

Creating Intuitive APIs With The Builder Design Pattern

Kelly
Morrison

```
package com.morrison.builders;
public class SimpleName {
    private final String first, last;
    private SimpleName(final String first, final String last) {
        this.first = first;
        this.last = last;
    }
    public String getFirst() {
        return this.first;
    }
    public String getLast() {
        return this.last;
    }
}
```



The CodeMash “Law of Two Feet”

If a session just isn't doing it for you, don't feel obligated to stay.

PDF of slides



Kelly Morrison, Ph.D. Computer Engineering



WHAT IS...

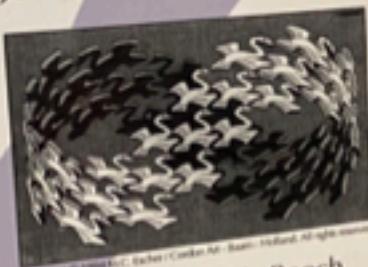


A BUILDER PATTERN?

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



PATTERN HATCHING

Design Patterns Applied



JOHN VLASSIDES
Foreword by James O. Coplien

SOFTWARE PATTERNS SERIES

BUILDER

Object Creational

Intent

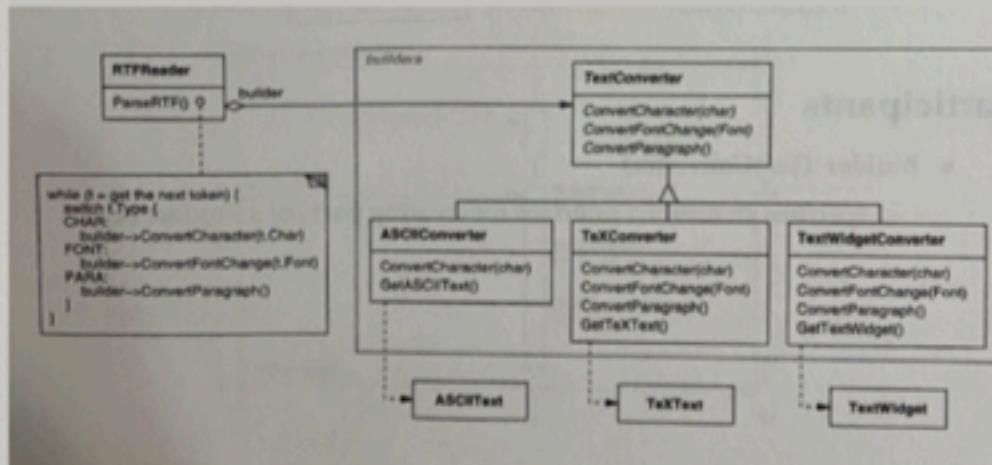
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Motivation

A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats. The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively. The problem, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.

A solution is to configure the RTFReader class with a TextConverter object that converts RTF to another textual representation. As the RTFReader parses the RTF document, it uses the TextConverter to perform the conversion. Whenever the RTFReader recognizes an RTF token (either plain text or an RTF control word), it issues a request to the TextConverter to convert the token. TextConverter objects are responsible both for performing the data conversion and for representing the token in a particular format.

Subclasses of TextConverter specialize in different conversions and formats. For example, an ASCIIConverter ignores requests to convert anything except plain text. A TeXConverter, on the other hand, will implement operations for all requests in order to produce a TeX representation that captures all the stylistic information in the text. A TextWidgetConverter will produce a complex user interface object that lets the user see and edit the text.





A photograph of Michael Scott from the TV show "The Office". He is wearing a dark suit and has his right hand resting against his chin in a thoughtful pose. He is looking slightly upwards and to the left with a confused or questioning expression. The background shows an office environment with a window featuring horizontal blinds.

**Why don't you explain
this to me like I'm five?**



**A Builder is basically
a better constructor.**

**You get a Builder, give it
some data, and ask it
to construct ("build")
an object for you**



**“Forget the
academic talk.
Show me some
real life
examples.”**



```
String name = new StringBuilder()  
    .append("First name: ")  
    .append("Wade")  
    .append(", Last name: ")  
    .append("Wilson")  
    .append(", Code name: ")  
    .append("Deadpool")  
    .toString();
```

```
Stream<String> stream = Stream.<String>builder()  
    .add("one")  
    .add("two")  
    .add("three")  
    .build();
```



```
String name = new StringBuilder()  
    .append("First name: ")  
    .append("Wade")  
    .append(", Last name: ")  
    .append("Wilson")  
    .append(", Code name: ")  
    .append("Deadpool")  
    .toString();
```

```
Stream<String> stream = Stream.<String>builder()  
    .add("one")  
    .add("two")  
    .add("three")  
    .build();
```



```
WebClient webClient = WebClient.builder()
    .baseUrl("https://api.example.com")
    .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
    .defaultCookie("cookieKey", "cookieValue")
    .build();
```

```
RestTemplate restTemplate = new RestTemplateBuilder()
    .rootUri("https://api.example.com")
    .basicAuthentication("user", "password")
    .setConnectTimeout(Duration.ofSeconds(5))
    .build();
```

```
UriComponents uriComponents = UriComponentsBuilder.newInstance()
    .scheme("https")
    .host("example.com")
    .path("/api/resource")
    .queryParam("param1", "value1")
    .queryParam("param2", "value2")
    .build();
```



```
WebClient webClient = WebClient.builder()
    .baseUrl("https://api.example.com")
    .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
    .defaultCookie("cookieKey", "cookieValue")
    .build();
```

```
RestTemplate restTemplate = new RestTemplateBuilder()
    .rootUri("https://api.example.com")
    .basicAuthentication("user", "password")
    .setConnectTimeout(Duration.ofSeconds(5))
    .build();
```

```
UriComponents uriComponents = UriComponentsBuilder.newInstance()
    .scheme("https")
    .host("example.com")
    .path("/api/resource")
    .queryParam("param1", "value1")
    .queryParam("param2", "value2")
    .build();
```



```
ImmutableList<String> list = ImmutableList.<String>builder()
    .add("one")
    .add("two")
    .add("three")
    .build();
```

```
Cache<String, String> cache = CacheBuilder.newBuilder()
    .maximumSize(100)
    .expireAfterWrite(10, TimeUnit.MINUTES)
    .build();
```



```
ImmutableList<String> list = ImmutableList.<String>builder()
    .add("one")
    .add("two")
    .add("three")
    .build();
```

```
Cache<String, String> cache = CacheBuilder.newBuilder()
    .maximumSize(100)
    .expireAfterWrite(10, TimeUnit.MINUTES)
    .build();
```



The details may differ,
but the idea is the same.

