails.contains( "full1@redhat.com" ) );
        assertTrue( clientMails.contains( "full2@redhat.com" ) );
        assertTrue( !clientMails.contains( "noCompany@ederign.me" ) );
    }
}
```

```
public List<String> getClientMails() {
     ArrayList<String> emails = new ArrayList<>();

     for ( Client client : clients ) {
         if ( client.getCompany() != null ) {
             String email = client.getEmail();
             if ( email != null ){
                 emails.add( email );
             }
         }
     }

     return emails;
 }
```



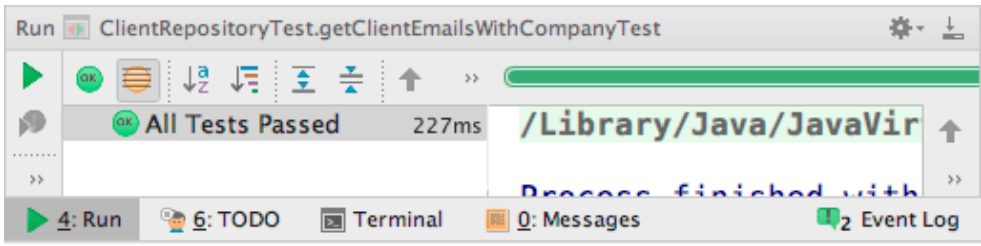
```
    public List<String> getClientMails() {
        ArrayList<String> emails = new ArrayList<>();
        List<Client> pipeline = clients;
        for ( Client client : pipeline ) {
            if ( client.getCompany() != null ) {
                String email = client.getEmail();
                if ( email != null ){
                    emails.add( email );
                }
            }
        }
        return emails;
    }
}
```

Extract Variable



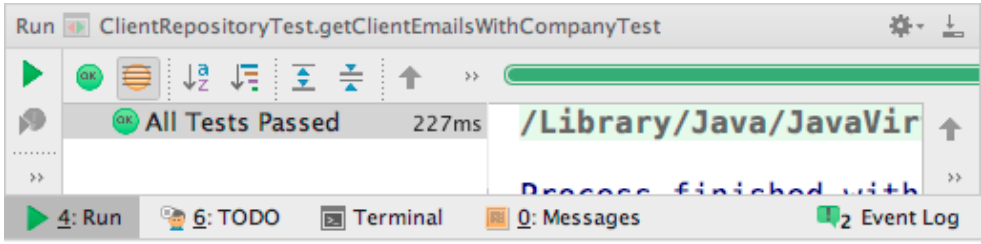
```
    public List<String> getClientMails() {
        ArrayList<String> emails = new ArrayList<>();
        List<Client> pipeline = clients
                .stream()
                .filter( c -> c.getCompany() != null )
                .collect( Collectors.toList() );
        for ( Client client : pipeline ) {
            if ( client.getCompany() != null ) {
                String email = client.getEmail();
                if ( email != null ) {
                    emails.add( email );
                }
            }
        }

        return emails;
    }
}
```

Filter Operation



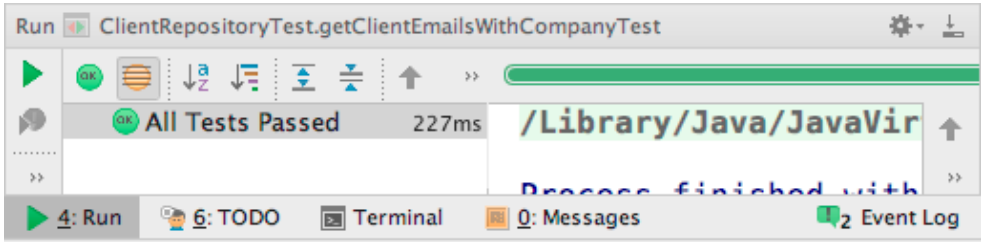
```
public List<String> getClientMails() {
     ArrayList<String> emails = new ArrayList<>();
     List<String> pipeline = clients
              .stream()
              .filter( c -> c.getCompany() != null )
              .map( c -> c.getEmail() )
              .collect( Collectors.toList() );
     for ( String mail : pipeline ) {
         String email = client.getEmail();
         if ( email != null ) {
             emails.add( mail );
         }
     }

