Builder Pattern

- HARSHIL SHAH
- NITHIN V
- VINOD ADWANI
- YASHASWA JAIN

Creating a Class (Using setters, getters)

```java
public class User {
    private String firstName; // required
    private String lastName; // required
    private int age; // optional
    private String phone; // optional
    private String address;  //optional

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
}
```

Problem with this Approach

- Leads to inconsistent states.
- Client can just create an empty object and then set only the attributes that he/she is interested in.
- Object will not have a complete state until all the *setX* methods have been invoked.
- This approach makes the *User* class mutable.

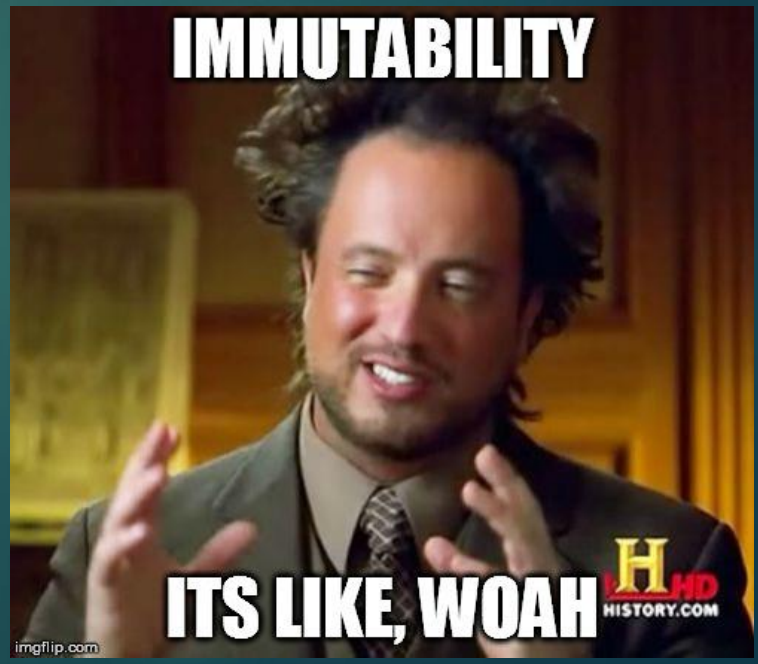
Immutable Classes are cool!

```
1  public class User {
2      private final String firstName;     //required
3      private final String lastName;     //required
4      private final int age;     //optional
5      private final String phone;     //optional
6      private final String address;     //optional
7  ...
8  }
```

- Always make immutable classes!
  Unless there's a really good reason not to do so.

Why immutable?

- Immutable classes are faster (can be cached!) and safer.
- Simplicity - each class is in one fixed state only.
- Thread Safe - because the state cannot be changed, no synchronization is required.



Second Approach (and it works!)

```java
public User(String firstName, String lastName) {
    this(firstName, lastName, 0);
}

public User(String firstName, String lastName, int age) {
    this(firstName, lastName, age, '');
}

public User(String firstName, String lastName, int age, String phone) {
    this(firstName, lastName, age, phone, '');
}

public User(String firstName, String lastName, int age, String phone, String
address) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.phone = phone;
    this.address = address;
}
```

Problems!

- As number of attributes increases, the code becomes harder to read and maintain.
- Constructors = 2^(number of attributes) :('


DESIGN PATTERNS
TO THE RESCUE
imgflip.com

Builder Pattern

```
public class User {
    private final String firstName; // required
    private final String lastName; // required
    private final int age; // optional
    private final String phone; // optional
    private final String address; // optional

    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public int getAge() {
        return age;
    }

    public String getPhone() {
        return phone;
    }

    public String getAddress() {
        return address;
    }
```

```
    public static class UserBuilder {
        private final String firstName;
        private final String lastName;
        private int age;
        private String phone;
        private String address;

        public UserBuilder(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

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

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

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

        public User build() {
            return new User(this);
        }

    }
}
```

Points to note:

- The User constructor is private
- The class is immutable.
- The builder constructor only receives the "required" attributes and all the required attributes need to be final in Builder class.

Object in one Line!

```
1  public User getUser() {
2      return new
3              User.UserBuilder('Jhon', 'Doe')
4              .age(30)
5              .phone('1234567')
6              .address('Fake address 1234')
7              .build();
8  }
```

Advantages:

- Build a object in 1 line of code.
- Easier to read and write.
- A single builder can be used to create multiple objects (Not possible in other languages).
- A builder could be passed to a method to enable this method to create one or more objects. The method need not know any kind of details about how the objects are created.