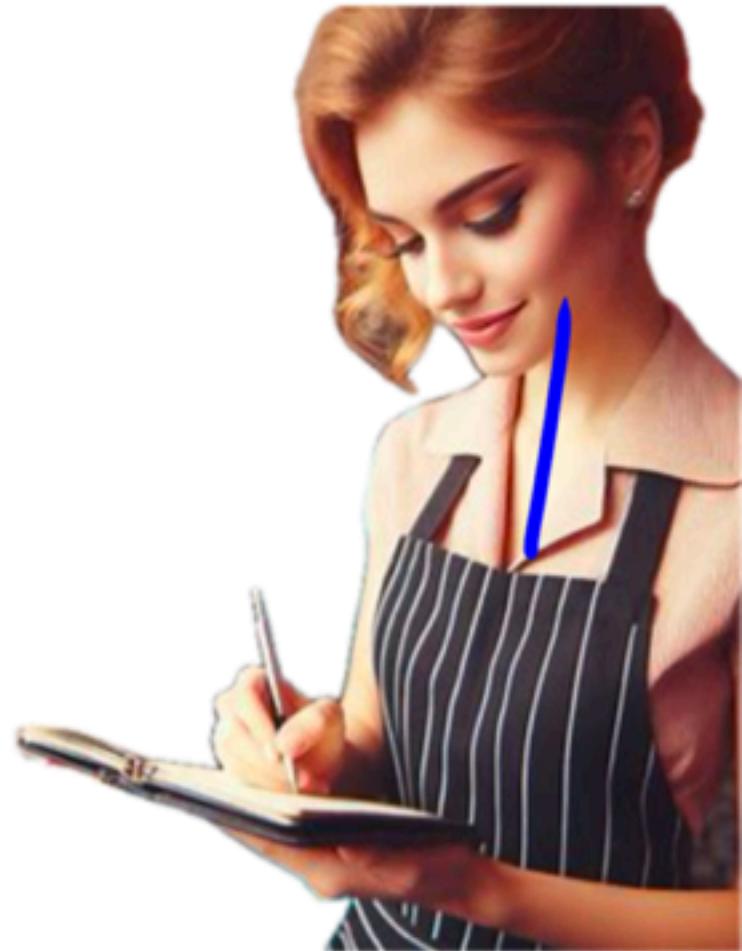
Get a builder.
Give them instructions.
Tell them to build it.

I like to think of
a Builder as a
waiter in a
diner.



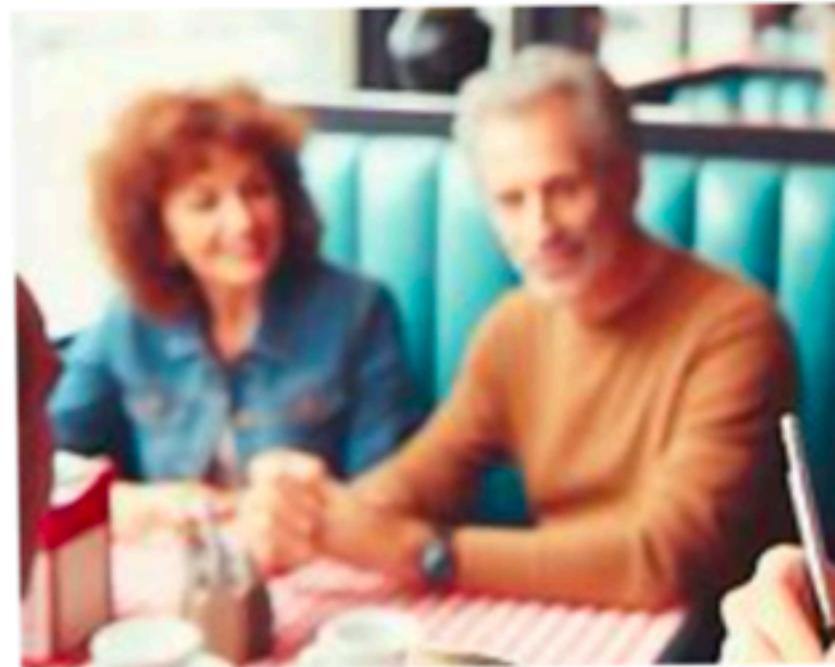
The waiter is a Builder.

They have a notepad for taking orders.



The customers give orders to the waiter.

Things can easily be changed at this point.



The waiter takes the order to the chef.

The chef builds the meal.

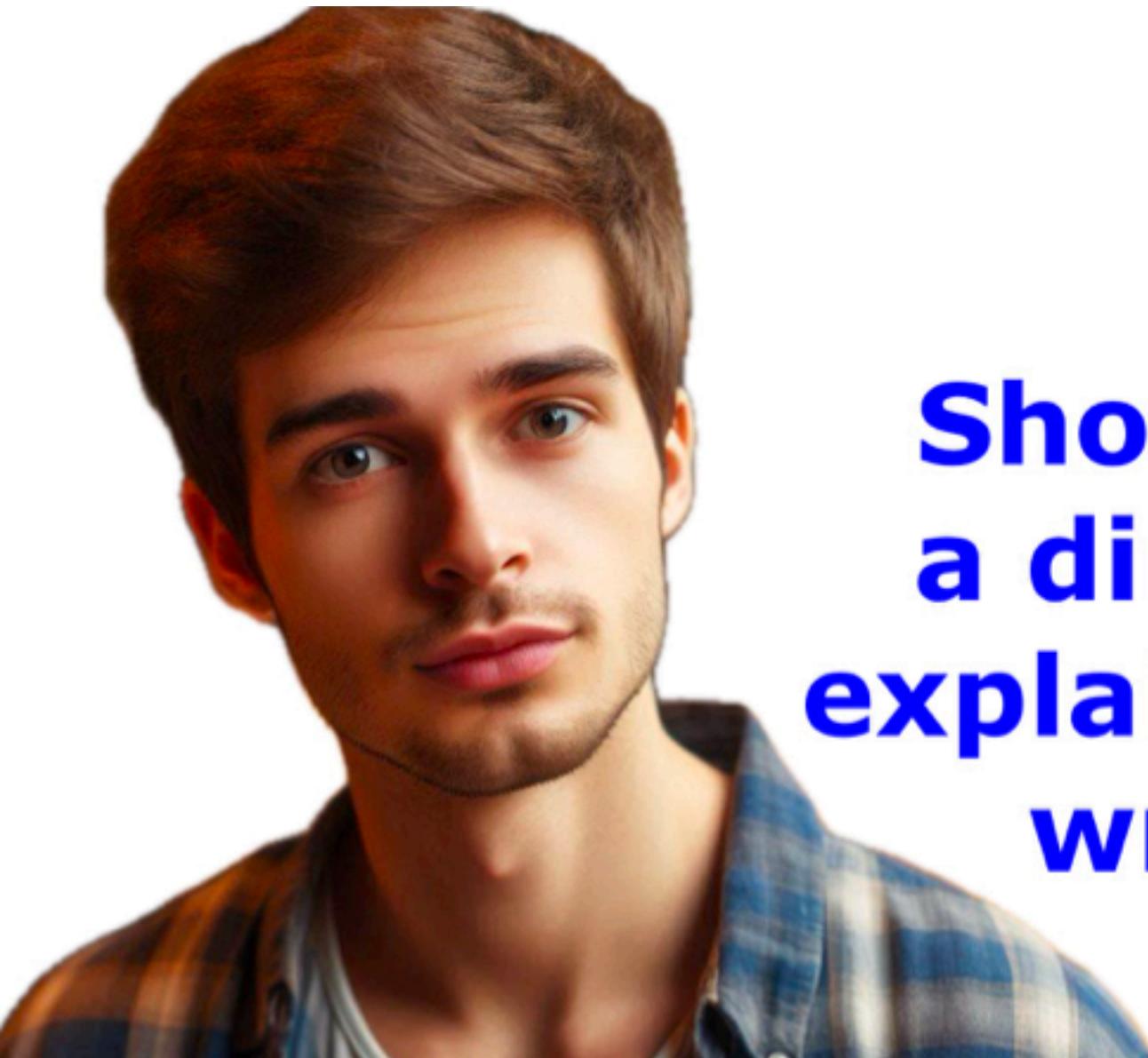
The waiter checks the meal for problems: missing items, food at wrong temperature, etc.

The meal is delivered to the customer.



If you understood that, you know
how Builder works!





