

# 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

# 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を作しましょう



@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

実はですね

- Java Type System is **Turing Complete**

# 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

# 今日の目標

- Fluent Designが素晴らしい

# 今日の目標

- Fluent Designが素晴らしい
- Java Genericsが素晴らしい

# 今日の目標

- Fluent Designが素晴らしい
- Java Genericsが素晴らしい
- **JAVA**が素晴らしい

# 今日の目標

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

# Fluent Style

# Fluent Style

```
CanvasObj obj = Graphics.builder()
```

# Fluent Style

```
CanvasObj obj = Graphics.builder()  
    .newGroup()
```



# Fluent Style

```
CanvasObj obj = Graphics.builder()  
    .newGroup()  
    .newCircle().x(10).y(10).radius(20).circle()
```

# Fluent Style

```
CanvasObj obj = Graphics.builder()  
    .newGroup()  
        .newCircle().x(10).y(10).radius(20).circle()  
        .newText().text("Hello").text()
```

# Fluent Style

```
CanvasObj obj = Graphics.builder()  
    .newGroup()  
        .newCircle().x(10).y(10).radius(20).circle()  
        .newText().text("Hello").text()  
    .end()
```

# Fluent Style

```
CanvasObj obj = Graphics.builder()  
    .newGroup()  
        .newCircle().x(10).y(10).radius(20).circle()  
        .newText().text("Hello").text()  
        .end()  
    .newGroup().rotation(90)
```

# 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()
```

# 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()
```

# 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")
```

# 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()
```



# 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()
```

# 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()
```

# 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

# 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()`.

# Fluentの素晴らしさ

- 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()`.

# Fluentの素晴らしさ

- 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*.

# 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*.



# Fluentの素晴らしさ

- 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

# 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 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();
```

**1** action pattern

# Fluent Style