     return emails;
 }
```

Map Operation



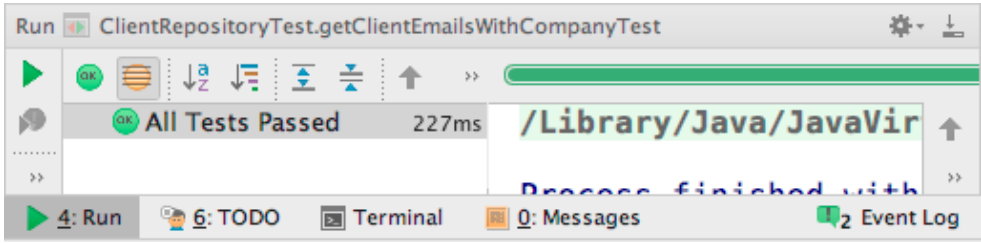
```
public List<String> getClientMails() {
    ArrayList<String> emails = new ArrayList<>();
    List<String> pipeline = clients
            .stream()
            .filter( c -> c.getCompany() != null )
            .map( c -> c.getEmail() )
            .filter( m -> m != null )
            .collect( Collectors.toList() );
    for ( String mail : pipeline ) {
        if ( mail != null ) {
            emails.add( mail );
        }
    }

    return emails;
}
```

Filter Operation

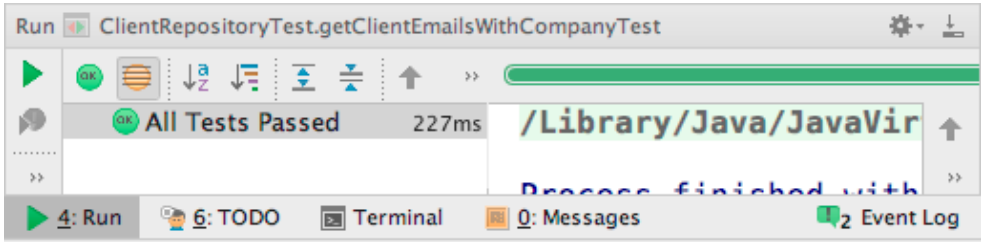


```
public List<String> getClientMails() {
    ArrayList<String> emails = new ArrayList<>();
    return clients
            .stream()
            .filter( c -> c.getCompany() != null )
            .map( c -> c.getEmail() )
            .filter( m -> m != null )
            .collect( Collectors.toList() );
    for ( String mail : pipeline ) {
        if ( mail != null ) {
            emails.add( mail );
        }
    }

    return emails;
}
```

Pipeline



```java
public List<String> getClientMails() {
    return clients
            .stream()
            .filter( c -> c.getCompany() != null )
            .map( c -> c.getEmail() )
            .filter( m -> m != null )
            .collect( Collectors.toList() );
}
```



Strategy

"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it."
GAMMA, Erich et al.

```java
public class ShoppingCartTest {

    ShoppingCart cart;

    @Before
    public void setup() {
        Item item1 = new Item( 10 );
        Item item2 = new Item( 20 );
        cart = new ShoppingCart( Arrays.asList( item1, item2 ) );
    }

    @Test
    public void totalTest() {
        cart.pay( ShoppingCart.PaymentMethod.CREDIT ) );
    }
}
```

```
public class ShoppingCart {

    private List<Item> items;

     public ShoppingCart( List<Item> items ) {
         this.items = items;
     }

     public void pay( PaymentMethod method ) {
         int total = cartTotal();
         if ( method == PaymentMethod.CREDIT ) {
             System.out.println( “Pay with credit “ + total);
         } else if ( method == PaymentMethod.MONEY ) {
             System.out.println( “Pay with money “ + total );
         }
    }

     private int cartTotal() {
         return  items
                 .stream()
                 .mapToInt( Item::getValue )
                 .sum();
     }
…
}
```

```
public class ShoppingCart {

    private List<Item> items;

     public ShoppingCart( List<Item> items ) {
         this.items = items;
     }

     public void pay( PaymentMethod method ) {
         int total = cartTotal();
         if ( method == PaymentMethod.CREDIT ) {
             System.out.println( “Pay with credit “ + total );
         } else if ( method == PaymentMethod.MONEY ) {
             System.out.println( “Pay with money “ + total );
         }
     }

      private int cartTotal() {
         return  items
                 .stream()
                 .mapToInt( Item::getValue )
                 .sum();
     }
…
}
```

```java
public interface Payment {

    public void pay(int amount);

