Java best practice

Nguyen Trung Hieu

Special thanks

This presentation was made from the ideas of Joshua Bloch’s Efficicetive Java 3rd Edition, 2018

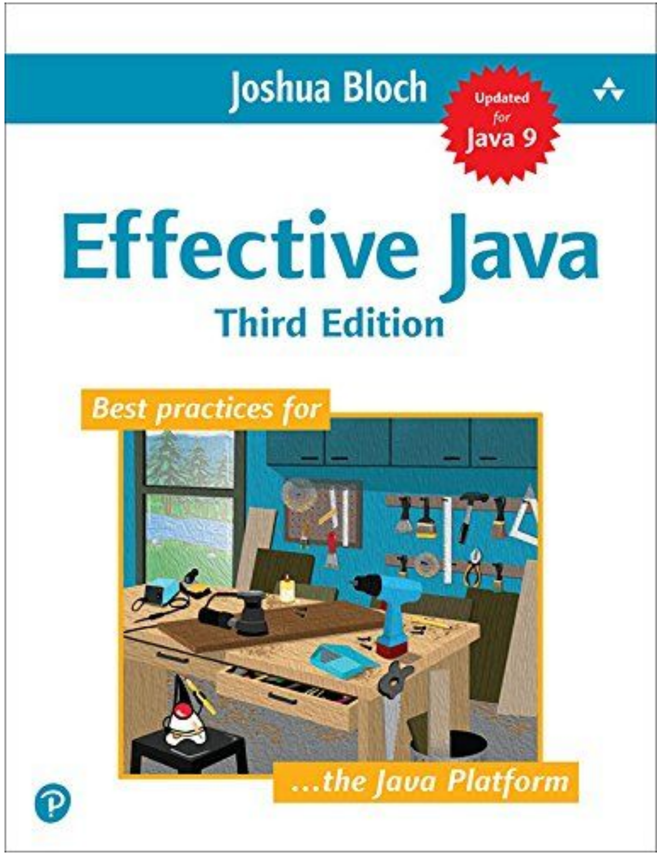


Table of contents

Consider static factory methods over constructors

The problem with constructors

```
public class RandomIntGenerator {
    private final int min;
    private final int max;

    public int next() {...}
}
```

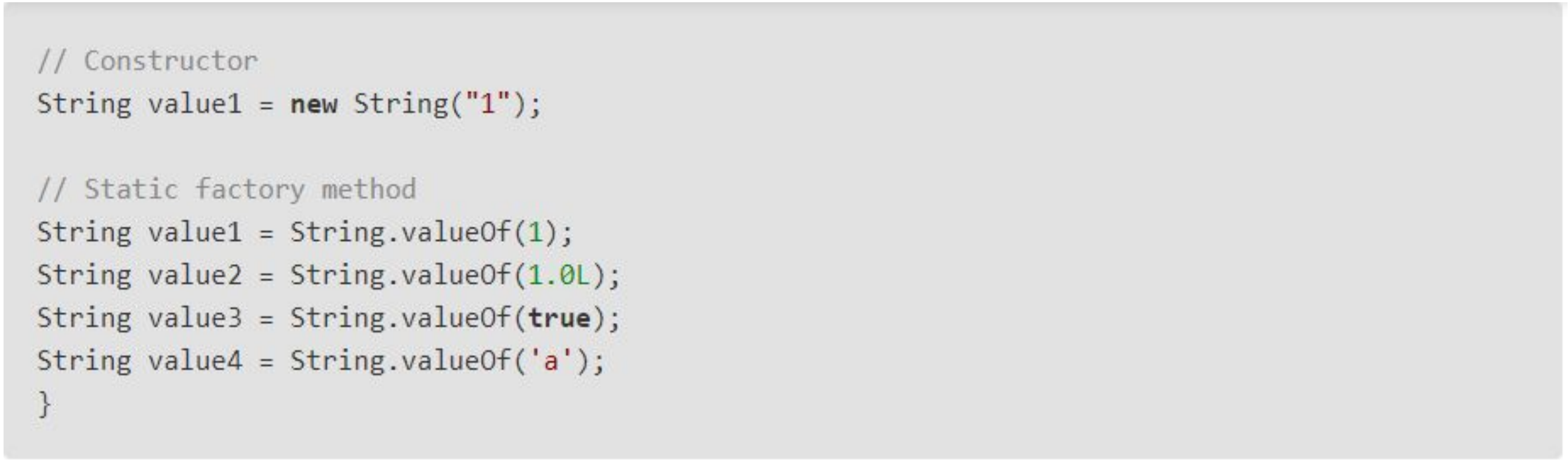
The problem with constructors

```java
public class RandomIntGenerator {
    private final int min;
    private final int max;

    public int next() {...}

    public RandomIntGenerator(int min, int max) {
    this.min = min;
    this.max = max;
    }
    public RandomIntGenerator(int min) {
    this.min = min;
    this.max = Integer.MAX_VALUE;
    }
    //Compilation error
    public RandomIntGenerator(int max) {
    this.min = Integer.MIN_VALUE;
    this.max = max;
    }

}
```