**"What?
Show me how
a diner dinner
explains how to
write code."**





**Let's look under
the hood
and see how
a Builder
is implemented.**

```
public static final void main(final String[] args) {  
    SimpleName me = new SimpleNameBuilder()  
        .setFirstName("Dua")  
        .setLastName("Lipa")  
        .build();  
  
    System.out.println(me);  
}
```

```
public class SimpleName {  
  
    private final String first, last;  
  
    private SimpleName(final String first, final String last) { ...  
  
        public String getFirst() { ...  
  
        public String getLast() { ...
```

```
public class SimpleName {  
  
    private final String first, last;  
  
    private SimpleName(final String first, final String last) {...  
  
    public String getFirst() {...  
  
    public String getLast() {...  
  
    public static class SimpleNameBuilder {  
        private String first, last;  
  
        public SimpleNameBuilder() {...  
  
        public SimpleNameBuilder setFirstName(final String first) {...  
  
        public SimpleNameBuilder setLastName(final String last) {...  
  
        public SimpleName build() {...  
    }
```

```
public class SimpleName {  
  
    private final String first, last;  
  
    private SimpleName(final String first, final String last) {  
        this.first = first;  
        this.last = last;  
    }  
  
    public String getFirst() {  
        return this.first;  
    }  
  
    public String getLast() {  
        return this.last;  
    }  
}  
  
public static class SimpleNameBuilder {  
    private String first, last;  
  
    public SimpleNameBuilder() {  
        this.first = "";  
        this.last = "";  
    }  
  
    public SimpleNameBuilder setFirstName(final String first) {  
        this.first = first;  
        return this;  
    }  
  
    public SimpleNameBuilder setLastName(final String last) {  
        this.last = last;  
        return this;  
    }  
  
    public SimpleName build() {  
        return new SimpleName(first, last);  
    }  
}
```

```
public class SimpleName {  
  
    private final String first, last;  
  
    private SimpleName(final String first, final String last) {  
        this.first = first;  
        this.last = last;  
    }  
  
    public String getFirst() {  
        return this.first;  
    }  
  
    public String getLast() {  
        return this.last;  
    }  
}
```



```
public static class SimpleNameBuilder {  
    private String first, last;  
  
    public SimpleNameBuilder() {  
        this.first = "";  
        this.last = "";  
    }  
  
    public SimpleNameBuilder setFirstName(final String first) {  
        this.first = first;  
        return this;  
    }  
  
    public SimpleNameBuilder setLastName(final String last) {  
        this.last = last;  
        return this;  
    }  
  
    public SimpleName build() {  
        return new SimpleName(first, last);  
    }  
}
```





Hold on. Are we being
“architecture astronauts”?

What's wrong with plain old getters and setters?

```
public class SimplerName {  
  
    private String first, last;  
  
    public SimplerName() {  
        this.first = "";  
        this.last = "";  
    }  
  
    public void setFirstName(final String firstName) {  
        this.first = firstName;  
    }  
  
    public void setLastName(final String lastName) {  
        this.last = lastName;  
    }  
  
    public String getFirstName() {  
        return this.first;  
    }  
  
    public String getLastName() {  
        return this.last;  
    }  
}
```

It's mutable.

**It can be changed elsewhere
in the program.**

**This can cause problems with
multithreaded code.**

```
public class BestPicture {  
    private String film;  
  
    public synchronized void setFilm(String film) {  
        this.film = film;  
    }  
  
    public synchronized String getFilm() {  
        return film;  
    }  
}
```

```
public class BestPicture {
    private String film;

    public synchronized void setFilm(String film) {
        this.film = film;
    }

    public synchronized String getFilm() {
        return film;
    }

    public static void main(String[] args) {
        BestPicture bestPicture = new BestPicture();

        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 30; i++) {
                bestPicture.setFilm("The Avengers");
                try {
                    Thread.sleep(500); // Sleep for 0.5 seconds
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });

        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 30; i++) {
                bestPicture.setFilm("Justice League");
                try {
                    Thread.sleep(500); // Sleep for 0.5 seconds
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
    }

    thread1.start();
    thread2.start();

    try {
        thread1.join();
        thread2.join();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }

    System.out.println("Final value of film: " + bestPicture.getFilm());
}
```

I ran it 10 times.

Final value of film: The Avengers

Final value of film: Justice League

Final value of film: The Avengers

Final value of film: The Avengers

Fine. Let's make it immutable.



```
public class BestPicture {  
    private final String film;  
  
    public BestPicture(final String film) {  
        this.film = film;  
    }  
  
    public synchronized String getFilm() {  
        return film;  
    }  
}
```

Ok, it's immutable.

Are you happy now?



**There are still
many problems
with
constructors.**



Long, confusing constructors

```
public class Employee {  
  
    private final String firstName, middleName, lastName, socialSecurity,  
        title, department, birthday, salary;  
  
    public Employee(String firstName, String middleName, String lastName,  
        String socialSecurity, String title,  
        String department, String birthday, String salary) {  
        this.firstName = firstName;  
        this.middleName = middleName;  
        this.lastName = lastName;  
        this.socialSecurity = socialSecurity;  
        this.title = title;  
        this.department = department;  
        this.birthday = birthday;  
        this.salary = salary;  
    }  
  
    public static void main(String[] args) {  
        Employee employee = new Employee("Tony", "Stark", null, null,  
            "Avenger", null, null, null)  
    }  
}
```

**It's possible to
create an object in
an inconsistent or
invalid state.**

```
public class Fraction {  
  
    private int numerator, denominator;  
  
    public Fraction() {  
        this.numerator = 0;  
        this.denominator = 0;  
    }  
  
    public void setNumerator(final int numerator) {  
        this.numerator = numerator;  
    }  
  
    public void setDenominator(final int denominator) {  
        this.denominator = denominator;  
    }  
  
    public int asDecimal() {  
        return this.numerator / this.denominator;  
    }  
  
    public static void main(String[] args) {  
        Fraction fraction = new Fraction();  
        fraction.setNumerator(50);  
        System.out.println(fraction.asDecimal());  
    }  
}
```

Too many setters can confuse a programmer.

```
public class PaymentMethod {  
    // Credit card  
    public void setCreditCardNumber(String number) {}  
    public void setCreditCardExpirationDate(String date) {}  
    public void setCvv(String cvv) {}  
    // Gift card  
    public void setGiftCardNumber(String number) {}  
    public void setGiftCardAmount(String amount) {}  
    public void setVendor(String vendor) {}  
    // Cash  
    public void setDollars(int dollars) {}  
    public void setCents(int cents) {}  
  
Run | Debug  
public static void main(final String[] args) {  
  
    PaymentMethod pm = new PaymentMethod();  
    pm.setCreditCardNumber("1234 5678 9012 3456");  
    pm.setVendor("Starbucks");  
    pm.setDollars(50);  
    pm.setCents(0);  
}
```



**The Builder
pattern can fix
these problems!**



**Builder lets us avoid passing unneeded values.
It reduces the possibility of getting a value wrong.**

```
public static void main(String[] args) {  
    Employee employee = new Employee("Tony", "Stark", null, null,  
        "Avenger", null, null, null);  
  
    Employee employee2 = new Employee.EmployeeBuilder()  
        .firstName("Tony")  
        .lastName("Stark")  
        .title("Avenger")  
        .build();  
}
```

```
public static class EmployeeBuilder {
    private String firstName, middleName, lastName, socialSecurity,
    title, department, birthday, salary;

    public EmployeeBuilder() {
        this.firstName = "";
        this.middleName = "";
        this.lastName = "";
        this.socialSecurity = "";
        this.title = "";
        this.department = "";
        this.birthday = "";
        this.salary = "";
    }

    public EmployeeBuilder firstName(final String firstName) {
        this.firstName = firstName;
        return this;
    }

    public EmployeeBuilder middleName(final String middleName) {
        this.middleName = middleName;
        return this;
    }

    public EmployeeBuilder lastName(final String lastName) {
        this.lastName = lastName;
        return this;
    }

    public EmployeeBuilder socialSecurity(final String socialSecurity) {
        this.socialSecurity = socialSecurity;
        return this;
    }

    public EmployeeBuilder title(final String title) {
        this.title = title;
        return this;
    }
}
```

```
public EmployeeBuilder department(final String department) {
    this.department = department;
    return this;
}

public EmployeeBuilder birthday(final String birthday) {
    this.birthday = birthday;
    return this;
}

public EmployeeBuilder salary(final String salary) {
    this.salary = salary;
    return this;
}

public Employee build() {
    return new Employee(firstName, middleName, lastName,
        socialSecurity, title, department, birthday, salary);
}
```

