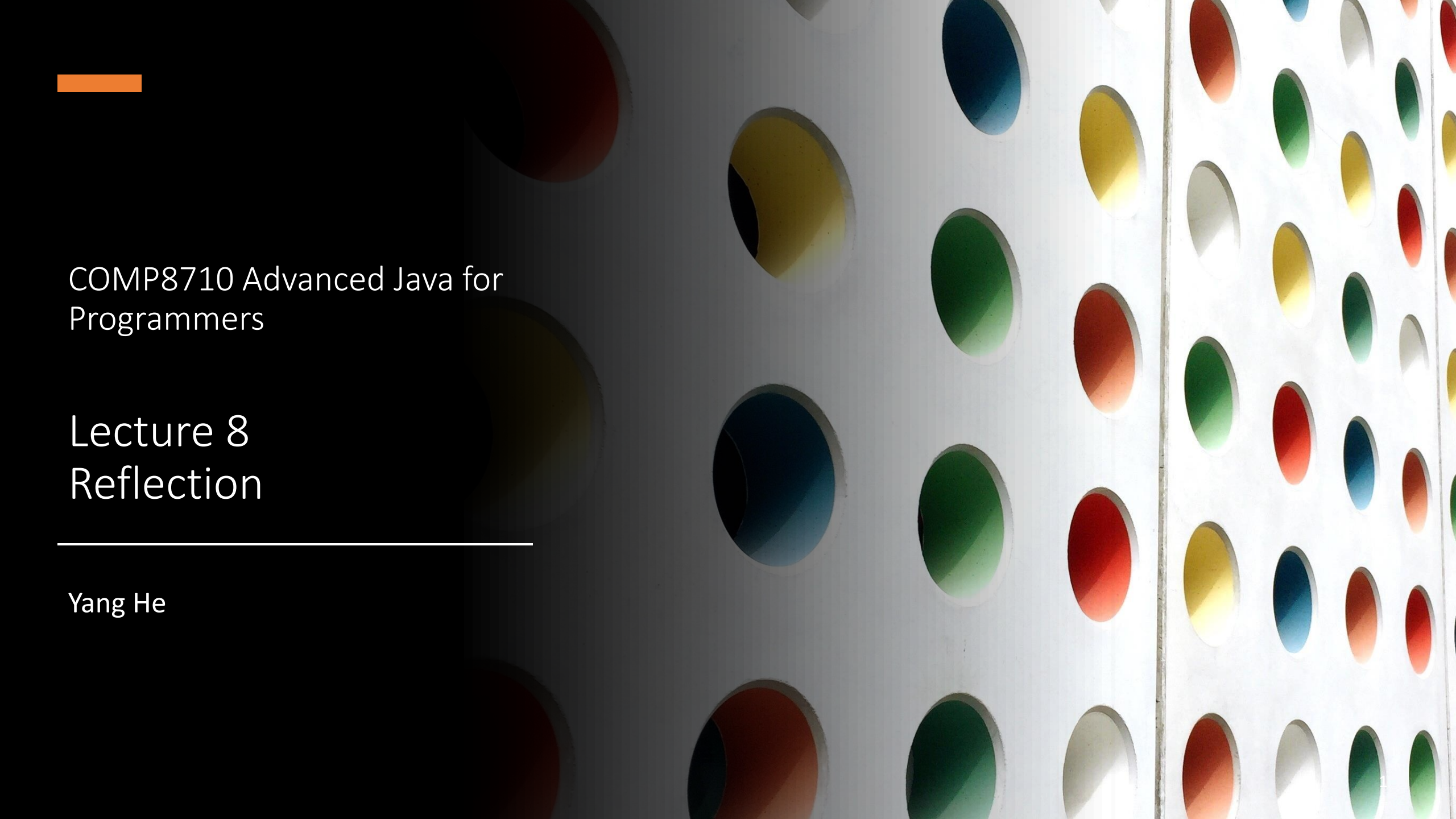


COMP8710 Advanced Java for Programmers

Lecture 8
Reflection

Yang He

Topics

- The Class class
- Generic arrays
- Object cloning
- Reflection details

Reflection

- Reflection is the capability of an object to know things about itself
- Even the keywords this and super indicate forms of reflection
  - this means that an object “knows” its own address in the memory
  - super means that it “knows” (parts of) its super-class
- Most other forms of reflection are tied to the class named Class