Solution - static factory method

```
public class RandomIntGenerator {
    private final int min;
    private final int max;

    private RandomIntGenerator(int min, int max) {
        this.min = min;
        this.max = max;
    }

    public static RandomIntGenerator between(int max, int min) {
        return new RandomIntGenerator(min, max);
    }

    public static RandomIntGenerator biggerThan(int min) {
        return new RandomIntGenerator(min, Integer.MAX_VALUE);
    }

    public static RandomIntGenerator smallerThan(int max) {
        return new RandomIntGenerator(Integer.MIN_VALUE, max);
    }

    public int next() {...}
}
```

1- Constructors vs factory method

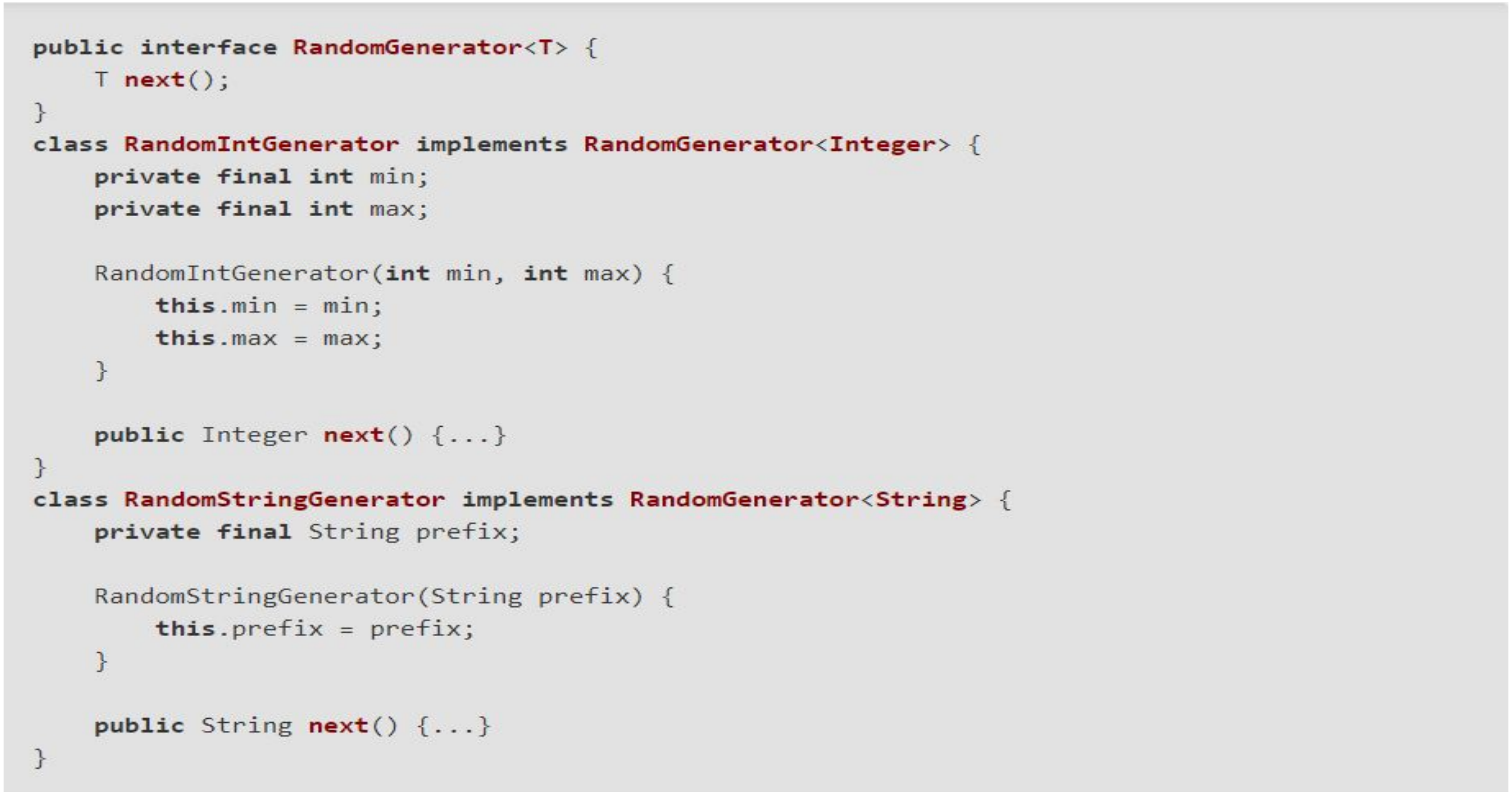


```
// Constructor
String value1 = new String("1");

// Static factory method
String value1 = String.valueOf(1);
String value2 = String.valueOf(1.0L);
String value3 = String.valueOf(true);
String value4 = String.valueOf('a');
}
```

1- Why static factory methods?

