CS 525 - ASD

Advanced Software Development

MS.CS Program

Department of Computer Science

Rene de Jong, MsC.

Maharishi University
OF MANAGEMENT

CS 525 - ASD

Advanced Software Development

Maharishi University
OF MANAGEMENT

Lesson 9 Builder pattern

Builder

- Builds a complex object using a step by step approach

Immutable class

- Once created, an immutable object can never be changed

```
public class Money {
  private BigDecimal value;

  public Money(BigDecimal value) {
    this.value = value;
  }

  public Money add(Money money){
    return new Money(value.add(money.getValue()));
  }

  public Money subtract(Money money){
    return new Money(value.subtract(money.getValue()));
  }

  public BigDecimal getValue() {
    return value;
  }
}
```

No setter methods

Mutation leads to the creation of new instances

Why immutable classes?

- Reasons to make a class immutable:
  - Less prone to errors
  - Easier to share
  - Thread safe

- Immutable classes in Java
  - java.lang.String
  - java.io.File
  - java.util.Locale
  - Almost all classes in java.time

Constructor with many parameters

```
public class Customer {
  private String firstName;
  private String lastname;
  private String phone;
  private String email;
  private int age;
  private int numberOfChildren;
  private int shoesize;
  private boolean isMarried;
  private double yearlyIncome;
  private double yearlyAmountSpendOnShoes;

  public Customer(String firstName, String lastname, String phone, String email, int age, int
    numberOfChildren, int shoesize, boolean isMarried, double yearlyIncome, double
    yearlyAmountSpendOnShoes) {
    this.firstName = firstName;
    this.lastname = lastname;
    this.phone = phone;
    this.email = email;
    this.age = age;
    this.numberOfChildren = numberOfChildren;
    this.shoesize = shoesize;
    this.isMarried = isMarried;
    this.yearlyIncome = yearlyIncome;
    this.yearlyAmountSpendOnShoes = yearlyAmountSpendOnShoes;
  }
}
```

```
Customer customer = new Customer("Mary", "Jones", "0623416754",
"mjones@gmail.com", 34, 3, 8, true, 50000.0, 2000.0);
```

Constructor is not expressive

What do these parameters mean?

Easy to make mistakes

If you have optional parameters, you need many constructors

Class can be immutable

Using setters

```
public class ApplicationUsingSetters {
  public static void main(String[] args) {
    Customer customer = new Customer();
    customer.setFirstName("Mary");
    customer.setLastname("Jones");
    customer.setPhone("0623416754");
    customer.setEmail("mjones@gmail.com");
    customer.setAge(34);
    customer.setNumberOfChildren(3);
    customer.setShoesize(8);
    customer.setMarried(true);
    customer.setYearlyIncome(50000.0);
    customer.setYearlyAmountSpendOnShoes(2000.0);
    System.out.println(customer);
  }
}
```

Clear what the parameters mean

Class is not immutable

What if we want

- Expressive code
- Immutable class

- Solution: Builder

Builder example

```
public class Customer {
  private String firstName;
  private String lastname;
  private String phone;
  private String email;
  private int age;
  private int numberOfChildren;
  private int shoesize;
  private boolean isMarried;
  private double yearlyIncome;
  private double yearlyAmountSpendOnShoes;

  public static class Builder {

    private String firstName="";
    private String lastname="";
    private String phone="";
    private String email="";
    private int age = 0;
    private int numberOfChildren = 0;
    private int shoesize = 0;
    private boolean isMarried = false;
    private double yearlyIncome = 0.0;
    private double yearlyAmountSpendOnShoes = 0.0;

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

Builder inner class

'Setter' method on the builder

Return 'this' for method chaining

Builder example

```java
public Builder withLastname(String lastname) {
  this.lastname = lastname;
  return this;
}
public Builder withPhone(String phone) {
  this.phone = phone;
  return this;
}
public Builder withEmail(String email) {
  this.email = email;
  return this;
}
public Builder withAge(int age) {
  this.age = age;
  return this;
}
public Builder withNumberOfChildren(int numberOfChildren) {
  this.numberOfChildren = numberOfChildren;
  return this;
}
public Builder withShoesize(int shoesize) {
  this.shoesize = shoesize;
  return this;
}
public Builder isMarried() {
  this.isMarried = true;
  return this;
}
```

Builder example

```
  public Builder isNotMarried() {
    this.isMarried = false;
    return this;
  }
  public Builder withYearlyIncome(double yearlyIncome) {
    this.yearlyIncome = yearlyIncome;
    return this;
  }
  public Builder withYearlyAmountSpendOnShoes(double yearlyAmountSpendOnShoes) {
    this.yearlyAmountSpendOnShoes = yearlyAmountSpendOnShoes;
    return this;
  }

  public Customer build() {
    return new Customer(this);
  }
}
```

The build() method does the actual creation of the object

Builder example

```
private Customer(Builder builder) {
  this.firstName = builder.firstName;
  this.lastname = builder.lastname;
  this.phone = builder.phone;
  this.email = builder.email;
  this.age = builder.age;
  this.numberOfChildren = builder.numberOfChildren;
  this.shoesize = builder.shoesize;
  this.isMarried = builder.isMarried;
  this.yearlyIncome = builder.yearlyIncome;
  this.yearlyAmountSpendOnShoes = builder.yearlyAmountSpendOnShoes;
}

@Override
public String toString() {
  return "Customer [firstName=" + firstName + ", lastname=" + lastname + ", phone=" + phone + ",
      email=" + email + ", age=" + age + ", numberOfChildren=" + numberOfChildren + ", shoesize="
      + shoesize + ", isMarried="+ isMarried + ", yearlyIncome=" + yearlyIncome + ",
      yearlyAmountSpendOnShoes=“ + yearlyAmountSpendOnShoes + "]";
}

}
```

The constructor has a Builder as argument

The client code

```
public class Application {

  public static void main(String[] args) {
    Customer customer1 = new Customer.Builder()
      .withFirstName("Mary")
      .withLastname("Jones")
      .withEmail("mjones@gmail.com")
      .withAge(34)
      .isMarried()
      .withNumberOfChildren(3)
      .withPhone("0623416754")
      .withShoesize(8)
      .withYearlyIncome(50000.0)
      .withYearlyAmountSpendOnShoes(2000.0)
      .build();
    System.out.println(customer1);

    Customer customer2 = new Customer.Builder()
      .withFirstName("Lucy")
      .withLastname("Jhonson")
      .isNotMarried()
      .withPhone("0698345234")
      .build();
    System.out.println(customer2);
  }
}
```

Clear code

Customer is immutable

Builder used in Quartz

```java
SchedulerFactory schedFact = new StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
// define the job and tie it to our HelloJob class
JobDetail job = JobDetail("myJob", "group1", HelloJob.class);

// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger(("myTrigger", "group1", new Date(), null,
       SimpleTrigger.REPEAT_INDEFINITELY, 40)

// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

Quartz 1.0

```java
SchedulerFactory schedFact = new StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1")
    .build();
// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger()
    .withIdentity("myTrigger", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(40)
        .repeatForever())
    .build();
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

Quartz 2.0

Main point

- The builder pattern is a great help if you want to create objects with many different parameters.
- All the intelligence of Nature is available at the level of the Unified Field