```
CanvasObj obj = Graphics.builder()
```

```
    .newGroup()
```

```
        .newCircle().x(10).y(10).radius(20).circle()
```

```
        .newText().text("Hello").text()
```

```
    .end()
```

```
    .newGroup().rotation(90)
```

**1**

action pattern

```
        .newCircle().x(10).y(20).radius(20).circle()
```

```
        .newText()
```

```
            .text("world")
```

```
            .font().style(Font.FontStyle.BOLD).font()
```

```
            .text()
```

```
    .end()
```

```
    .build();
```

**2**

finisher pattern

# Fluent Style

```
CanvasObj obj = Graphics.builder()
```

```
    .newGroup()
```

3

```
    .newCircle().x(10).y(10).radius(20).circle()
```

```
    .newText().text("Hello").text()
```

```
    .end()
```

```
    .newGroup().rotation(90)
```

1

action pattern

```
    .newCircle().x(10).y(20).radius(20).circle()
```

```
    .newText()
```

```
        .text("world")
```

```
        .font().style(Font.FontStyle.BOLD).font()
```

```
        .text()
```

```
    .end()
```

```
    .build();
```

2

finisher pattern

harvester pattern

# Fluent Style

```
CanvasObj obj = Graphics.builder()
```

```
    .newGroup()
```

3

```
    .newCircle().x(10).y(10).radius(20).circle()
```

harvester pattern

```
    .newText().text("Hello").text()
```

```
    .end()
```

```
    .newGroup().rotation(90)
```

1

action pattern

```
    .newCircle().x(10).y(20).radius(20).circle()
```

```
    .newText()
```

```
        .text("world")
```

```
        .font().style(Font.FontStyle.BOLD).font()
```

```
        .text()
```

nesting pattern

4

```
    .end()
```

```
    .build();
```

2

finisher pattern

# Fluent Style

+a

