Fluent Designのすすめ

Java Genericsを用いて入れ子構造と継承に溢れた楽しい

Fluent APIを作りましょう

@haochenxie

2017/6/24
ジェネリクス勉強会

Haochen Xie
名古屋大学

https://haochenxie.name

Java vs. Generics

実はですね

- Java Type System is **Turing Complete**
- Java Type System is **Unsafe !!**

Java Generics is **Turing Complete**

Java Generics Are Turing Complete

Radu Grigore
University of Kent, United Kingdom

Radu Grigore. 2017. Java generics are turing complete. SIGPLAN Not. 52, 1 (January 2017), 73-85. DOI: https://doi.org/10.1145/3093333.3009871

Java Generics is **Unsafe !!**

Java and Scala's Type Systems are Unsound *

The Existential Crisis of Null Pointers

Nada Amin
EPFL, Switzerland
nada.amin@epfl.ch

Ross Tate
Cornell University, USA
ross@cs.cornell.edu

Nada Amin and Ross Tate. 2016. Java and scala's type systems are unsound: the existential crisis of null pointers. *SIGPLAN Not.* 51, 10 (October 2016), 838-848. DOI: https://doi.org/10.1145/3022671.2984004

今日の目標

- Fluent Designが素晴しい
- Java Genericsが素晴しい
- **JAVA**が素晴しい
- (Fluent Designのテクニック)

Fluent Style

```
CanvasObj obj = Graphics.builder()
        .newGroup()
            .newCircle().x(10).y(10).radius(20).circle()
            .newText().text("Hello").text()
            .end()
        .newGroup().rotation(90)
            .newCircle().x(10).y(20).radius(20).circle()
            .newText()
                .text("world")
                .font().style(Font.FontStyle.BOLD).font()
                .text()
            .end()
        .build();
```

Fluentの素晴しさ

- Consider a Java library. We say that it has a fluent interface when it encourages its users to chain method calls: `f().g().h()`.
- Programs written in the fluent style are more *readable* than programs written in the traditional, sequential style: `f(); g(); h()`.
- Why? In the fluent style, the order in which methods are called is *constrained by the type checker*.
- If the library is well designed, its types will constrain the chains of methods calls such that they are *easy to read and understand*.
- In the traditional style, we *cannot* use the type checker to enforce readability

Radu Grigore. 2017. Java generics are turing complete. SIGPLAN Not. 52, 1 (January 2017), 73-85.
DOI: https://doi.org/10.1145/3093333.3009871

Fluent Style

+α

継承のやりかた

```
CanvasObj obj = Graphics.builder()
        .newGroup()
            .newCircle().x(10).y(10).radius(20).circle()
            .newText().text("Hello").text()
            .end()
        .newGroup().rotation(90)
            .newCircle().x(10).y(20).radius(20).circle()
            .newText()
                .text("world")
                .font().style(Font.FontStyle.BOLD).font()
                .text()
            .end()
        .build();
```