Builder prevents building invalid objects.

```
public static class FractionBuilder {
    private int numerator = 0, denominator = 0;

    public FractionBuilder() {}

    public FractionBuilder numerator(final int numerator) {
        this.numerator = numerator;
        return this;
    }
    public FractionBuilder denominator(final int denominator) {
        this.denominator = denominator;
        return this;
    }
    public Fraction build() {
        if (this.denominator == 0) {
            throw new IllegalArgumentException("denominator must not be 0");
        }
        return new Fraction(numerator, denominator);
    }
}

public static void main(String[] args) {
    // Old method: doesn't throw an exception until asDecimal() is called.
    Fraction fraction = new Fraction();
    fraction.setNumerator(50);
    System.out.println(fraction.asDecimal());

    // New method: will not build a fraction with a non-zero denominator.
    Fraction fraction2 = new FractionBuilder().numerator(50).build();
}
```

And now for something that will blow your mind - multiple Builders for a class!



But... why?



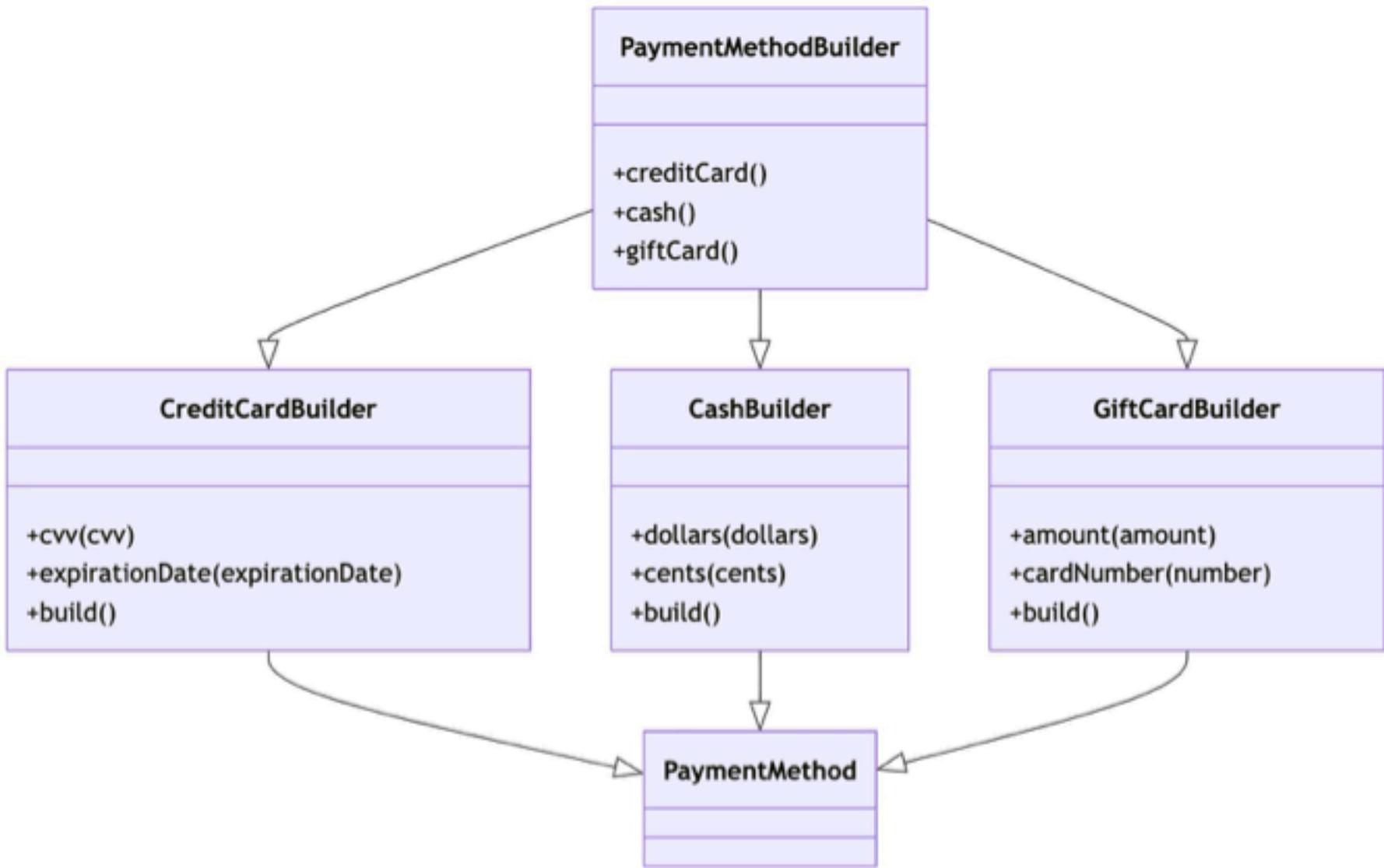
Remember this class?

Some of the options only make sense in relation to certain others.

```
public class PaymentMethod {  
    // Credit card  
    public void setCreditCardNumber(String number) {}  
    public void setCreditCardExpirationDate(String date) {}  
    public void setCvv(String cvv) {}  
    // Gift card  
    public void setGiftCardNumber(String number) {}  
    public void setGiftCardAmount(String amount) {}  
    public void setVendor(String vendor) {}  
    // Cash  
    public void setDollars(int dollars) {}  
    public void setCents(int cents) {}  
  
    Run | Debug  
    public static void main(final String[] args) {  
  
        PaymentMethod pm = new PaymentMethod();  
        pm.setCreditCardNumber("1234 5678 9012 3456");  
        pm.setVendor("Starbucks");  
        pm.setDollars(50);  
        pm.setCents(0);  
    }  
}
```



**Let's use
Builders
to fix it!**



We use one Builder to select a more specialized Builder.

```
public class PaymentMethod {  
    private final PaymentMethodType paymentMethodType;  
    private final String creditCardNumber, cvv, expirationDate;  
    private final String cash;  
    private final String giftCardNumber, couponNumber;  
  
    private PaymentMethod(PaymentMethodType paymentMethodType, String creditCardNumber,  
        String cvv, String expirationDate, String cash, String giftCardNumber, String couponNumber) {  
        // ... set values ...  
    }  
  
    public class PaymentMethodBuilder {  
        public PaymentMethodBuilder() {  
        }  
        public CreditCardBuilder creditCard() {  
            return new CreditCardBuilder();  
        }  
        public CashBuilder cash() {  
            return new CashBuilder();  
        }  
        public GiftCardBuilder giftCard() {  
            return new GiftCardBuilder();  
        }  
    }  
}
```