継承のやりかた

```
CanvasObj obj = Graphics.builder()
```

```
.newGroup()
```

3

```
.newCircle().x(10).y(10).radius(20).circle()
```

```
.newText().text("Hello").text()
```

```
.end()
```

```
.newGroup().rotation(90)
```

1

action pattern

```
.newCircle().x(10).y(20).radius(20).circle()
```

```
.newText()
```

```
.text("world")
```

```
.font().style(Font.FontStyle.BOLD).font()
```

```
.text()
```

4

```
.end()
```

```
.build();
```

2

finisher pattern

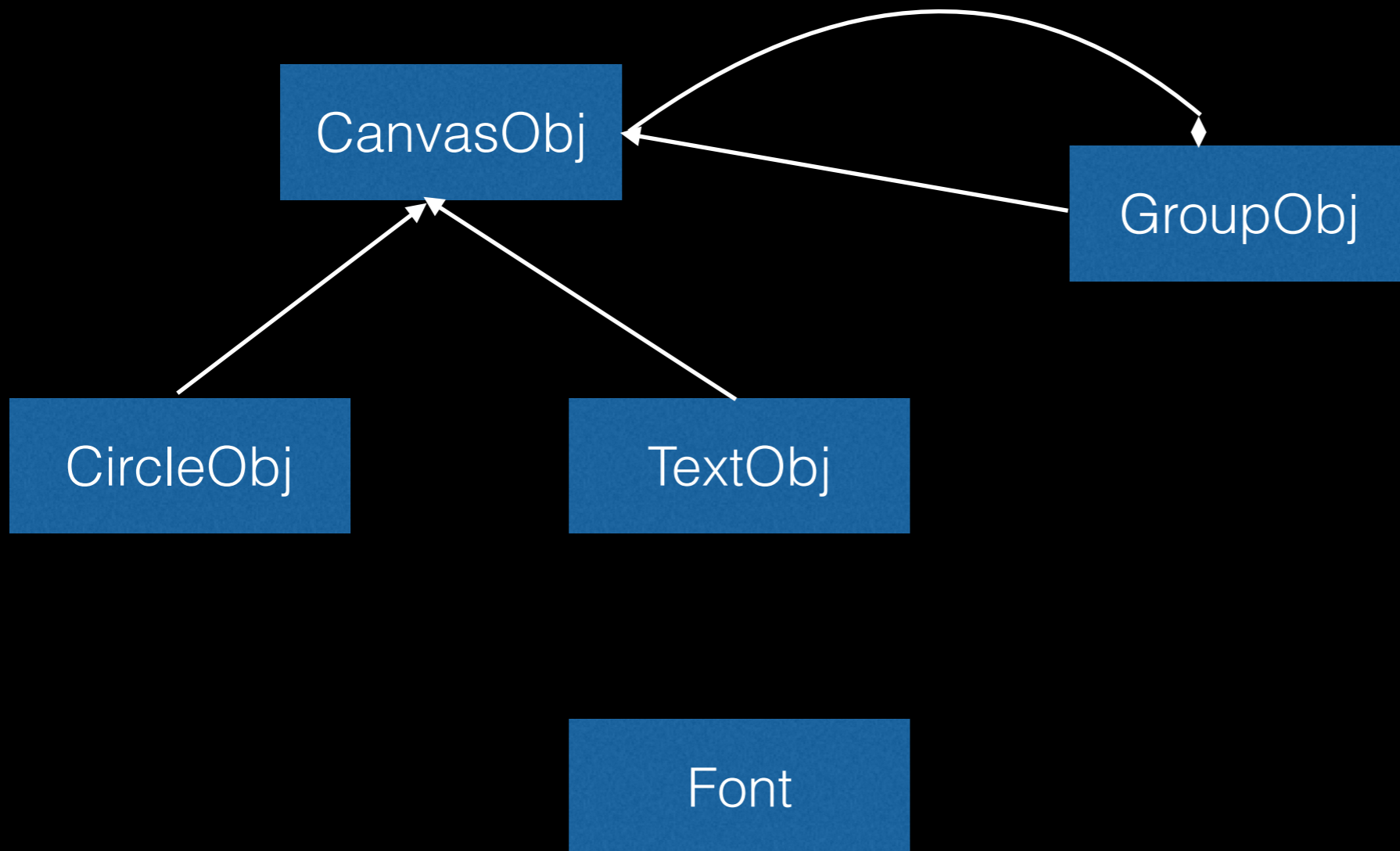
harvester pattern

nesting pattern