}
```

```java
public class CreditCard implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "Pay with Credit: “+ amount);
    }
}
```

```java
public class Money implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "Pay with Money: “+ amount);
}
```

```
public class ShoppingCart {
    …
    public void pay( Payment method ) {
        int total = cartTotal();
        method.pay( total );
    }

    private int cartTotal() {
     return  items
             .stream()
             .mapToInt( Item::getValue )
             .sum();
    }
}
```

Strategy

```java
public interface Payment {

    public void pay(int amount);

}
```

```java
public class CreditCard implements
Payment {
    @Override
    public void pay( int amount ) {
        System.out.println(
          "make credit payment logic" );
    }
}
```

```java
public class Money implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println(
          "make money payment logic" );
    }
}
```

```java
public class DebitCard implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println(
          "make debit payment logic" );
    }
}
```

```java
    public void totalTest() {
        assertEquals( 30, cart.pay( new CreditCard() ) );
        assertEquals( 30, cart.pay( new Money() ) );
        assertEquals( 30, cart.pay( new DebitCard() ) );
    }
}
```

java.util.function

Interface Consumer<T>

Type Parameters:

`T - the type of the input to the operation`

All Known Subinterfaces:

`Stream.Builder<T>`

Functional Interface:

`This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.`

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`void`	**`accept(T t)`** Performs this operation on the given argument.
`default Consumer<T>`	**`andThen(Consumer<? super T> after)`** Returns a composed `Consumer` that performs, in sequence, this operation followed by the `after` operation.

```
public void pay( Payment method ) {
        int total = cartTotal();
        method.pay( total );
    }
```

```
public class ShoppingCart {
    …

    public void pay( Consumer<Integer> method ) {
        int total = cartTotal();
        method.accept( total );
    }

    …
}
```

```
public class ShoppingCart {
        …

    public void pay( Consumer<Integer> method ) {
        int total = cartTotal();
        method.accept( total );
    }
        …
}
```

```
public void totalTest() {
    cart.pay( amount -> System.out.println( "Pay with Credit: " + amount ) );
    cart.pay( amount -> System.out.println( "Pay with Money: " + amount ) );
    cart.pay( amount -> System.out.println( "Pay with Debit: " + amount ) );
}
```

```java
public class PaymentTypes {

    public static void money( int amount ) {
        System.out.println( "Pay with Money: " + amount );
    }

    public static void debit( int amount ) {
        System.out.println( "Pay with Debit: " + amount );
    }

    public static void credit( int amount ) {
        System.out.println( "Pay with Credit: " + amount );
    }

}
```

```java
public void totalTest() {
    cart.pay( PaymentTypes::credit );
    cart.pay( PaymentTypes::debit );
    cart.pay( PaymentTypes::money );
}
```

```
public class ShoppingCart {
    …
    public void pay( Consumer<Integer> method ) {
        int total = cartTotal();
        method.accept( total );
    }

    private int cartTotal() {
        return  items
                .stream()
                .mapToInt( Item::getValue )
                .sum();
    }
}
```

Strategy

Decorator

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.."
GAMMA, Erich et al.

```
new BufferedReader(new FileReader(new File("some.file")));
```

```
public class Item {
    private int price;

    public Item( int price ) {
        this.price = price;
    }

    public int getPrice() {
        return price;
    }
}
```

Extras

Shipping
Taxes
Packing

```java
public interface Item {
    int getPrice();
}
```

```java
public class Book implements Item {
    private int price;

    public Book( int price ) {
        this.price = price;
    }

    @Override
    public int getPrice() {
        return price;
    }
}
```

```
public abstract class ItemExtras implements Item {

    private Item item;

    public ItemExtras( Item item ) {
        this.item = item;
    }

    @Override
    public int getPrice() {
        return item.getPrice();
    }
}
```

```java
public class InternationalDelivery extends ItemExtras {

    public InternationalDelivery( Item item ) {
        super( item );
    }

    @Override
    public int getPrice() {
        return 5  + super.getPrice();
    }
}
```

```
public class GiftPacking extends ItemExtras {

    public GiftPacking( Item item ) {
        super( item );
    }

    @Override
    public int getPrice() {
        return 15 + super.getPrice();
    }
}
```

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

     Item book = new Book( 10 );
     book.getPrice(); //10

     Item international = new InternationalDelivery( book );
     international.getPrice(); //15

 }
```