3 harvester pattern

1 action pattern

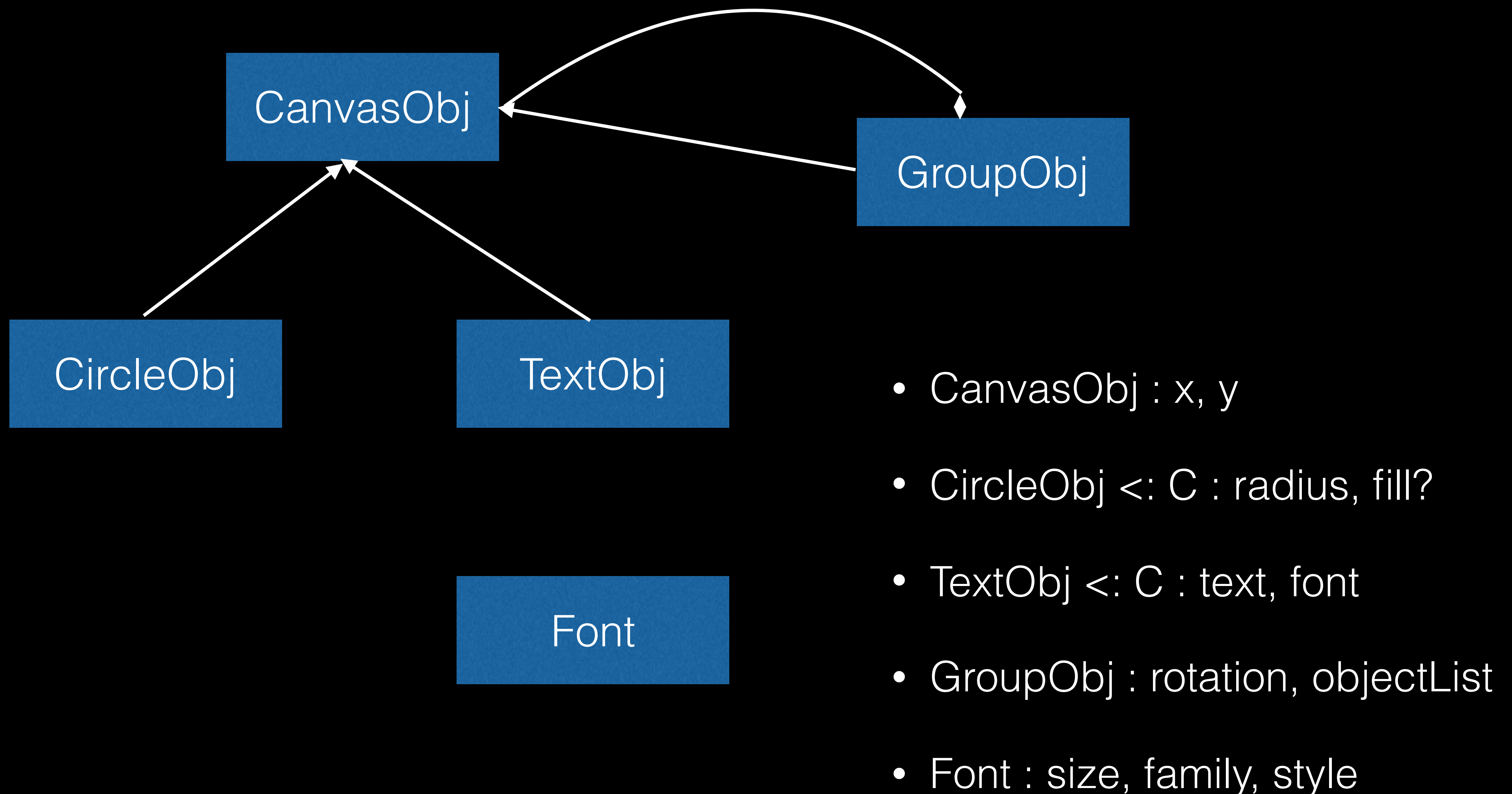
nesting pattern

4

2 finisher pattern

Technique Session

A Tiny Graphics Library



Level 1: Basics
Action & Finisher Pattern

```
static class CircleBuilder0 {
    private CircleObj state = new CircleObj();
    CircleBuilder0 x(double x) {
        state.setX(x);
        return this;
    }
    …
    CircleObj build() {
        return state;
    }
}
```

1 action pattern

2 finisher pattern

Level 3: Harvester Pattern

```
static class FontBuilder<ContT> {
    interface FontHarvester<ContT> {
        ContT harvest(Font font);
    }
    Font state = new Font();
    FontHarvester<ContT> harvester;
    FontBuilder(…) { … }
    Font build() { return state; }
    ContT font() {
        return harvester.harvest(build());
    }
}
```

「Continuation Type」
の略である

レベル表 (因みに)

- Level 0 = Hello world
- Level 1 = Beginner Programmer
- Level 2 = Usual Programmer
- Level 3 = Expert
- Level 4 = Expert + ?? 例えば**スペシャルスキル**持ち?

Level 3: Nesting Pattern

```
static class GroupBuilder<ContT> {
    interface GroupHarvester<ContT> { … }
    Group state = new Group();
    GroupHarvester<ContT> harvester;
    GroupBuilder(GroupHarvester<ContT> harvester) { … }
    Group build() { return state; }
    GroupBuilder<ContT> add(CanvasObj obj) { … }
    ContT end() { return harvester.harvest(build()); }
    GroupBuilder<GroupBuilder<ContT>> newGroup() {
        return new GroupBuilder<>(obj -> {
            return add(obj);
        });
    }
}
```