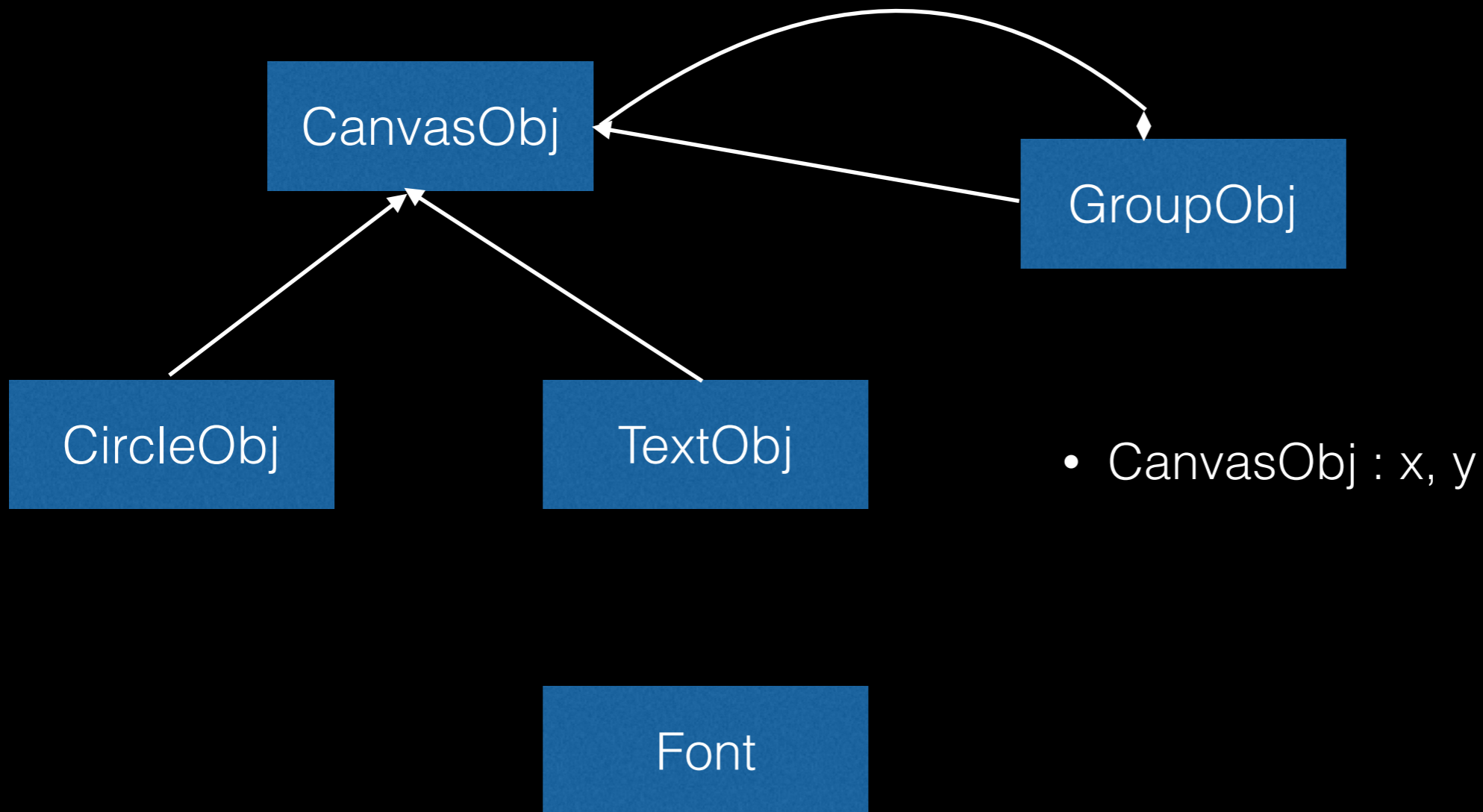
# Technique Session



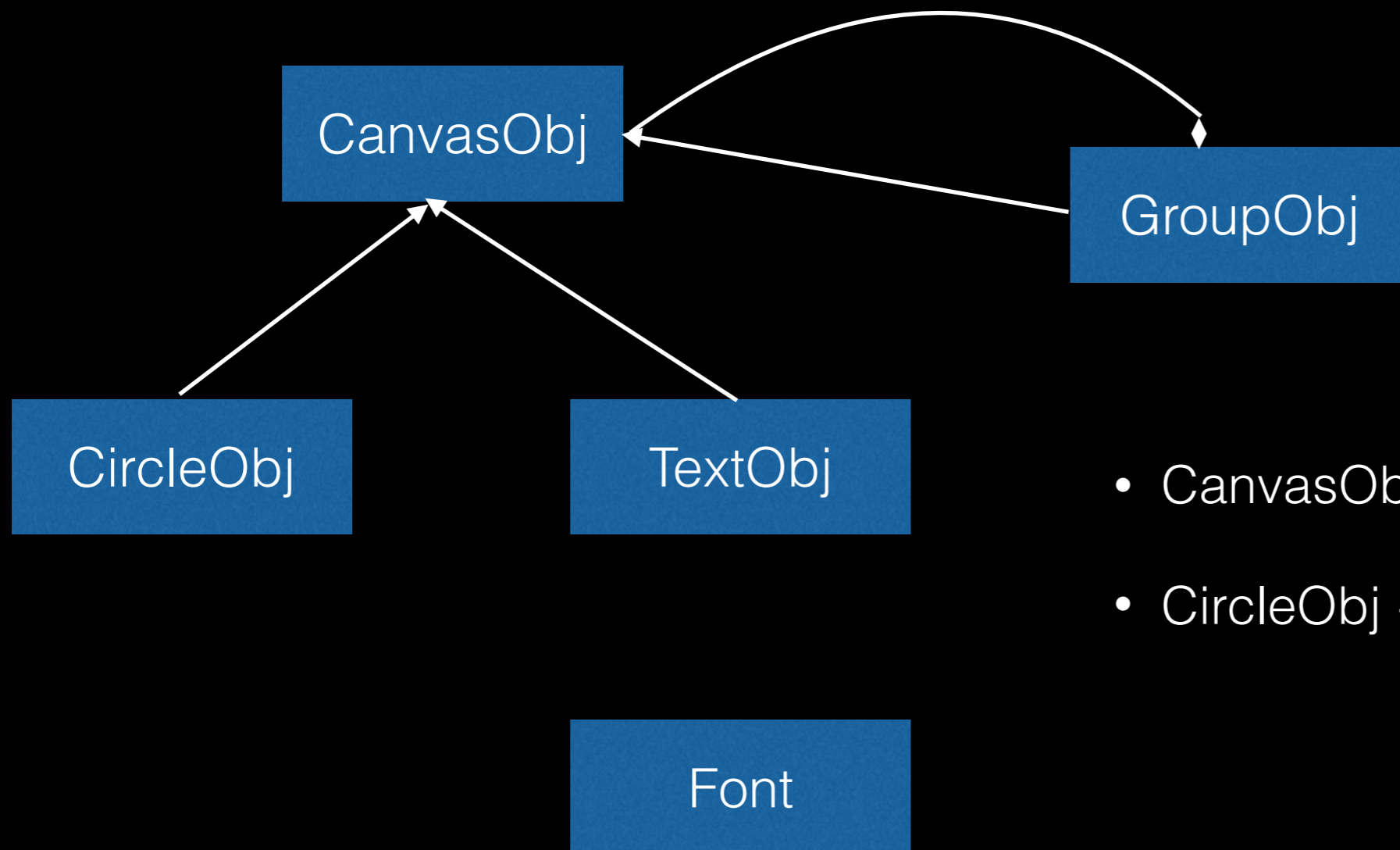
# A Tiny Graphics Library



# A Tiny Graphics Library

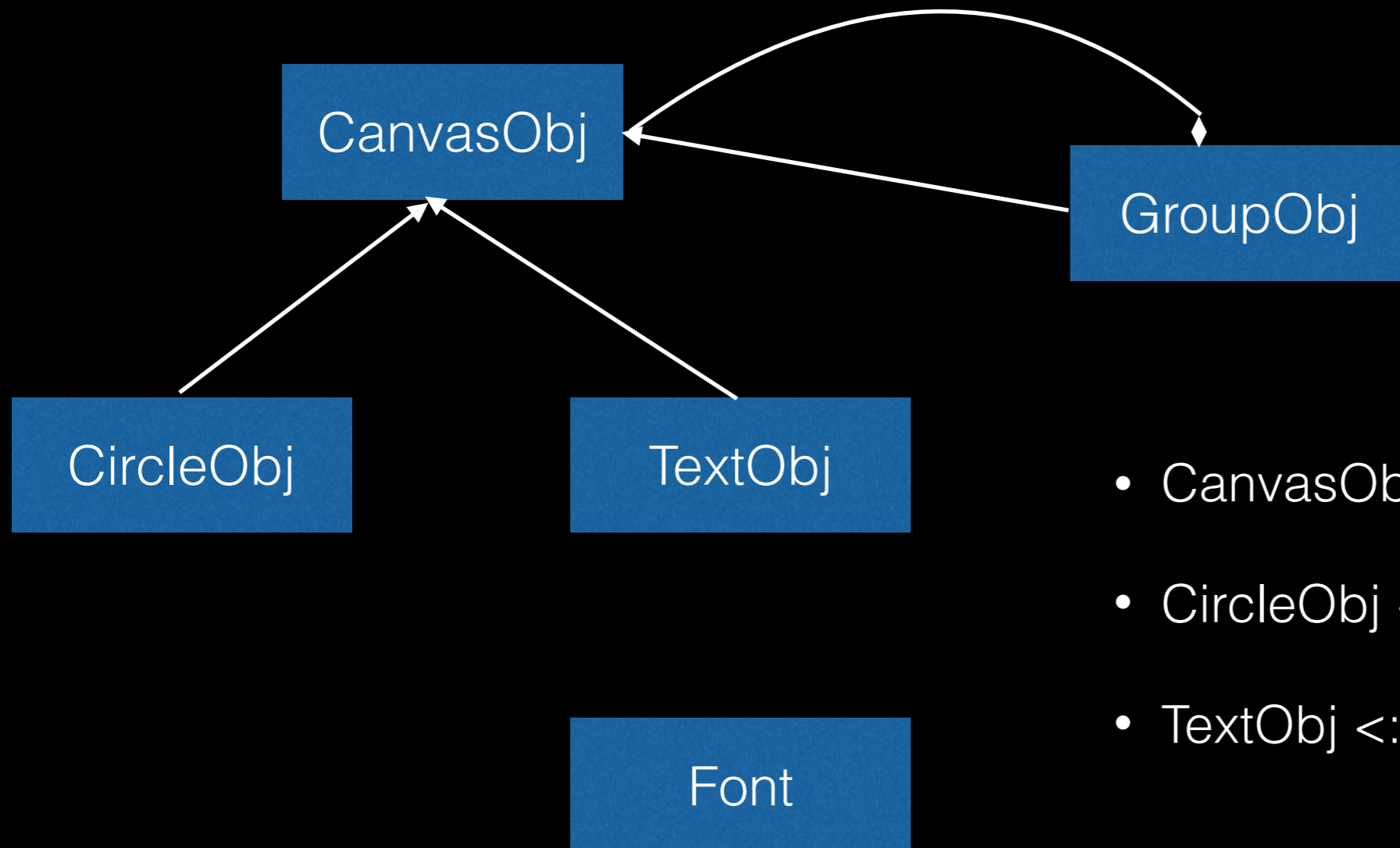


# A Tiny Graphics Library



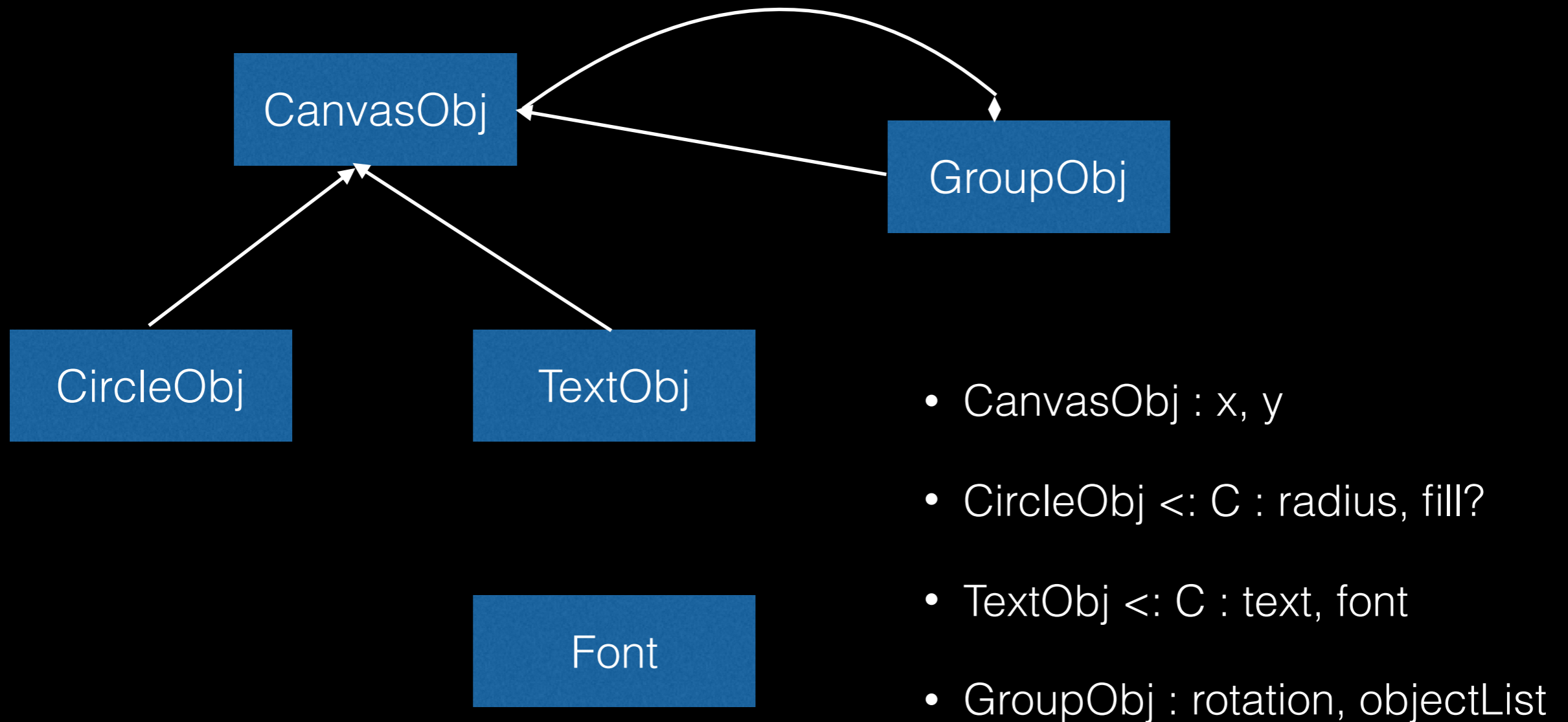
- CanvasObj : x, y
- CircleObj <: C : radius, fill?

# A Tiny Graphics Library

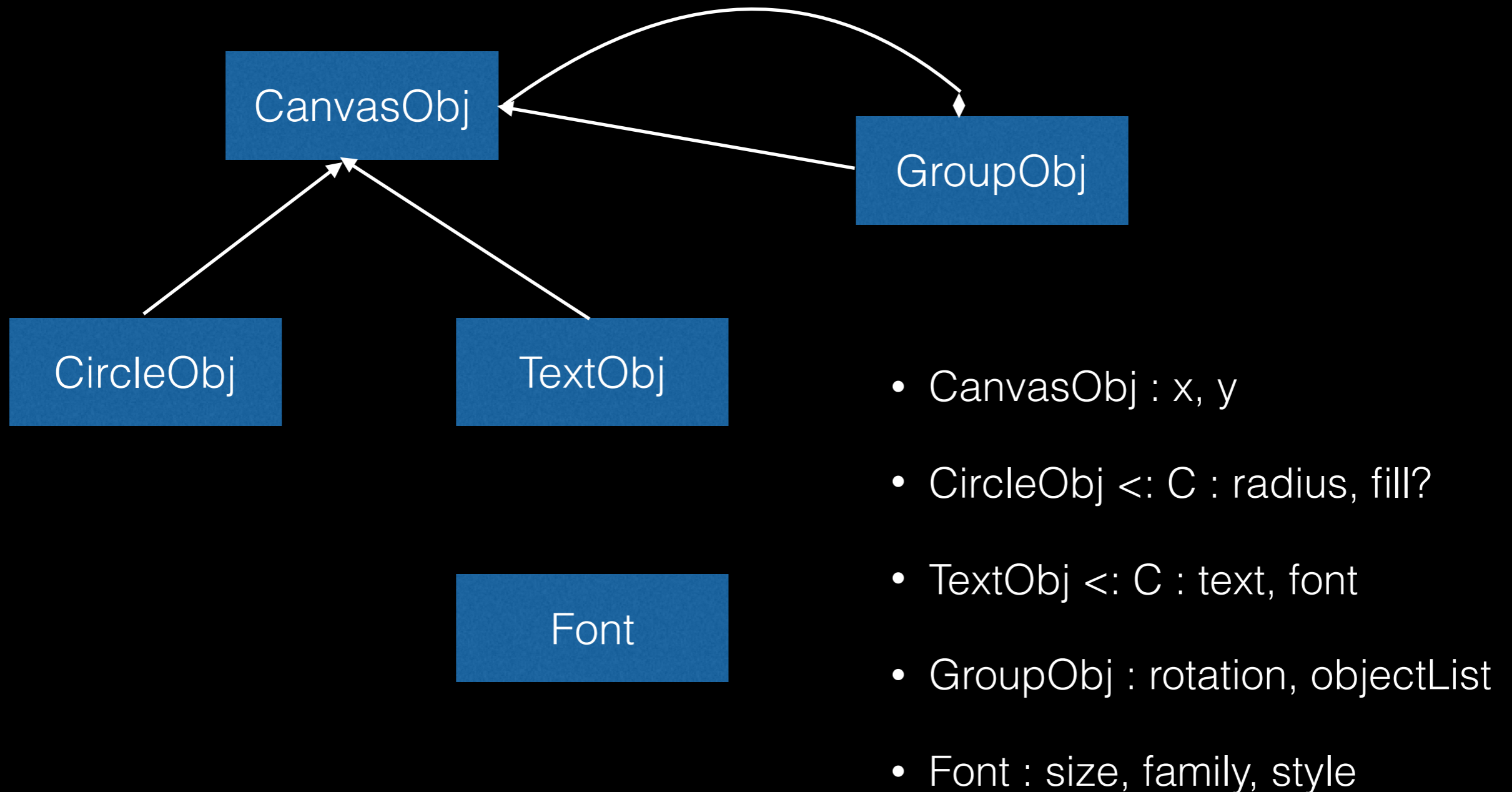


- CanvasObj : x, y
- CircleObj <: C : radius, fill?
- TextObj <: C : text, font

# A Tiny Graphics Library



# 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;  
    }  
}
```