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

      Item book = new Book( 10 );
      book.getPrice(); //10

      Item internationalGift = new GiftPacking(
                                    new InternationalDelivery( book ) );
      internationalGift.getPrice(); //30

}
```

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

        Item book = new Book( 10 );
        book.getPrice(); //10

        Item internationalGiftWithTaxes = new InternacionalTaxes(
                                                    new GiftPacking(
                                                        new InternationalDelivery( book );
        internationalGiftWithTaxes.getPrice(); //80
    }
}
```

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

        Item book = new Item( 10 );
        book.getPrice(); //10

        Function<Integer, Integer> giftPacking = value -> value + 15;
        giftPacking.apply( book.getPrice() ); //25
}
```

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

        Item book = new Item( 10 );
        book.getPrice(); //10

        Function<Integer, Integer> giftPacking = value -> value + 15;
        giftPacking.apply( book.getPrice() ); //25

        Function<Integer, Integer> intTaxes = value -> value + 50;
        intTaxes.apply( book.getPrice() ); //60

}
```

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

        Item book = new Item( 10 );
        book.getPrice(); //10

        Function<Integer, Integer> giftPacking = value -> value + 15;
        giftPacking.apply( book.getPrice() ); //25

        Function<Integer, Integer> intTaxes = value -> value + 50;
        intTaxes.apply( book.getPrice() ); //60

        giftPacking.andThen( intTaxes ).apply( book.getPrice() ); //75

}
```

```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    public Item( int price ) {
        this.price = price;
    }

    public Item( int price, Function<Integer, Integer>... itemExtras) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

    public int getPrice() {
        int priceWithExtras = price;
        for ( Function<Integer, Integer> itemExtra : itemExtras ) {
            priceWithExtras = itemExtra.apply( priceWithExtras );
        }
        return priceWithExtras;
    }

    public void setItemExtras( Function<Integer, Integer>... itemExtras ) {
        this.itemExtras = itemExtras;
    }
}
```

```
public static void main( String[] args ) {
    Item book = new Item( 10 );
    Function<Integer, Integer> giftPacking = value -> value + 15;
    Function<Integer, Integer> intTaxes = value -> value + 50;

    book.setItemExtras( giftPacking, intTaxes );

    book.getPrice(); //75

}
```

```
public static void main( String[] args ) {
    Item book = new Item( 10 );
    Function<Integer, Integer> giftPacking = value -> value + 15;
    Function<Integer, Integer> intTaxes = value -> value + 50;

    book.setItemExtras( giftPacking, intTaxes );

    book.getPrice(); //75

}
```

```
 public class Packing {

      public static Integer giftPacking( Integer value ) {
          return value + 15;
      }
 //other packing options here
 }

public class Taxes {

     public static Integer internacional( Integer value ) {
         return value + 50;
     }
//other taxes here
}
```

```
public static void main( String[] args ) {
    Item book = new Item( 10,
                          Packing::giftPacking,
                        Taxes::internacional );

    book.getPrice(); //75
}
```

```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    public Item( int price ) {
        this.price = price;
    }

    public Item( int price, Function<Integer, Integer>... itemExtras) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

    public int getPrice() {
        int priceWithExtras = price;
        for ( Function<Integer, Integer> itemExtra : itemExtras ) {
            priceWithExtras = itemExtra.apply( priceWithExtras );
        }
        return priceWithExtras;
    }

    public void setItemExtras( Function<Integer, Integer>... itemExtras ) {
        this.itemExtras = itemExtras;
    }
}
```

```java
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    public Item( int price ) {
        this.price = price;
    }

    public Item( int price, Function<Integer, Integer>... itemExtras ) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

    public int getPrice() {
        Function<Integer, Integer> extras =
                Stream.of( itemExtras )
                .reduce( Function.identity(), Function::andThen );
        return extras.apply( price );
    }

    public void setItemExtras( Function<Integer, Integer>... itemExtras ) {
        this.itemExtras = itemExtras;
    }
}
```

Template

"Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."
GAMMA, Erich et al.