Here's what the credit card builder could look like.

```
public class CreditCardBuilder {  
    private String creditCardNumber = "", cvv = "", expirationDate = "";  
    public CreditCardBuilder() {}  
    public CreditCardBuilder creditCardNumber(final String creditCardNumber) {  
        this.creditCardNumber = creditCardNumber;  
        return this;  
    }  
    public CreditCardBuilder cvv(final String cvv) {  
        this.cvv = cvv;  
        return this;  
    }  
    public CreditCardBuilder expirationDate(final String expirationDate) {  
        this.expirationDate = expirationDate;  
        return this;  
    }  
    public PaymentMethod build() {  
        return new PaymentMethod(PaymentMethodType.CREDIT_CARD,  
            creditCardNumber, cvv, expirationDate, null, null, null);  
    }  
  
    public class CashBuilder {  
        public PaymentMethod build() {-  
    }  
  
    public class GiftCardBuilder {  
        public PaymentMethod build() {-  
    }  
}
```

```
PaymentMethod pay1 = new PaymentMethodBuilder()  
    .creditCard()  
    .cvv("1234")  
    .expirationDate("12/31/2028")  
    .build();
```

-> PaymentMethodBuilder
-> CreditCardBuilder
-> CreditCardBuilder
-> CreditCardBuilder
-> PaymentMethod


```
PaymentMethod pay2 = new PaymentMethodBuilder()  
    .giftCard()  
    .amount("150.00")  
    .cardNumber("12345678901234")  
    .build();
```

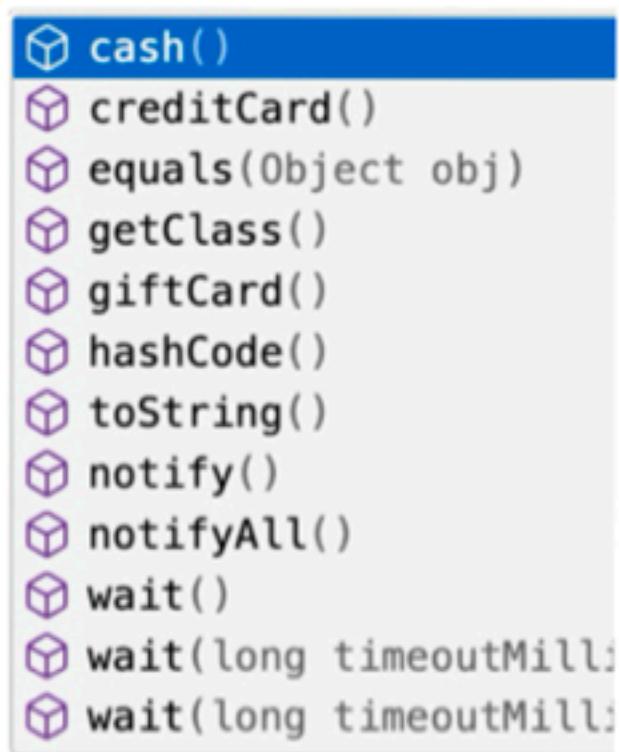
-> PaymentMethodBuilder
-> GiftCardBuilder
-> GiftCardBuilder
-> GiftCardBuilder
-> PaymentMethod


```
PaymentMethod pay3 = new PaymentMethodBuilder()  
    .cash()  
    .dollars(50)  
    .cents(0)  
    .build();
```

-> PaymentMethodBuilder
-> CashBuilder
-> CashBuilder
-> CashBuilder
-> PaymentMethod

Now you only see the types of payment method at first.

```
paymentMethod3 = new PaymentMethodBuilder().
```



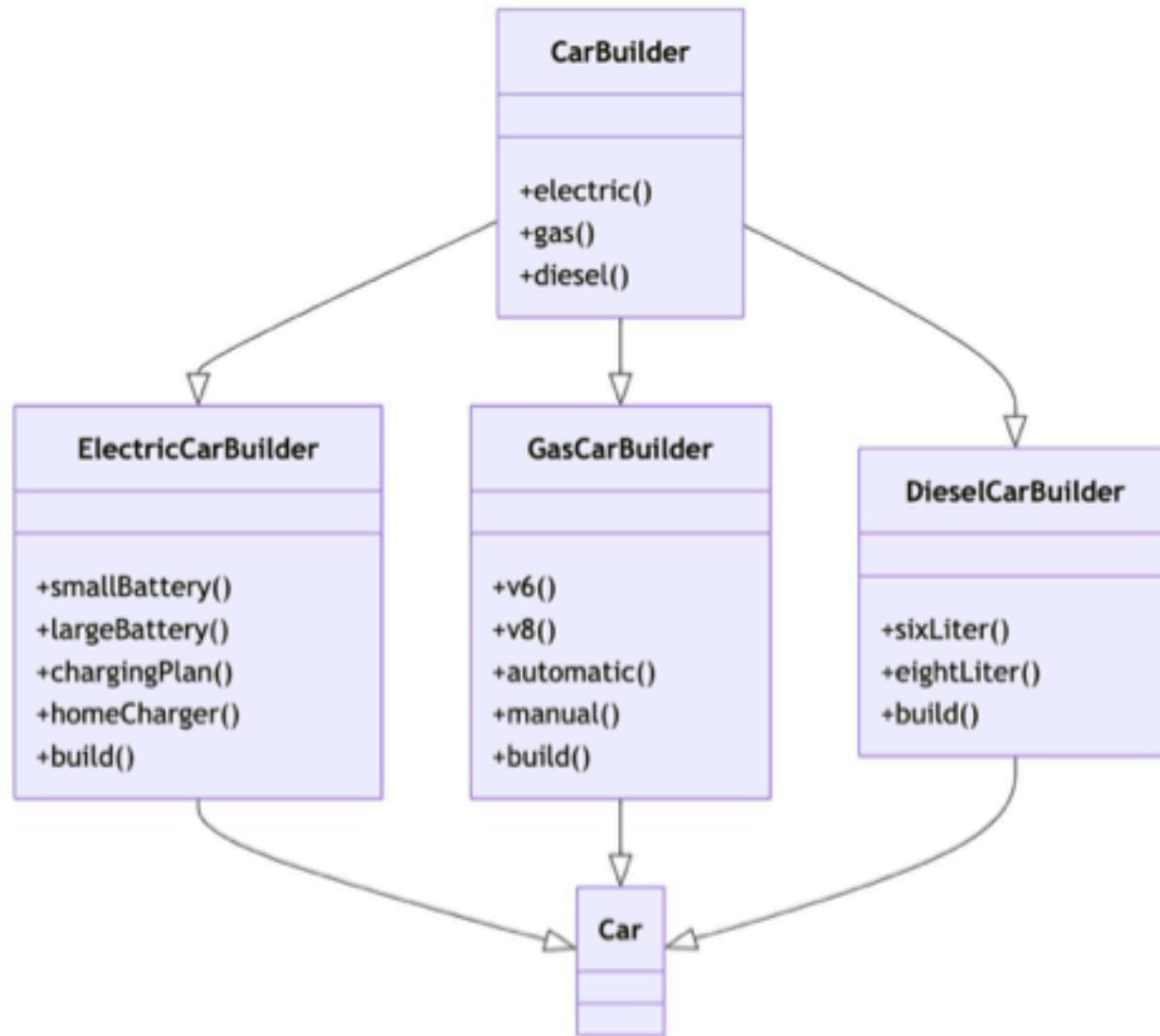
This works beautifully with IDEs!

```
PaymentMethod paymentMethod3 = new PaymentMethodBuilder()  
    .creditCard()
```

build()	PaymentMethod
creditCardNumber	String
cvv	String
expirationDate	String
creditCardNumber(String creditCardNumb...	CreditCardBui...
cvv(String cvv)	CreditCardBuilder
equals(Object obj)	boolean
expirationDate(String expirationDate)	CreditCardBuilder
getClass()	Class<?>
hashCode()	int
toString()	String
notify()	void