Summary

- Excellent choice for classes with lots of parameters (especially parameters are optional)!
- Easier to read, write and maintain.
- Classes can remain immutable, makes your code safer.

Builder Pattern Dobaara

- HARSHIL SHAH
- NITHIN V
- VINOD ADWANI
- YASHASWA JAIN

Builder Pattern

- Object creation software design pattern
- The intention to find a solution when the increase of object constructor parameter combinations leads to an "**exponential list of constructors".**
- It uses another object, a builder, that receives each initialization parameter step by step and then returns the constructed object at once.

Definition

- The intent of the Builder design pattern is to separate the construction of a complex object from its representation.
- By doing so the same construction process can create different representations

Components

Builder

- An abstract interface for creating parts of a Product object.

Concrete Builder

- Implements the builder interface.
- Defines and keeps track of the representation it creates.

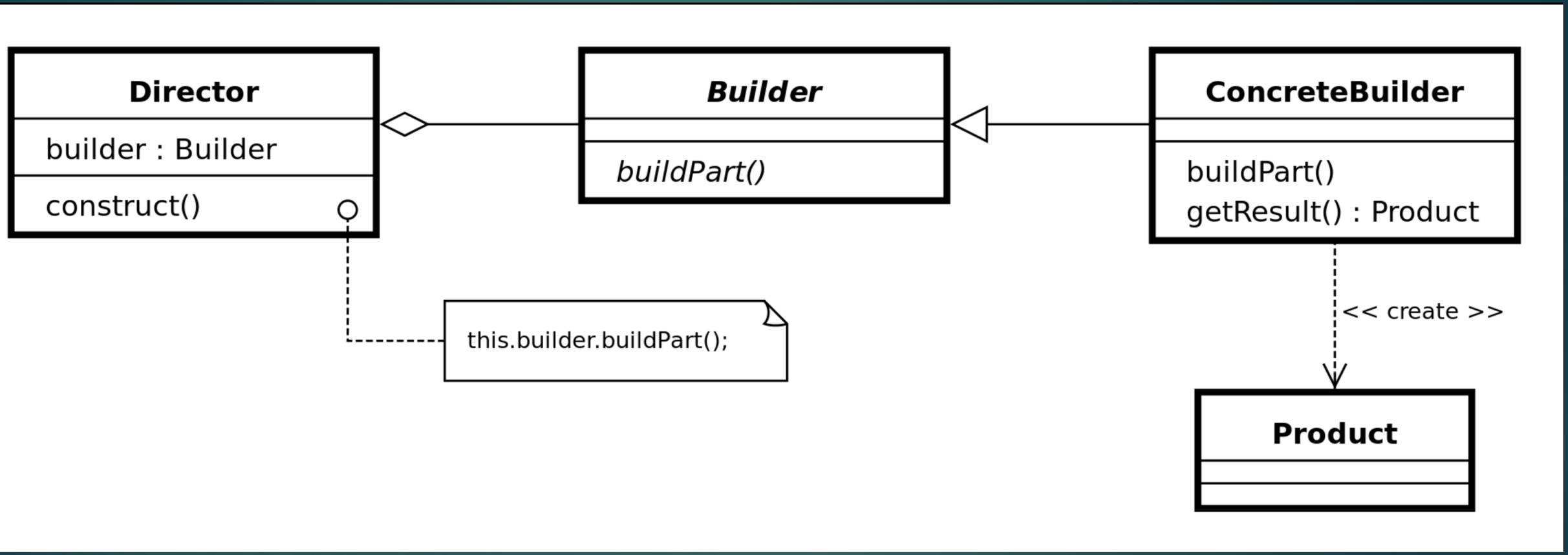
Director

- Uses the Concrete Builder and constructs the object.

Product

- The object that is being built.

Structure


Director
builder : Builder
construct()
Builder
buildPart()
ConcreteBuilder
buildPart()
getResult() : Product
this.builder.buildPart();
<< create >>
Product

Advantages

- Provides control over steps of construction process.
- Encapsulates code for construction and representation.
- Allows you to vary a product's internal representation.

Disadvantages

- Requires creating a separate ConcreteBuilder for each different type of Product (Not a disadvantage in Java!)
- Requires the builder classes to be mutable.
- Introduces a lot of code, bigger problem if builder pattern is misused.