CS 351
Design of Large Programs
The Builder Pattern

Brooke Chenoweth

University of New Mexico

Spring 2025

Example: Car Class

```
public class Car {
  private int doors;
  private int wheels;
  private int seats;

  public Car(int doors, int wheels, int seats) {
    this.doors = doors;
    this.wheels = wheels;
    this.seats = seats;
  }

  // getters, other methods, etc.
}
```

Let's make some cars!

- `Car c1 = new Car(2, 4, 4);`
- `Car c2 = new Car(4, 4, 5);`
- `Car c3 = new Car(4, 4, 7);`

Let's make some cars!

- `Car c1 = new Car(2, 4, 4);`
  A 2-door car with four wheels and four seats.
- `Car c2 = new Car(4, 4, 5);`
- `Car c3 = new Car(4, 4, 7);`

Let's make some cars!

- `Car c1 = new Car(2, 4, 4);`

  A 2-door car with four wheels and four seats.

- `Car c2 = new Car(4, 4, 5);`

  A 4-door car with four wheels and five seats.

- `Car c3 = new Car(4, 4, 7);`

Let's make some cars!

- `Car c1 = new Car(2, 4, 4);`

  A 2-door car with four wheels and four seats.

- `Car c2 = new Car(4, 4, 5);`

  A 4-door car with four wheels and five seats.

- `Car c3 = new Car(4, 4, 7);`

  A 4-door car with four wheels and seven seats.

Let's make some cars!

- `Car c1 = new Car(2, 4, 4);`

  A 2-door car with four wheels and four seats.

- `Car c2 = new Car(4, 4, 5);`

  A 4-door car with four wheels and five seats.

- `Car c3 = new Car(4, 4, 7);`

  A 4-door car with four wheels and seven seats.
  Most cars have 4 wheels, maybe make another constructor with a default value?

Let's make some cars!

- `Car c1 = new Car(2, 4, 4);`
  A 2-door car with four wheels and four seats.
- `Car c2 = new Car(4, 4, 5);`
  A 4-door car with four wheels and five seats.
- `Car c3 = new Car(4, 4, 7);`
  A 4-door car with four wheels and seven seats.
  Most cars have 4 wheels, maybe make another constructor with a default value?
- `Car c4 = new Car(4, 7, 4);`

Let's make some cars!

- `Car c1 = new Car(2, 4, 4);`
  A 2-door car with four wheels and four seats.
- `Car c2 = new Car(4, 4, 5);`
  A 4-door car with four wheels and five seats.
- `Car c3 = new Car(4, 4, 7);`
  A 4-door car with four wheels and seven seats.
  Most cars have 4 wheels, maybe make another constructor with a default value?
- `Car c4 = new Car(4, 7, 4);`
  A 4-door car with seven wheels and four seats?
  Arguments of the same type are easy to confuse.

Problems with Constructors

- Too many arguments – easily confused
- Optional arguments – overload constructors with defaults?
- Too many constructors

Use setters instead?

```
Car car = new Car();
car.setWheels(4);
car.setSeats(2);
```

Use setters instead?

```
Car car = new Car();
car.setWheels(4);
car.setSeats(2);
```

- We forgot to set the doors!
- Do we have reasonable default values?
- What if we didn't finish configuring the object?
- Does it even make sense to change the number of wheels after the Car is constructed?

The Builder Pattern

Use the *Builder Pattern* to encapulate the construction of a product and allow it to be constructed in steps.

CarBuilder

```
public class CarBuilder {
  private int doors;
  private int wheels = 4;
  private int seats;

  public void setDoors(int doors) {
    this.doors = doors;
  }

  public void setWheels(int wheels) {
    this.wheels = wheels;
  }

  public void setSeats(int seats) {
    this.seats = seats;
  }

  public Car getCar() {
    return new Car(doors, wheels, seats);
  }
}
```

Let's make a Car!

```
CarBuilder cb = new CarBuilder();
cb.setDoors(2);
cb.setSeats(4);
Car car = cb.getCar();
```

- We can build up a complex object with a step by step approach.
- We could add error checking before actually constructing the Car object to make sure we've properly configured all the fields.
- We could use a builder to create an immutable object.

CarBuilder with Fluent Interface

```
public class CarBuilder {
  private int doors;
  private int wheels = 4;
  private int seats;

  public CarBuilder setDoors(int doors) {
    this.doors = doors;
    return this;
  }

  public CarBuilder setWheels(int wheels) {
    this.wheels = wheels;
    return this;
  }

  public CarBuilder setSeats(int seats) {
    this.seats = seats;
    return this;
  }

  public Car getCar() {
    return new Car(doors, wheels, seats);
  }
}
```

Fluent Interface

The Builder pattern is often implemented with a *fluent interface*, where each method in the builder returns a reference to the builder object itself so we can easily chain the methods together.

```
Car car = new CarBuilder().setDoors(2)
                          .setSeats(4)
                          .getCar();
```

This coding idiom of returning `this` and *method chaining* is independent of the Builder pattern, but crops up often enough that it's worth mentioning here.