**Let's look at some more
multiple builder
examples**



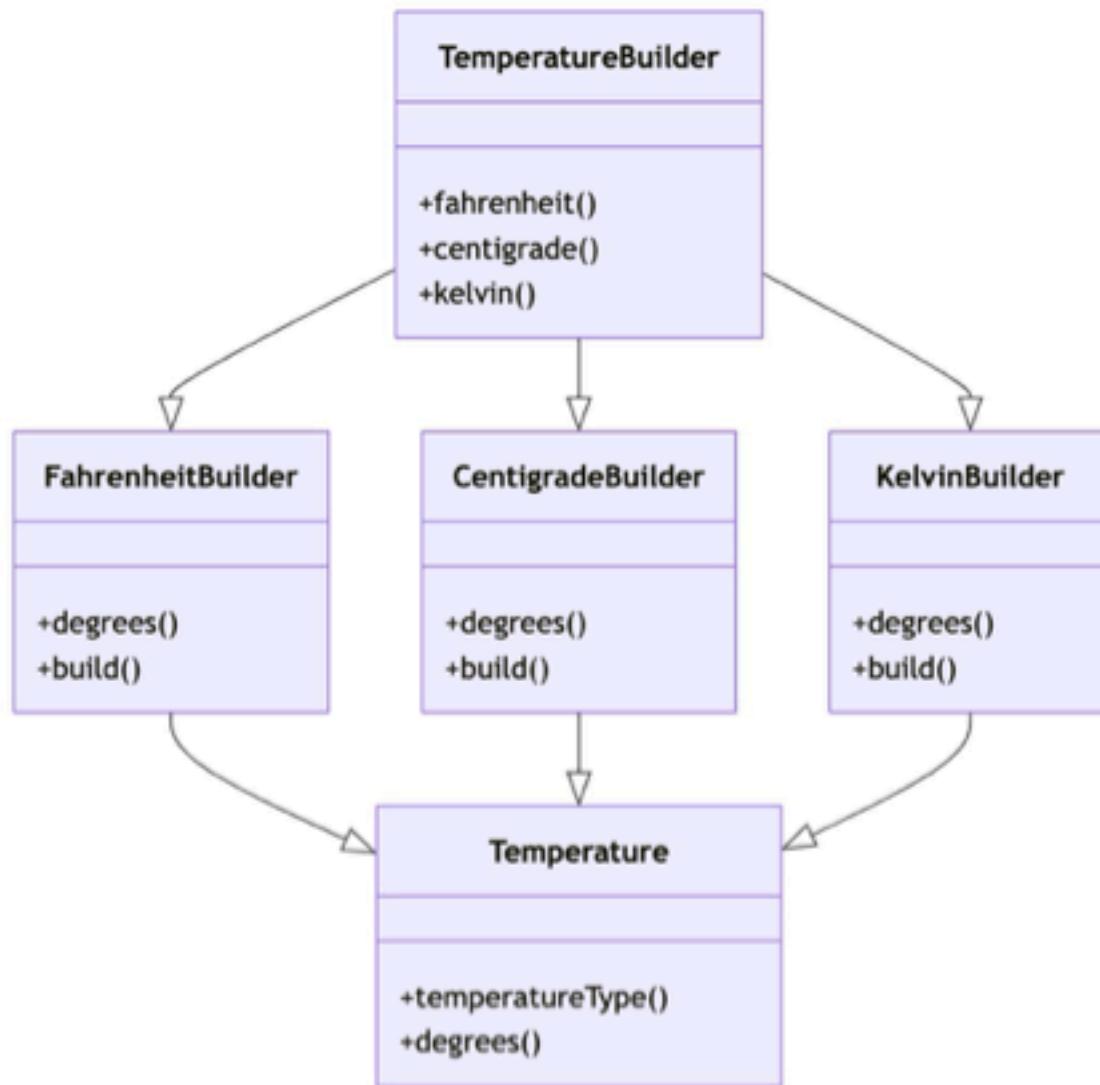
```
Car tesla = new CarBuilder()  
    .electric()  
    .largeBattery()  
    .homeCharger()  
    .build();
```

```
Car dadsCar = new CarBuilder()  
    .gas()  
    .v8()  
    .manual()  
    .build();
```

```
Car workTruck = new CarBuilder()  
    .diesel()  
    .eightLiter()  
    .build();
```

Here is the outline of the code.

```
public class Car {  
    public static CarBuilder builder() {  
        return new CarBuilder();  
    }  
  
    private static final class CarBuilder {  
        public ElectricCarBuilder electric() {  
            return new ElectricCarBuilder();  
        }  
        public GasCarBuilder gas() {-  
            public DieselCarBuilder diesel() {-  
        }  
  
        public static final class ElectricCarBuilder {  
            public ElectricCarBuilder smallBattery() {  
                return this;  
            }  
            public ElectricCarBuilder largeBattery() {-  
                public ElectricCarBuilder chargingPlan() {-  
                    public ElectricCarBuilder homeCharger() {-  
                        public Car build() {  
                            return new Car();  
                        }  
                    }  
                }  
            }  
        }  
  
        public static final class GasCarBuilder {  
            public GasCarBuilder v6() {-  
                public GasCarBuilder v8() {-  
                    public GasCarBuilder automatic() {-  
                        public GasCarBuilder manual() {-  
                            public Car build() {-  
                        }  
                    }  
                }  
            }  
        }  
  
        public static final class DieselCarBuilder {  
            public DieselCarBuilder sixLiter() {-  
                public DieselCarBuilder eightLiter() {-  
                    public Car build() {-  
                }  
            }  
        }  
    }  
}
```



```
Temperature temp1 = new TemperatureBuilder()
    .fahrenheit()
    .degrees(98)
    .build();

Temperature temp2 = new TemperatureBuilder()
    .centigrade()
    .degrees(17)
    .build();

Temperature temp3 = new TemperatureBuilder()
    .kelvin()
    .degrees(0)
    .build();
```

Let's look at some enhancements to the Builder pattern.

First, we'll hide the name of the Builder class.

Exposes name of Builder class

```
Employee employee2 = new Employee.EmployeeBuilder()  
    .firstName("Tony")  
    .lastName("Stark")  
    .title("Avenger")  
    .build();
```

Add a static builder() method

```
public class Employee {  
  
    public Employee(String firstName, String middleName, String lastName,  
                   String socialSecurity, String title,  
                   String department, String birthday, String salary) {  
        // ... initialization  
    }  
  
    public static EmployeeBuilder builder() {  
        return new EmployeeBuilder();  
    }  
  
}  
  
}  
}
```

```
Employee employee = Employee.builder()  
    .firstName("Tony")  
    .lastName("Stark")  
    .title("Avenger")  
    .build();
```

I'm writing a Java
program. Can I create
Builders with Lombok?

Builders with Lombok

```
package com.morrison;

import lombok.Builder;
import lombok.Getter;
import lombok.ToString;

@Builder
@Getter
@ToString
public class User {
    private final String firstName;
    private final String lastName;
    private final String email;
    private final int age;
    private final String address;

    Run | Debug
    public static final void main(String[] args) {

        User dj = User.builder()
            .firstName("Calvin")
            .lastName("Harris")
            .build();

        System.out.println(dj);
        System.out.println(String.format("Hello, %s %s", dj.getFirstName(), dj.getLastName()));

    }
}
```

Using Java 14?

It has records, basically an immutable class.