# Level 1: Basics

## Action & Finisher Pattern

```
static class CircleBuilder0 {  
    private CircleObj state = new CircleObj();  
    CircleBuilder0 x(double x) { 1 action pattern  
        state.setX(x);  
        return this;  
    }  
    ...  
    CircleObj build() {  
        return state;  
    }  
}
```



# Level 1: Basics

## Action & Finisher Pattern

```
static class CircleBuilder0 {  
    private CircleObj state = new CircleObj();  
    CircleBuilder0 x(double x) { 1 action pattern  
        state.setX(x);  
        return this;  
    }  
    ...  
    CircleObj build() { 2 finisher pattern  
        return state;  
    }  
}
```

# 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());  
    }  
}
```

# 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)

# レベル表 (因みに)

- Level 0 = Hello world

# レベル表 (因みに)

- Level 0 = Hello world
- Level 1 = Beginner Programmer

# レベル表 (因みに)

- Level 0 = Hello world
- Level 1 = Beginner Programmer
- Level 2 = Usual Programmer

# レベル表 (因みに)

- Level 0 = Hello world
- Level 1 = Beginner Programmer
- Level 2 = Usual Programmer
- Level 3 = Expert



# 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 + ??

# 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

# 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)

```
@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!! (1)

```
@Data
static abstract class CanvasObj {
    double x, y;
}

@Data
static class CircleObj extends CanvasObj {
    double radius;
    boolean fill;
}
```

lombokです

# 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!! (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:

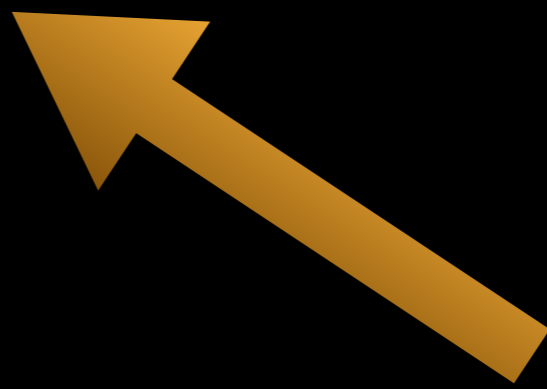
## 継承で遊びましょう!! (3)

```
static class CircleBuilder extends
    CanvasObjBuilder<CircleObj, CircleBuilder> {
    @Getter CircleObj state = new CircleObj();
    CircleBuilder radius(double radius) {
        return self(obj -> obj.setRadius(radius));
    }
    CircleObj build() {
        return state;
    }
}
```