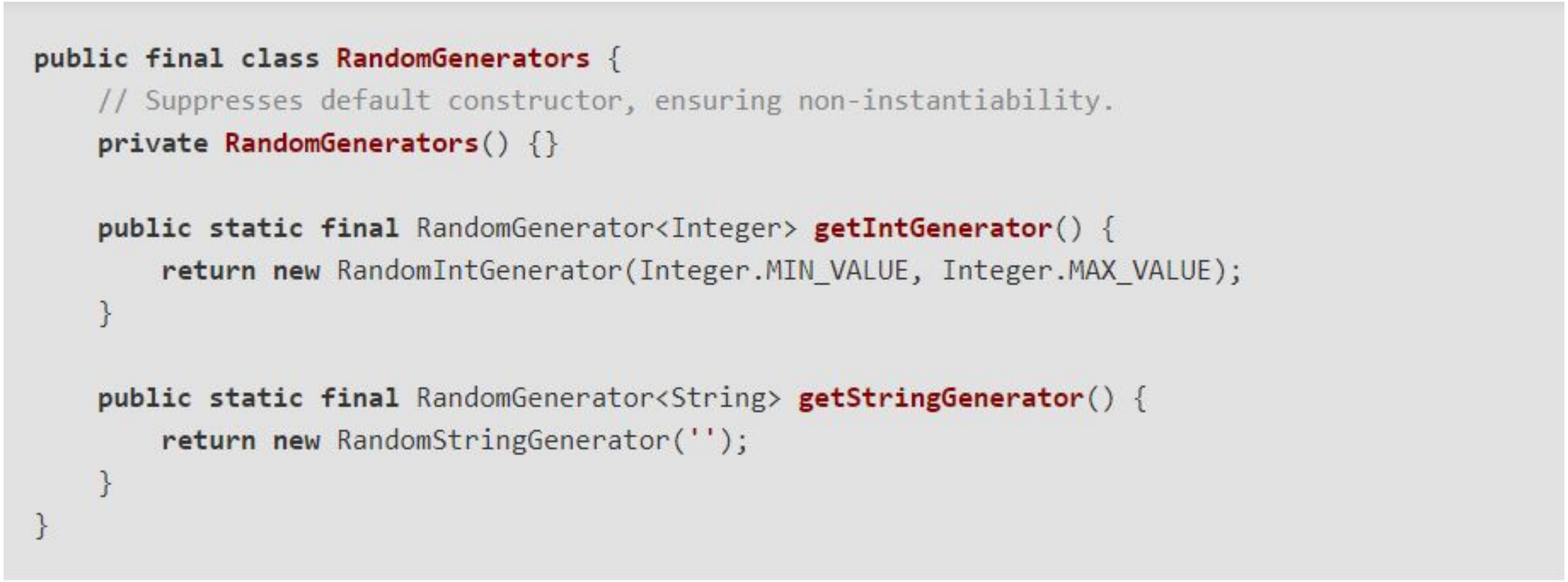
- Methods have name, Constructors not.
- Returns same object from repeated invocations
- Can return an object of any subtype of their return type
- Move logic out of constructor

1- Why static factory methods?



```java
public interface RandomGenerator<T> {
    T next();
}
class RandomIntGenerator implements RandomGenerator<Integer> {
    private final int min;
    private final int max;

    RandomIntGenerator(int min, int max) {
        this.min = min;
        this.max = max;
    }

    public Integer next() {...}
}
class RandomStringGenerator implements RandomGenerator<String> {
    private final String prefix;

    RandomStringGenerator(String prefix) {
        this.prefix = prefix;
    }

    public String next() {...}
}
```

1- Why static factory methods?



```java
public final class RandomGenerators {
    // Suppresses default constructor, ensuring non-instantiability.
    private RandomGenerators() {}

    public static final RandomGenerator<Integer> getIntGenerator() {
        return new RandomIntGenerator(Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    public static final RandomGenerator<String> getStringGenerator() {
        return new RandomStringGenerator('');
    }
}
```

1- Why static factory methods?

```java
public class User {

    private static final Logger LOGGER = Logger.getLogger(User.class.getName());
    private final String name;
    private final String email;
    private final String country;

    // standard constructors / getters

    public static User createWithLoggedInstantiationTime(
      String name, String email, String country) {
        LOGGER.log(Level.INFO, "Creating User instance at : {0}", LocalTime.now());
        return new User(name, email, country);
    }
}
```

