

# Inheritance

Polymorphism, Subtyping, Reuse

YEGOR BUGAYENKO

Lecture #8 out of 8

90 minutes

All visual and text materials presented in this slidedeck are either originally made by the author or taken from public Internet sources, such as website. Copyright belongs to their respected authors.



Polymorphism

Implementation Inheritance

Quiz

Read and Watch

Chapter #1:

# Polymorphism

## Liskov Substitution Principle



“If for each object  $o_1$  of type  $S$  there is an object  $o_2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behavior of  $P$  is unchanged when  $o_1$  is substituted for  $o_2$ , then  $S$  is a subtype of  $T$ ”

— Barbara Liskov, Keynote Address: Data Abstraction and Hierarchy, Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications, 1987

## SOLID (the “L” part)

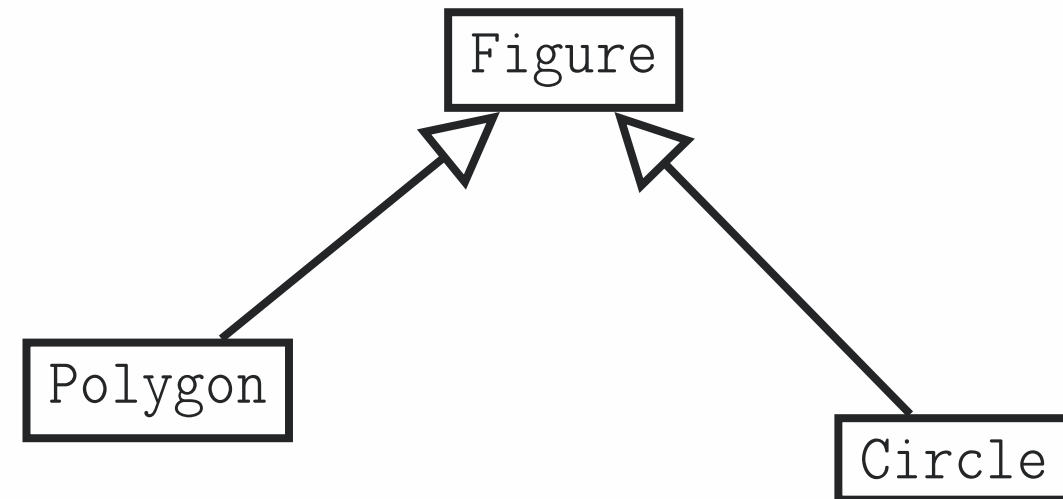


“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it”

— Robert C. Martin, Design Principles and Design Patterns discussing, 2000

## Subtyping

```
1 interface Figure
2     float surface();
3
4 interface Circle extends Figure
5     float perimeter();
6
7 interface Polygon extends Figure
8     int sides();
9
10 void paint(Figure f)
11     float s = f.surface();
12     // ...
```



Circle  $\sqsubseteq$  Figure

Circle <: Figure

## Parametric Polymorphism (Generics)

```
1 class StackOfStrings {  
2     push(String str) // ...  
3     String pop() // ...  
4  
5 class StackOfIntegers {  
6     push(Integer num) // ...  
7     Integer pop() // ...  
8  
9 var s1 = new StackOfStrings();  
10 s1.push("Hello, world!");  
11  
12 var s2 = new StackOfIntegers();  
13 s2.push(42);
```

```
1 class <T> Stack<T> {  
2     push(T item) // ...  
3     T pop() // ...  
4 }  
5  
6 var s1 = new Stack<String>();  
7 s1.push("Hello, world!");  
8  
9 var s2 = new Stack<Integer>();  
10 s2.push(42);
```

## Ad Hoc Polymorphism (Method Overloading)

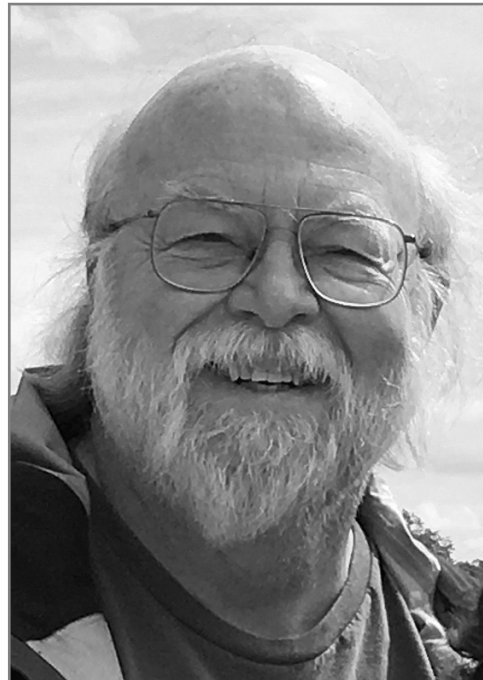
```
1 class Cart {  
2     void add(int pid) // ...  
3     void addString(String pid) {  
4         this.add(Integer.parseInt(pid));  
5     }  
6 }  
7  
8 var c = new Cart();  
9 c.add(42);  
10 c.addString("17");  
11 c.addString("Hello, world!");
```

```
1 class Cart {  
2     void add(int pid) // ...  
3     void add(String pid) {  
4         this.add(Integer.parseInt(pid));  
5     }  
6 }  
7  
8 var c = new Cart();  
9 c.add(42);  
10 c.add("17");  
11 c.add("Hello, world!");
```



Chapter #2:

## Implementation Inheritance



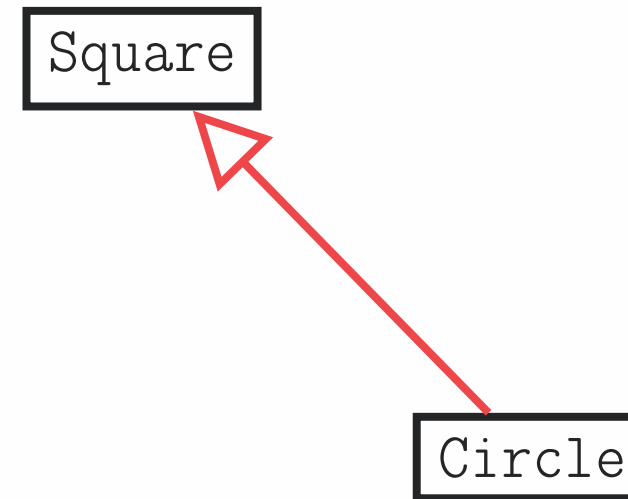
“Someone asked him: “If you could do Java over again, what would you change?” “I’d leave out classes,” he replied”

— James Gosling

Inheriting means “receive (money, property, or a title) as an heir at the death of the previous holder.” Who is dead, you ask? An object is dead if it allows other objects to inherit its encapsulated code and data.

## Code reuse

```
1 class Square
2     private float width;
3     float surface()
4         return width * width;
5
6 class Circle extends Square
7     Circle(float radius)
8         super(radius);
9     @Override float surface()
10        return 3.14 * super.surface();
```



Here, the `Circle` is not a `Square`. It merely reuses the code that was negligently left open in the `Square`.

## Composition over inheritance

### Implementation Inheritance:

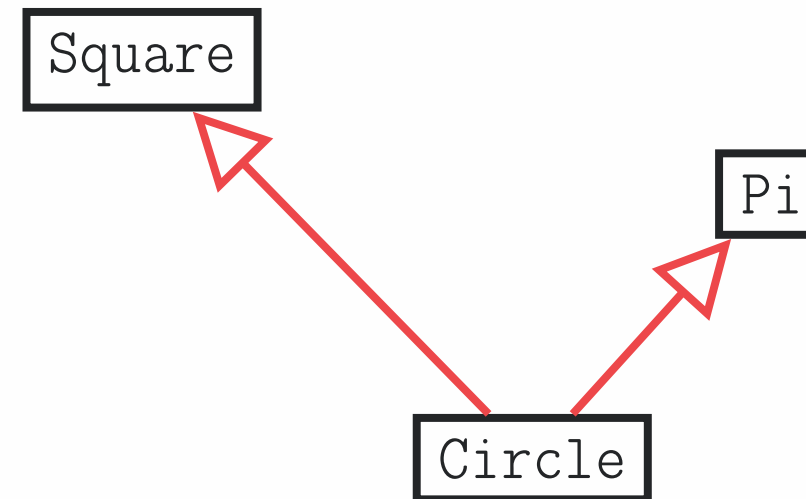
```
1 class Square
2     private float width;
3     float surface()
4         return width * width;
5
6 class Circle extends Square
7     Circle(float radius)
8         super(radius);
9     @Override float surface()
10        return 3.14 * super.surface();
```

### Composition:

```
1 final class Square
2     private float width;
3     float surface()
4         return width * width;
5
6 final class Circle
7     private Square s;
8     Circle(float radius)
9         this.s = new Square(radius);
10    float surface()
11        return 3.14 * s.surface();
```

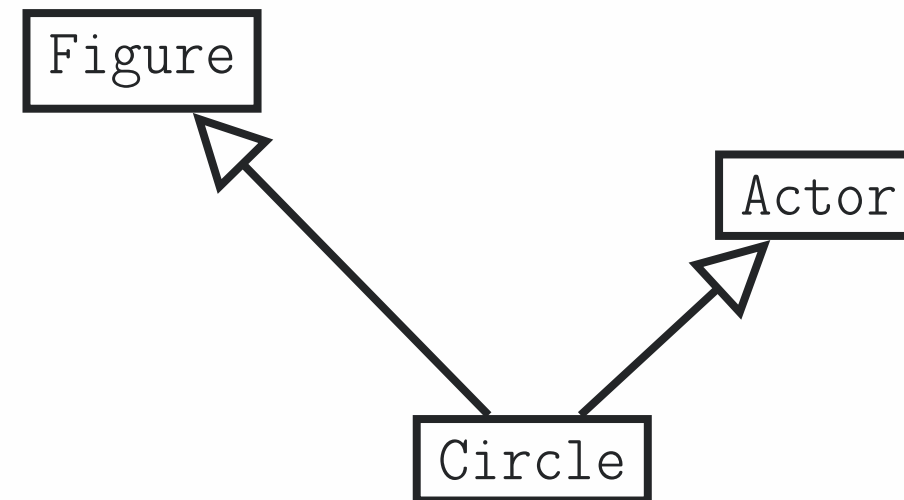
## Multiple inheritance

```
1 class Pi
2     float value()
3     return 3.1415926;
4
5 class Square
6     private float width;
7     float surface()
8     return width * width;
9
10 class Circle extends Square, Pi
11     Circle(float r): Square(r), Pi() {}
12     virtual float surface()
13     return Pi.value() * Square.surface();
```



## Multiple super types

```
1 interface Actor
2     void move(int dx, int dy);
3
4 interface Figure
5     float surface();
6
7 class Circle implements Figure, Actor
8     Circle(float r)
9     @Override float surface()
10         // ...
11     @Override void move(int dx, int dy)
12         // ...
```



## Chapter #3:

# Quiz



I show this code to job interview candidates, asking them to find as many defects in it as they can: [yegor256/quiz](#) (Java).

How many problems you can find in this code?

Chapter #4:

Read and Watch

Why extends is evil by Allen Holub (2003)

Inheritance Is a Procedural Technique for Code Reuse by me (2016)

Inheritance vs. Subtyping (Webinar #24) by me (2017)