```
public abstract class Banking {

    public void processOperation( Operation op ) {
        preProcessing( op );
        process( op );
        postProcessing( op );
    }

    protected abstract void postProcessing( Operation op );

    protected abstract void preProcessing( Operation op );

    private void process( Operation op ) {
        //logic
        op.process( op );
    }
}
```

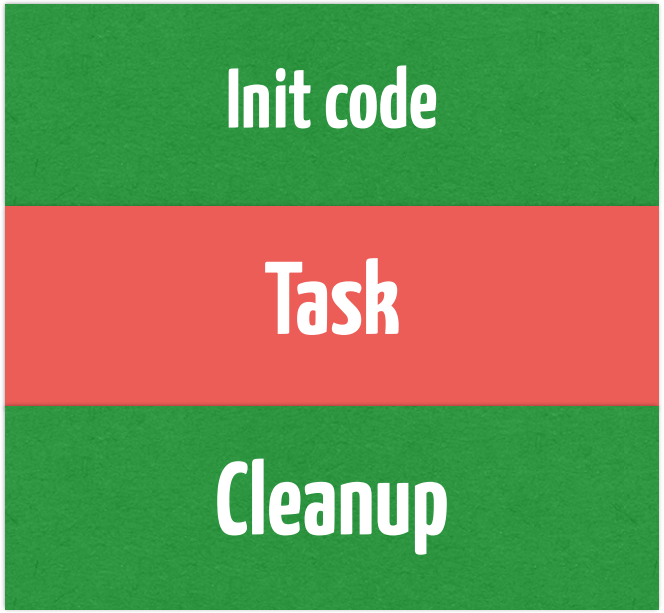
```java
public class VIPBanking extends Banking {

    @Override
    protected void preProcessing( Operation op ) {
        //pre processing vip logic
    }

    @Override
    protected void postProcessing( Operation op ) {
        //post processing vip logic
    }
}
```

```java
public class OnlineBanking extends Banking {

    @Override
    protected void preProcessing( Operation op ) {
        //pre processing online logic
    }

    @Override
    protected void postProcessing( Operation op ) {
        //post processing online logic
    }
}
```

```java
public class Banking {

    public void processOperation( Operation op ) {
        process( op );
    }

    public void processOperation( Operation op,
                                  Consumer<Operation> preProcessing,
                                  Consumer<Operation> postProcessing ) {
        preProcessing.accept( op );
        process( op );
        postProcessing.accept( op );
    }

    private void process( Operation op ) {
        //logic
        op.process( op );
    }
}
```

Execute Around

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

        BufferedReader br = new BufferedReader( new FileReader( "dora.txt" ) );
        try {
                br.readLine();
        }  finally {
                br.close();
        }
}
```


Init code
Task
Cleanup

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

        BufferedReader br = new BufferedReader(
                                    new FileReader( "dora.txt" ) );
        try {
                br.readLine();
        }  finally {
                br.close();
        }
}
```

```
try ( BufferedReader br = new BufferedReader(
                                new FileReader( "dora.txt" )){
 br.readLine();

}
```

```
 @Override
 public ServerTemplate store( final ServerTemplate serverTemplate,
                              final List<ServerTemplateKey> keys){

     final Path path = buildPath( serverTemplate.getId() );
         try {
              ioService.startBatch(path.getFileSystem());
              ioService.write(path, serverTemplate);
                        ioService.write(path, keys);
          } finally {
           ioService.endBatch();
        }
         return serverTemplate;

}
```

```
public void store( final ServerTemplate serverTemplate,
                   final List<ServerTemplateKeys> keys )

    try {
        ioService.startBatch( path.getFileSystem() );
        ioService.write( path, serverTemplate );
        ioService.write( path, keys );
    } finally {
        ioService.endBatch();
    }

}
```

```
public class IOService {
        …
    public void processInBatch( Path path, Consumer<Path> batchOp ) {
        try {
            startBatch( path.getFileSystem() );
            batchOp.accept( path );
        } finally {
            endBatch();
        }
    }
}
```

```
public void store( final ServerTemplate serverTemplate,
                   final List<ServerTemplateKeys> keys){

    try {
        ioService.startBatch( path.getFileSystem() );
        ioService.write( path, serverTemplate );
        ioService.write( path, keys );
    } finally {
        ioService.endBatch();
    }