1- Limitations

- **Classes without public or protected constructors cannot be subclassed**
- **Difficult to find in document**

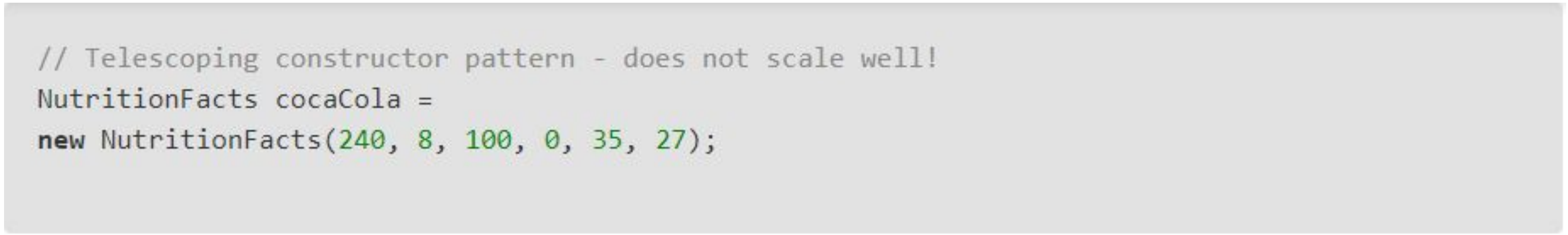
Consider a builder when faced with many constructor parameters

The problem

- Both static factory method and constructor does not scale well to large number of parameters

=> Traditional approach: Telescoping constructor pattern

Telescoping constructor

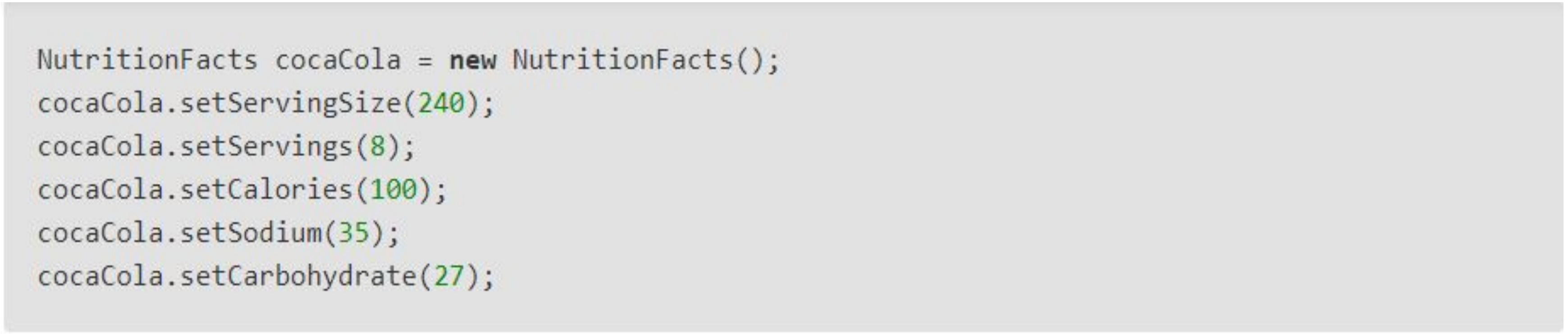


```
// Telescoping constructor pattern - does not scale well!
NutritionFacts cocaCola =
new NutritionFacts(240, 8, 100, 0, 35, 27);
```

- **When number of paramters increase, hard to write client code**
- **Hard to read**

=> Another solution: JavaBean pattern

Java Bean pattern

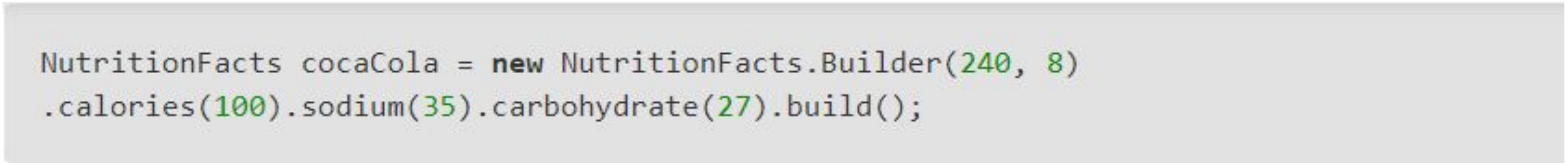


```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Limitation: a JavaBean may be in an inconsistent state partway through its construction => Class must be mutable

=> Final solution: Builder pattern

Builder pattern



```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
.calories(100).sodium(35).carbohydrate(27).build();
```

- Easy to read
- Work with immutable classes
- Scale friendly