Inheritance

Polymorphism, Subtyping, Reuse

Yegor Bugayenko

Lecture #8 out of 8
90 minutes

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   // ...
```
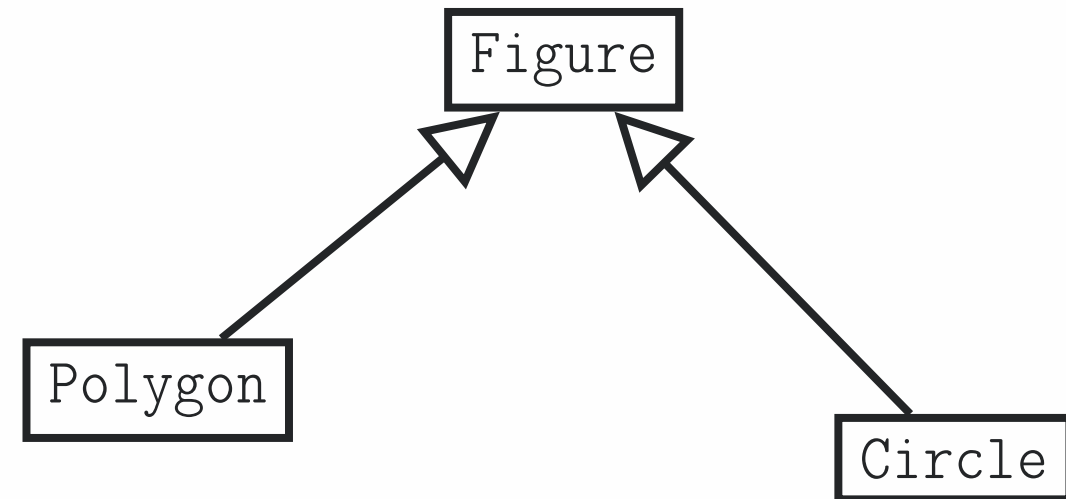

$Circle \sqsubseteq Figure$

$Circle <:\ Figure$

Parametric Polymorphism (Generics)

```
1  class StackOfStrings {
2    void push(String str) // ...
3    String pop() // ...
4
5  class StackOfIntegers {
6    void 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    void 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);
```

[LSP SOLID Subtyping Generics Overloading]

# 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!");
```
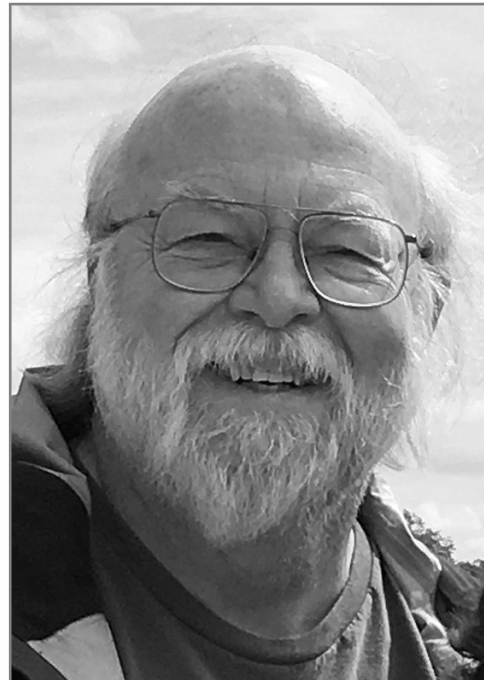
Chapter #2:

Implementation Inheritance
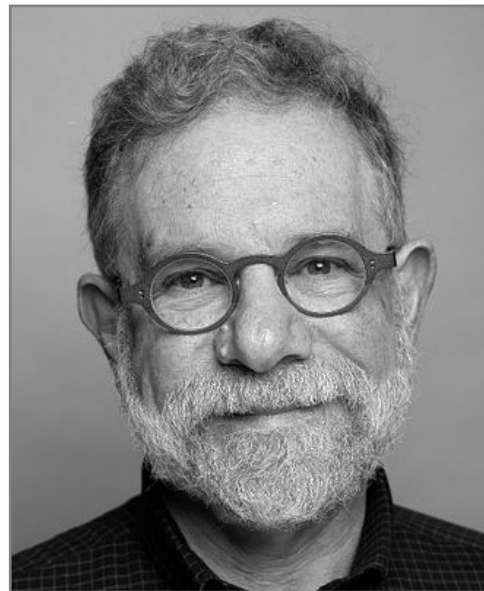
"The `extends` keyword is evil; maybe not at the Charles Manson level, but bad enough that it should be shunned whenever possible."