}
```

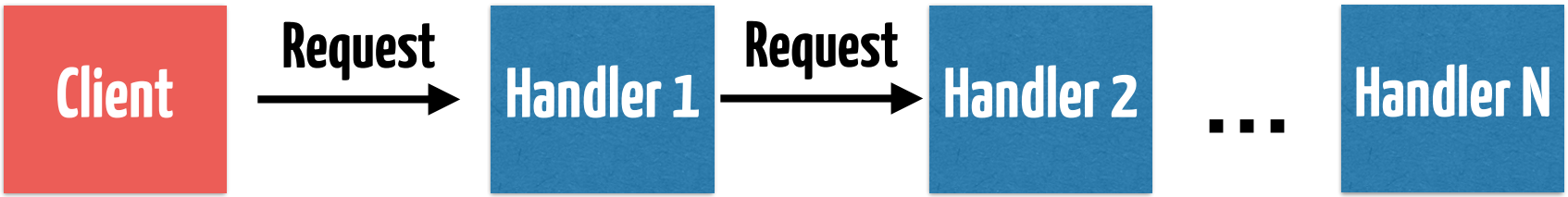
```
public void store( final ServerTemplate serverTemplate,
                   final List<ServerTemplateKeys> keys)
    ioService.processInBatch( path, ( path ) -> {
        ioService.write( path, serverTemplate );
        ioService.write( path, keys );
    } );
 }
```

```
public void delete( final ServerTemplate serverTemplate,
                    final List<ServerTemplateKeys> keys ) {

    ioService.processInBatch( path, ( path ) -> {
        ioService.delete( path, serverTemplate );
        ioService.delete( path, keys );
    } );
}
```

Chain of Responsibilities

"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."
GAMMA, Erich et al.

Chain of Responsibilities




Client
Payment
Payment Processor 1
Payment
Payment Processor 2
. . .
Payment Processor n

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

    PaymentProcessor paymentProcessor = getPaymentProcessor();

    paymentProcessor.process( new Payment( 10 ) );

}

private static PaymentProcessor getPaymentProcessor() {
    PaymentProcessor g = new PaymentProcessorA();

    g.setNext( new PaymentProcessorB() );
    g.setNext( new PaymentProcessorC() );

    return g;
}
```

```
public abstract class PaymentProcessor {

    private PaymentProcessor next;

    public void setNext( PaymentProcessor processors ) {
        if ( next == null ) {
            next = processors;
        } else {
            next.setNext( processors );
        }
    }

    public Payment process( Payment p ) {
        handle( p );
        if ( next != null ) {
            return next.process( p );
        } else {
            return p;
        }
    }

    protected abstract void handle( Payment p );

}
```

```java
public class PaymentProcessorA extends PaymentProcessor {

    @Override
    protected void handle( Payment p ) {
        System.out.println(
                            "PaymentProcessorA for payment: " +
                            p.getAmount() );
    }
}
```

```java
public class PaymentProcessorB extends PaymentProcessor {

    @Override
    protected void handle( Payment p ) {
        System.out.println(
                            "PaymentProcessorB for payment: " +
                            p.getAmount() );
    }
}
```