We can use Builder with records.

```
public record ConferenceTalk (String name, String presenter) {  
  
    public static void main(String[] args) {  
        var talk = new ConferenceTalk("Builder Design Patterns", "Kelly Morrison");  
        System.out.println(talk);  
        System.out.println("presenter = " + talk.presenter());  
    }  
}
```

Fields are private and final.
Getters are provided for you.
You also get: `toString()`, `equals()`, and `hashCode()`.

Using Java 14 records

```
public record ConferenceTalk(String name, String presenter) {  
    private ConferenceTalk(Builder builder) {  
        this(builder.name, builder.presenter);  
    }  
  
    public static Builder builder() {  
        return new Builder();  
    }  
  
    private static class Builder{  
        private String name = "";  
        private String presenter = "";  
  
        public Builder() {}  
  
        public Builder name(final String name) {  
            this.name = name;  
            return this;  
        }  
  
        public Builder presenter(final String presenter) {  
            this.presenter = presenter;  
            return this;  
        }  
  
        public ConferenceTalk build() {  
            return new ConferenceTalk(this);  
        }  
    }  
}  
  
ConferenceTalk talk = ConferenceTalk.builder()  
    .name("Builder Design Pattern")  
    .presenter("Kelly Morrison")  
    .build();
```

**I'm using IntelliJ IDEA and I heard it
can generate Builders for me.**

Yes, it can.

But does it do it well?

Let's look at an example.

```
c Employee.java ×  
1 package com.morrison.builders;  
2  
3 public class Employee {  
4     private final String firstName;  
5     private final String lastName;  
6  
7     Employee(String firstName, String lastName) {  
8         this.firstName = firstName;  
9         this.lastName = lastName;  
10    }  
11  
12    public String getFirstName() {  
13        return firstName;  
14    }  
15  
16    public String getLastName() {  
17        return lastName;  
18    }
```

```
Employee.java
```

```
1 package com.morrison.builders;
2
3 public class Employee {
4     private final String firstName;
5     private final String lastName;
6
7     Employee(String firstName, String lastName) {
8         this.firstName = firstName;
9         this.lastName = lastName;
10    }
11
12    public Employee() {
13        this("John", "Doe");
14    }
15
16    public Employee(String firstName) {
17        this(firstName, "Doe");
18    }
19
20    public String getFirstName() {
21        return firstName;
22    }
23
24    public String getLastName() {
25        return lastName;
26    }
27
28    public void setFirstName(String firstName) {
29        this.firstName = firstName;
30    }
31
32    public void setLastName(String lastName) {
33        this.lastName = lastName;
34    }
35
36    @Override
37    public String toString() {
38        return "Employee{" +
39            "firstName=" + firstName +
40            ", lastName=" + lastName +
41            '}';
42    }
43}
```

A context menu is open at the end of the constructor call in line 7. The menu title is "Show Context Actions". The visible items are:

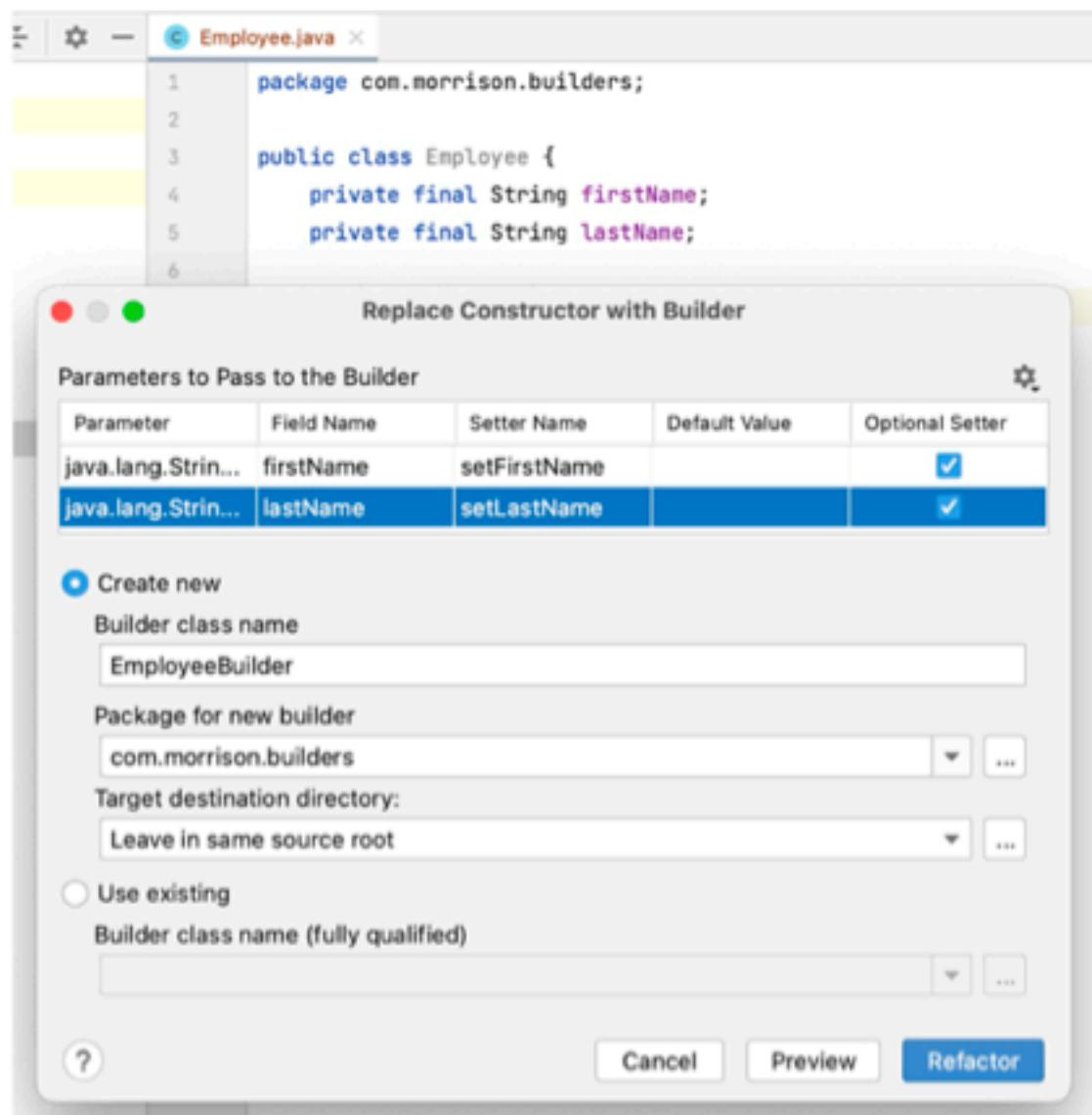
- Paste
- Copy / Paste Special
- Column Selection Mode
- Find Usages
- Go To
- Folding
- Analyze
- Refactor
- Generate...

```
Employee.java
```

```
1 package com.morrison.builders;
2
3 public class Employee {
4     private final String firstName;
5     private final String lastName;
6
7     Employee(String firstName, String lastName) {
8         this.firstName = firstName;
9         this.lastName = lastName;
10    }
11
12    public Employee() {
13        this("John", "Doe");
14    }
15
16    public Employee(String firstName) {
17        this(firstName, "Doe");
18    }
19
20    public String getFirstName() {
21        return firstName;
22    }
23
24    public String getLastName() {
25        return lastName;
26    }
27
28    public void setFirstName(String firstName) {
29        this.firstName = firstName;
30    }
31
32    public void setLastName(String lastName) {
33        this.lastName = lastName;
34    }
35
36    @Override
37    public String toString() {
38        return "Employee{" +
39            "firstName=" + firstName +
40            ", lastName=" + lastName +
41            '}';
42    }
43}
```