# Level 4:

## 継承で遊びましょう!! (3)

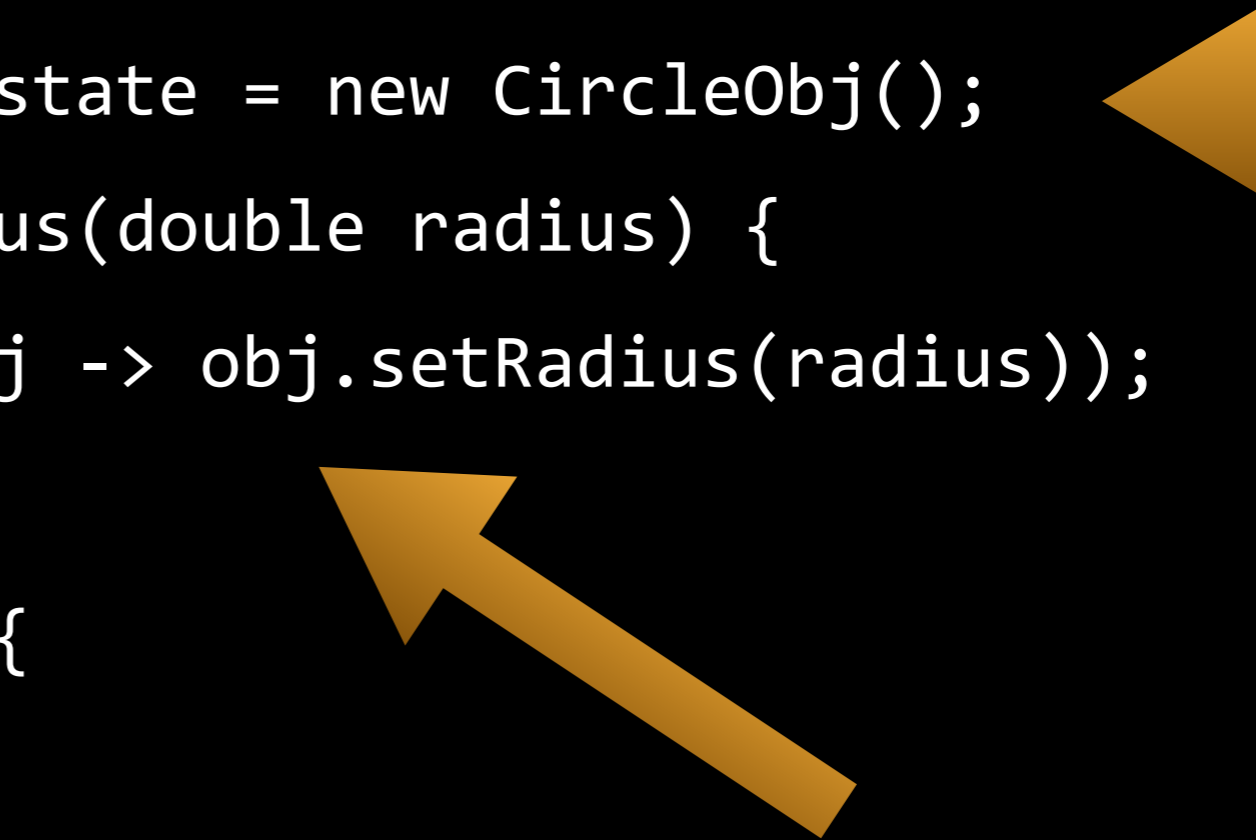
```
static class CircleBuilder extends
    CanvasObjBuilder<CircleObj, CircleBuilder> {
    @Getter CircleObj state = new CircleObj();
    CircleBuilder radius(double radius) {
        return self(obj -> obj.setRadius(radius));
    }
    CircleObj build() {
        return state;
    }
}
```



# Level 4:

## 継承で遊びましょう!! (3)

```
static class CircleBuilder extends
    CanvasObjBuilder<CircleObj, CircleBuilder> {
    @Getter CircleObj state = new CircleObj();
    CircleBuilder radius(double radius) {
        return self(obj -> obj.setRadius(radius));
    }
    CircleObj build() {
        return state;
    }
}
```



Live Coding Session (?)

# Wrap-up

# Wrap-up

- **JAVA**が素晴らしい

# Wrap-up

- **JAVA**が素晴らしい
- Java Genericsが素晴らしい

# Wrap-up

- **JAVA**が素晴らしい
- Java Genericsが素晴らしい
- Fluent Designが素晴らしい



# Wrap-up

- **JAVA**が素晴らしい
- Java Genericsが素晴らしい
- Fluent Designが素晴らしい
- Fluent Designのテクニック

# Wrap-up

- **JAVA**が素晴らしい
- Java Genericsが素晴らしい
- Fluent Designが素晴らしい
- Fluent Designのテクニック

## 1. Action & Finisher Pattern

# Wrap-up

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

# Wrap-up

- **JAVA**が素晴らしい
- Java Genericsが素晴らしい
- Fluent Designが素晴らしい
- 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)

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

(おまけです)

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

# 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);
}
```