対となる

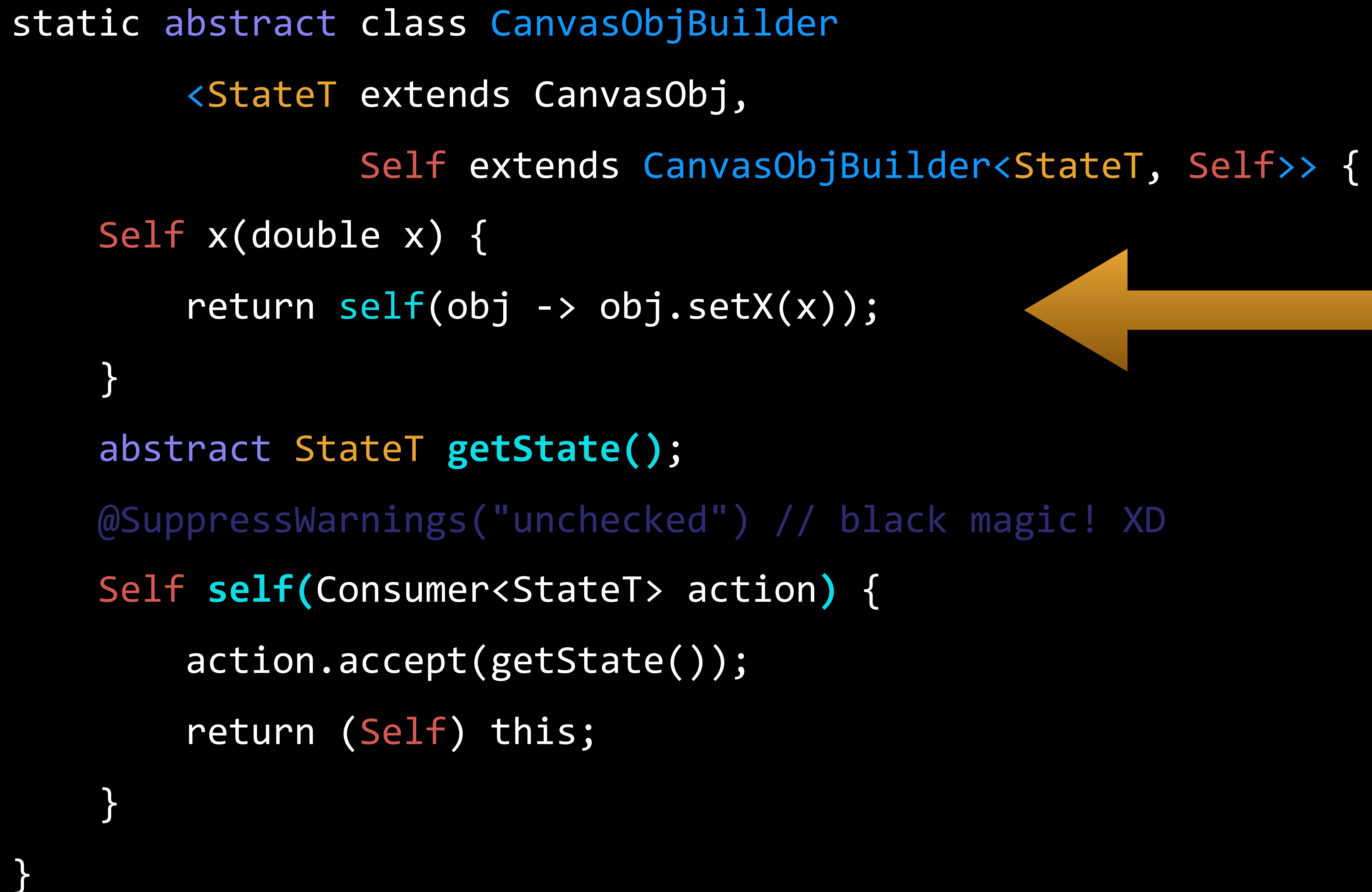
Harvester + Action Pattern
他ならない

Level 4:
継承で遊びましょう!! (1)