A context menu is open at the end of the constructor call in line 7. The menu title is "Show Context Actions". The visible items are:

- Safe delete 'Employee(String, String)' (highlighted)
- Change access modifier (highlighted)
- Insert '@Contract(pure = true)'
- Replace constructor with builder
- Replace constructor with factory method
- Add Javadoc
- Generate overloaded constructor with default parameters



```
public class EmployeeBuilder {  
    private String firstName;  
    private String lastName;  
  
    public EmployeeBuilder setFirstName(String firstName) {  
        this.firstName = firstName;  
        return this;  
    }  
  
    public EmployeeBuilder setLastName(String lastName) {  
        this.lastName = lastName;  
        return this;  
    }  
  
    public Employee createEmployee() { return new Employee(firstName, lastName); }  
}
```

**Generated in a separate file.
Non-standard name for build method.**

```
public class EmployeeBuilder {  
    private String firstName;  
    private String lastName;  
  
    public EmployeeBuilder setFirstName(String firstName) {  
        this.firstName = firstName;  
        return this;  
    }  
  
    public EmployeeBuilder setLastName(String lastName) {  
        this.lastName = lastName;  
        return this;  
    }  
  
    public Employee createEmployee() { return new Employee(firstName, lastName); }  
}
```

**Let's see how well ChatGPT does
when creating Builders...**



ChatGPT

The official app by OpenAI

In-A

ChatGPT 4o mini ▾

Log in

Reckon you could fix me up with one of them Builder patterns for an immutable Java class called Employee that has a first name, last name, and age? It should not let me create an employee with an age less than 16 or older than 130.



Sure thing! The Builder pattern is great for creating immutable objects, especially when you have validation logic like this. Here's how you can design an immutable 'Employee' class with a Builder pattern in Java:

1. Define the 'Employee' class with private fields and a private constructor.
2. Create a static nested 'Builder' class within the 'Employee' class.
3. Ensure that the 'Employee' class is immutable and validate the age within the 'Builder' class.

```
public final class Employee {  
    private final String firstName;  
    private final String lastName;  
    private final int age;  
  
    // Private constructor to be used by the Builder  
    private Employee(Builder builder) {  
        this.firstName = builder.firstName;  
        this.lastName = builder.lastName;  
        this.age = builder.age;  
    }  
  
    // Getters  
    public String getFirstName() {  
        return firstName;  
    }  
}
```



```
// Static nested Builder class  
public static class Builder {  
    private String firstName;  
    private String lastName;  
    private int age;  
  
    // Method to set the first name  
    public Builder setFirstName(String firstName) {  
        this.firstName = firstName;  
        return this;  
    }  
  
    // Method to set the last name  
    public Builder setLastName(String lastName) {  
        this.lastName = lastName;  
        return this;  
    }  
  
    // Method to set the age with validation  
    public Builder setAge(int age) {  
        if (age < 16 || age > 130) {  
            throw new IllegalArgumentException("Age must be between 16 and 130");  
        }  
        this.age = age;  
        return this;  
    }  
  
    // Method to build the Employee instance  
    public Employee build() {  
        return new Employee(this);  
    }  
}
```

How do I create a Builder In Go?

```
package main

import "fmt"

// User struct defines the fields for the user.
type User struct {
    FirstName string
    LastName  string
    Email     string
    Age       int
    Address   string
}

// UserBuilder struct provides a builder for constructing a User.
type UserBuilder struct {
    firstName string
    lastName  string
    email     string
    age       int
    address   string
}

// NewUserBuilder initializes and returns a new UserBuilder.
func NewUserBuilder() *UserBuilder {
    return &UserBuilder{}
}

// SetFirstName sets the first name of the user in the builder.
func (b *UserBuilder) SetFirstName(firstName string) *UserBuilder {
    b.firstName = firstName
    return b
}

// SetLastName sets the last name of the user in the builder.
func (b *UserBuilder) SetLastName(lastName string) *UserBuilder {
    b.lastName = lastName
    return b
}

// SetEmail sets the email of the user in the builder.
func (b *UserBuilder) SetEmail(email string) *UserBuilder {
    b.email = email
    return b
}

// SetAge sets the age of the user in the builder.
func (b *UserBuilder) SetAge(age int) *UserBuilder {
    b.age = age
    return b
}

// SetAddress sets the address of the user in the builder.
func (b *UserBuilder) SetAddress(address string) *UserBuilder {
    b.address = address
    return b
}

// Build constructs and returns a User instance.
func (b *UserBuilder) Build() *User {
    return &User{
        FirstName: b.firstName,
        LastName:  b.lastName,
        Email:     b.email,
        Age:       b.age,
        Address:   b.address,
    }
}
```

```
func main() {
    // Example usage of the UserBuilder to construct a User.
    builder := NewUserBuilder()
    user := builder.SetFirstName("John").
        SetLastName("Doe").
        SetEmail("john.doe@example.com").
        SetAge(30).
        SetAddress("123 Main St").
        Build()

    fmt.Printf("User: %+v\n", user)
}
```

What about TypeScript?

Can I implement Builders in TypeScript?

```
interface ComputerBuilder {
    setCPU(cpu: string): this;
    setRAM(ram: string): this;
    setStorage(storage: string): this;
    setGraphicsCard(graphicsCard: string): this;
    build(): Computer;
}

class Computer {
    private readonly cpu: string;
    private readonly ram: string;
    private readonly storage: string;
    private readonly graphicsCard: string;

    constructor(cpu: string, ram: string, storage: string, graphicsCard: string) {
        this.cpu = cpu;
        this.ram = ram;
        this.storage = storage;
        this.graphicsCard = graphicsCard;
    }

    public toString(): string {
        return `Computer [CPU=${this.cpu}, RAM=${this.ram},
        Storage=${this.storage}, GraphicsCard=${this.graphicsCard}]`;
    }

    static builder(): ComputerBuilder {
        return new ConcreteComputerBuilder();
    }
}
```

```
class ConcreteComputerBuilder implements ComputerBuilder {
    private cpu: string = 'Default CPU';
    private ram: string = 'Default RAM';
    private storage: string = 'Default Storage';
    private graphicsCard: string = 'Integrated';

    public setCPU(cpu: string): this {
        this.cpu = cpu;
        return this;
    }

    public setRAM(ram: string): this {
        this.ram = ram;
        return this;
    }

    public setStorage(storage: string): this {
        this.storage = storage;
        return this;
    }

    public setGraphicsCard(graphicsCard: string): this {
        this.graphicsCard = graphicsCard;
        return this;
    }

    public build(): Computer {
        return new Computer(this.cpu, this.ram, this.storage, th:
```

```
const computer = Computer.builder()
    .setCPU('Intel i9')
    .setRAM('32GB')
    .setStorage('1TB SSD')
    .setGraphicsCard('NVIDIA RTX 3080')
    .build();

console.log(computer.toString());
```

Summary / Takeaways

Builder lets you construct immutable objects for concurrent / functional programming.

Multiple Builders can simplify construction by guiding or limiting options.

Builders promote “fail fast” object creation and prevent invalid / inconsistent state.

Builders work well with Java 14 records.

Builders can gate the creation of objects (limited number, etc.)

Knowledge of Builders can help you prompt AI to generate better code.

[Slides on GitHub](#)



[LinkedIn](#)



CGI

kellyivymorrison@gmail.com

[Slides on GitHub from CodeMash 2023 on the Visitor design pattern](#)