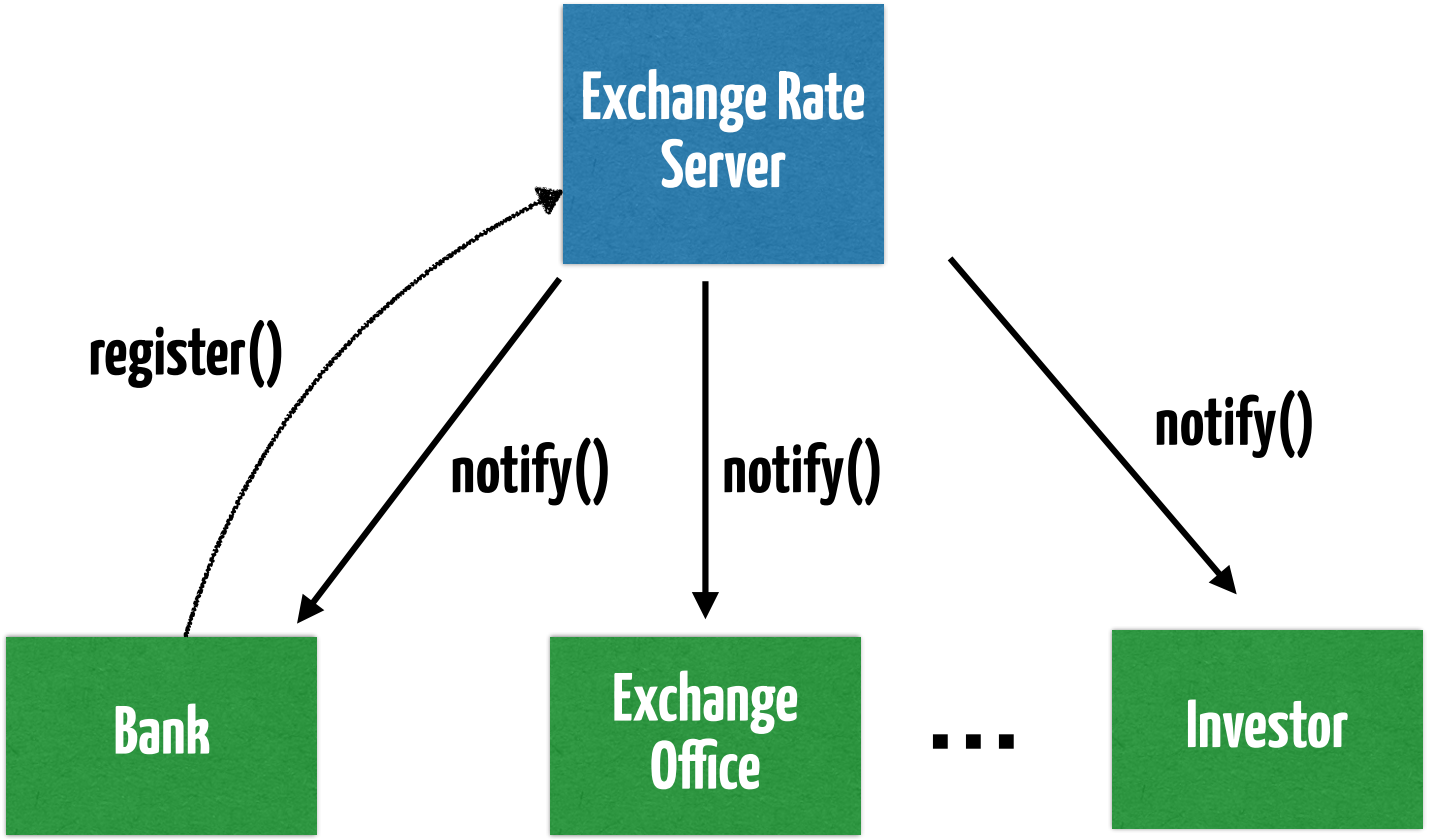
```java
public static void main( String[] args ) {

    PaymentProcessor paymentProcessor = getPaymentProcessor();

    paymentProcessor.process( new Payment( 10 ) );
    //PaymentProcessorA for payment: 10
    //PaymentProcessorB for payment: 10
    //PaymentProcessorC for payment: 10
}

private static PaymentProcessor getPaymentProcessor() {
    PaymentProcessor g = new PaymentProcessorA();

    g.setNext( new PaymentProcessorB() );
    g.setNext( new PaymentProcessorC() );

    return g;
}
```

```
Function<Payment, Payment> processorA =
          p -> {
              System.out.println( "Processor A " + p.getAmount() );
              return p;
          };

  Function<Payment, Payment> processorB =
          p -> {
              System.out.println( "Processor B " + p.getAmount() );
              return p;
          };

  Function<Payment, Payment> processorC =
          p -> {
              System.out.println( "Processor C " + p.getAmount() );
              return p;
          };
```

```
Function<Payment, Payment> processorA =
        p -> {
            System.out.println( "Processor A " + p.getAmount() );
            return p;
        };

Function<Payment, Payment> processorB =
        p -> {
            System.out.println( "Processor B " + p.getAmount() );
            return p;
        };

Function<Payment, Payment> processorC =
        p -> {
            System.out.println( "Processor C " + p.getAmount() );
            return p;
        };

Function<Payment, Payment> chain =
        processorA.andThen( processorB ).andThen( processorC );

chain.apply( new Payment( 10 ) );
   //Processor A 10
   //Processor B 10
   //Processor C 10
```

Observer

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
GAMMA, Erich et al.