— Allen Holub, Why extends is evil, InfoWorld, 2003

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

— James Gosling

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();
```
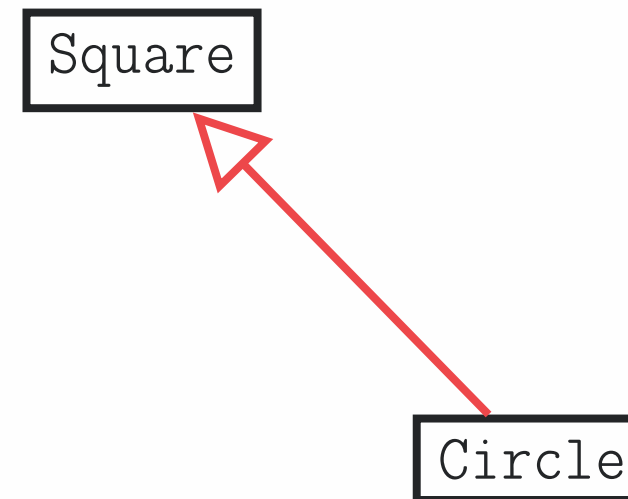
Square

Circle

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

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.

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();
```

All classes, without exceptions, should be either `final` or `abstract`

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();
```

[Reuse Composition Multiple Parents]
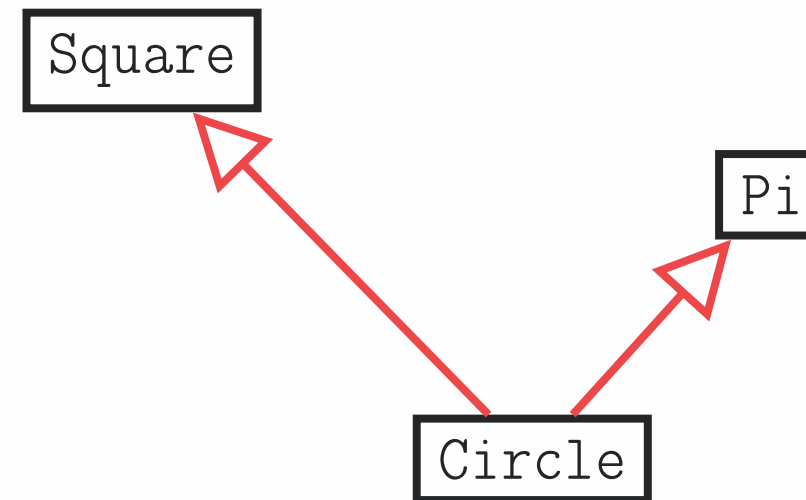
Multiple super types

```
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     // ...
```
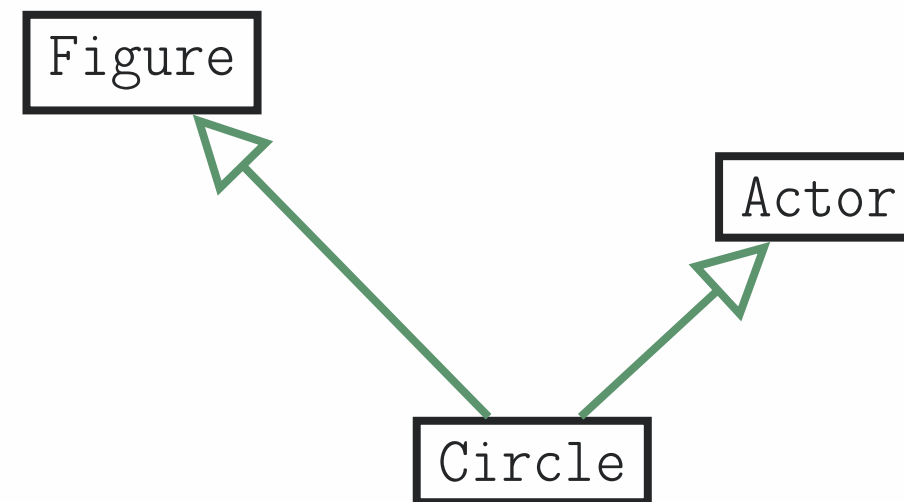
Chapter #3:
Quiz

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?

```
1  public class Parser {
2    private File file;
3    public synchronized void setFile(File f) {
4      file = f;
5    }
6    public synchronized File getFile() {
7      return file;
8    }
9    public String getContent() throws IOException {
10     FileInputStream i = new FileInputStream(file);
11     String output = "";
12     int data;
13     while ((data = i.read()) > 0) {
14       output += (char) data;
15     }
16     return output;
17   }
18   public String getContentWithoutUnicode()
19     throws IOException {
20     FileInputStream i = new FileInputStream(file);
21     String output = "";
22     int data;
23     while ((data = i.read()) > 0) {
24       if (data < 0x80) {
25         output += (char) data;
26       }
27     }
28     return output;
29   }
30   public void saveContent(String content) {
31     FileOutputStream o = new FileOutputStream(file);
32     try {
33       for (int i = 0; i < content.length(); i += 1) {
34         o.write(content.charAt(i));
35       }
36     } catch (IOException e) {
37       e.printStackTrace();
38     }
39   }
40 }
```

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)

References