lombokです

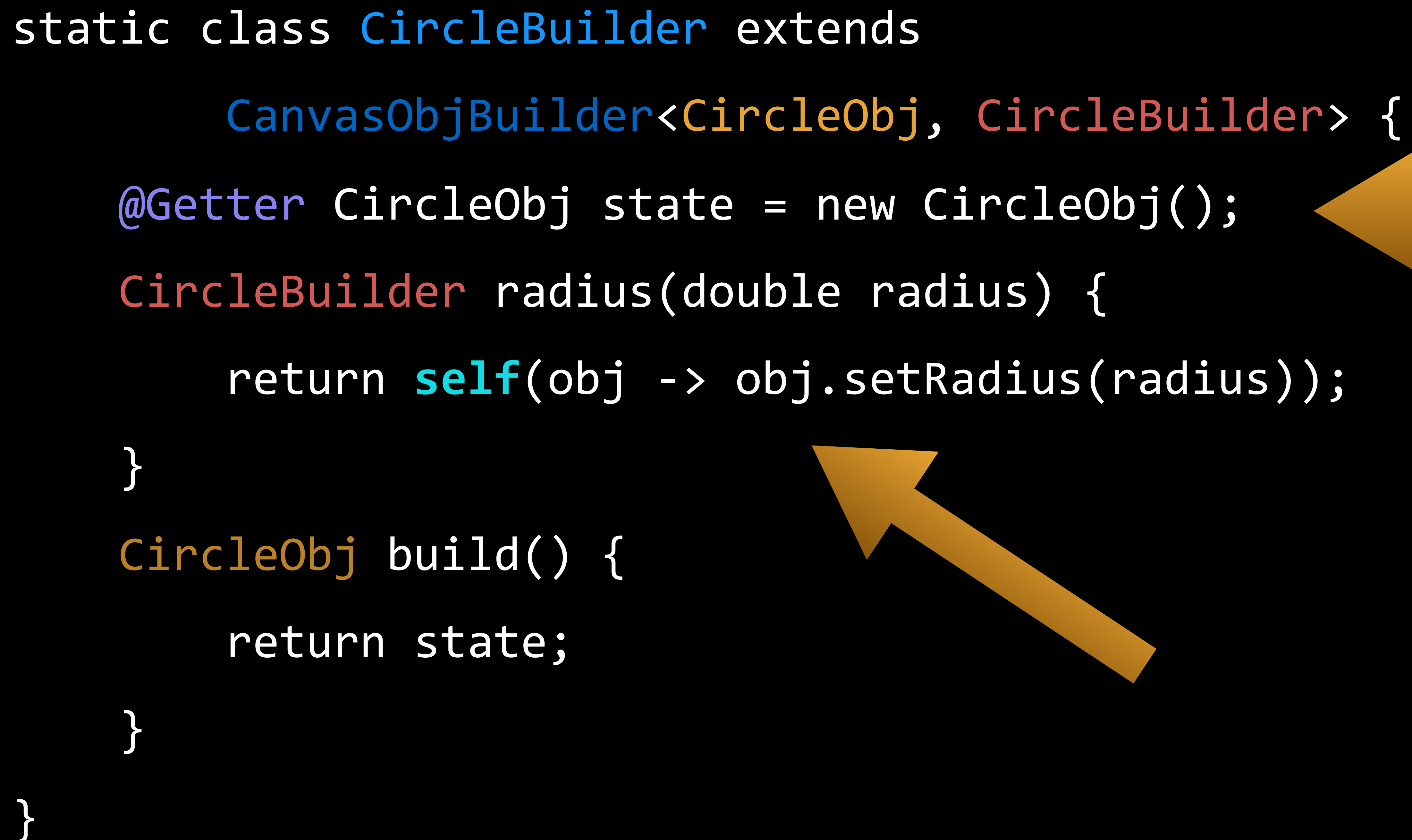
```
@Data
static abstract class CanvasObj {
    double x, y;
}
@Data
static class CircleObj extends CanvasObj {
    double radius;
    boolean fill;
}
```

Level 4:
継承で遊びましょう!! (2)



```
static abstract class CanvasObjBuilder
        <StateT extends CanvasObj,
                Self extends CanvasObjBuilder<StateT, Self>> {
    Self x(double x) {
        return self(obj -> obj.setX(x));
    }
    abstract StateT getState();
    @SuppressWarnings("unchecked") // black magic! XD
    Self self(Consumer<StateT> action) {
        action.accept(getState());
        return (Self) this;
    }
}
```

Level 4:
継承で遊びましょう!! (3)



Live Coding Session (?)

Wrap-up

- **JAVA**が素晴しい
- Java Genericsが素晴しい
- Fluent Designが素晴しい
- Fluent Designのテクニック
    1. Action & Finisher Pattern
    2. Harvest & Nesting Pattern
    3. 継承!!

Java Type System is **Unsafe !!** (1)

(おまけです)

```
static <T,U> U coerce(T t) {
    …
}


void experiment() {
    String zero = coerce(0);
    // type checks but at runtime throws
    // ClassCastException Integer -> String !!
}
```

Java Type System is **Unsafe !!** (2)

(おまけです)

```
interface Constraint<A,B extends A> {}
static class Bind<A,B extends A> {
    Bind(Constraint<A, B> constraint) { }
    A upcast(B b) { return b; }
}
static <T,U> U coerce(T t) {
    Constraint<U,? super T> constraint = null;
    return new Bind<>(constraint).upcast(t);
}
```