Exchange Rate Server
register()
notify()
notify()
notify()
Bank
Exchange Office
...
Investor

```
public interface Subject {
    void registerObserver(Observer observer);
}
```

```
    public interface Observer {
        void notify( ExchangeRate rate );
    }
```

```
    public class Bank implements Observer {

        @Override
        public void notify( ExchangeRate rate ) {
            //some cool stuff here
            System.out.println( "Bank: " + cotacao );
        }

    }


public class Investor implements Observer {

    @Override
    public void notify( ExchangeRate rate ) {
        //some cool stuff here
        System.out.println( "Investor: " + rate );
    }

}
```

```java
public class ExchangeRateServer implements Subject {

    private List<Observer> observers = new ArrayList<>();

    public void newExchangeRate( ExchangeRate rate ) {
        notifyObservers( rate );
    }

    @Override
    public void registerObserver( Observer observer ) {
        observers.add( observer );
    }

    private void notifyObservers( ExchangeRate rate ) {
        observers.forEach( o -> o.notify( rate ) );
    }

}
```

```
public class Main {

    public static void main( String[] args ) {

        Bank bank = new Bank();
        Investor investor = new Investor();

        ExchangeRateServer server = new ExchangeRateServer();


        server.registerObserver( bank );
        server.registerObserver( investor );

        server.newExchangeRate( new Rate( "USD", 4 ) );

    }
}
```

```
Bank:  Rate{currency='USD', valor=4}
Investor:  Rate{currency='USD', valor=4}
```

```java
@Override
public void registerObserver( Observer observer ) {
    observers.add( observer );
}
```

```java
public class Bank implements Observer {

    @Override
    public void notify( ExchangeRate rate ) {
        //some cool stuff here
        System.out.println( "Bank: " + rate );
    }

}
```

```
public class Main {

    public static void main( String[] args ) {


        ExchangeRateServer server = new ExchangeRateServer();


        server.registerObserver(
                rate -> System.out.println( "Bank:  " + rate ) );

        server.registerObserver(
                rate -> {
                    //some cool stuff here
                    System.out.println( "Investor:  " + rate )
                } );

        server.newExchangeRate( new ExchangeRate( "BRL", 1 ) );

    }
}
```

```
Bank:  Rate{currency='BRL', valor=1}
Investor:  Rate{currency='BRL', valor=1}
```

Currying

$$f(x,y) = y/x$$

$$f(2,3)$$
$$f(x,y) = y/x$$

```
f(2, y) = y / 2

g(y) = f(2,y) =
       y/2
```

$$g(y) = f(2,y) = y/2$$

$$g(3) = f(2,3) = 3/2$$

```
CtoF(x) = x * 9/5 +
          32
```

```
static double converter( double x, double f, double b ) {
    return x * f + b;
}

public static void main( String[] args ) {

    Double celsius = 15.0;
    Double fahrenheit = converter( celsius, 9.0 / 5, 32 ); //59 F
}
```

```
    static double converter( double x, double f, double b ) {
        return x * f + b;
    }
```

```
static DoubleUnaryOperator curriedConverter( double f, double b ) {
    return x -> x * f + b;
}
```

```
static DoubleUnaryOperator curriedConverter( double f, double b ) {
     return x -> x * f + b;
 }

 public static void main( String[] args ) {


     DoubleUnaryOperator convertCtoF = curriedConverter( 9.0 / 5, 32 );

     convertCtoF.applyAsDouble( 35 ); //95 F
     convertCtoF.applyAsDouble( 15 ); //59 F

 }
```

```
static DoubleUnaryOperator curriedConverter( double f, double b ) {
    return x -> x * f + b;
}

public static void main( String[] args ) {

    DoubleUnaryOperator convertCtoF = curriedConverter( 9.0 / 5, 32 );

    convertCtoF.applyAsDouble( 35 ); //95 F

    DoubleUnaryOperator convertKmToMi = curriedConverter( 0.6214, 0 );

    convertKmToMi.applyAsDouble( 804.672 ); //500mi
}
```

```
DoubleUnaryOperator convertBRLtoUSD = curriedConverter( 0.27, 0 );

double usd = convertBRLtoUSD.applyAsDouble( 100 );//27 USD

DoubleUnaryOperator convertUSDtoEUR = curriedConverter( 0.89, 0 );
convertUSDtoEUR.applyAsDouble( usd ); //24.03 EUR

convertBRLtoUSD.andThen( convertUSDtoEUR ).applyAsDouble( 100 );

                //24.03 EUR
```

Design Patterns + Java + Funcional Programming <3

Thank you :)

Eder
Ignatowicz
@ederign

redhat.