The Class class

- In a proper OO language just about anything is an object
- This means that classes are objects
  - In Java class String as an object is expressed as String.class
- If classes are objects, they must belong to some class
- This is the class named Class (java.lang.Class)
  - Since Java version 5 this is more specific: String.class belongs to the class Class<String>

What are its capabilities?

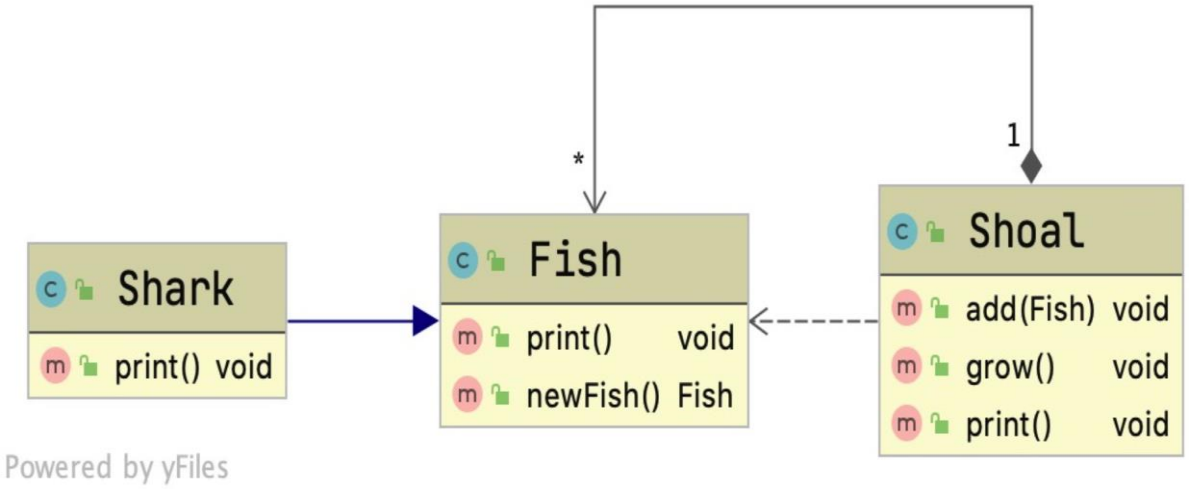
- Inspect and manipulate classes without knowing their name, methods, fields, etc.
- Create new instances of a class
- Enquire about methods, fields, constructors, etc.
- Call these methods, read/write to the fields, etc.
- Access private fields (boo! hiss!)

What is this good for?

- Creating tools that work with Java
  - IDEs: IntelliJ knows, e.g. which methods are available for an object
- Building frameworks/libraries
  - Working with user-defined classes that you don’t know in advance
- Can re-implement generic shallow cloning or different forms of serialization
  - E.g. convert a Java object to JSON
- Automatic testing
  - Automatically generate test cases for a method

Making new objects (1)

- It can often be useful to make new objects when the type is not known statically
- Assume we have a sea animal simulation program



Making new objects (2)

- E.g. We want to add a single kind of fish in a Shoal, and then grow it with the same types of fish

```
public class Shoal {
   private final List<Fish> fishList = new ArrayList<>();
   public void grow() {
      Fish fish = createNewFish(fishList.get(0));
      add(fish);
   }
   // ...
}

Shoal shoal = new Shoal();
shoal.add(new Shark());
shoal.grow();     // Grow with more Shark objects
```

Making new objects (3)

- Reflection allows us to create objects without knowing exactly what class we are using
- E.g.

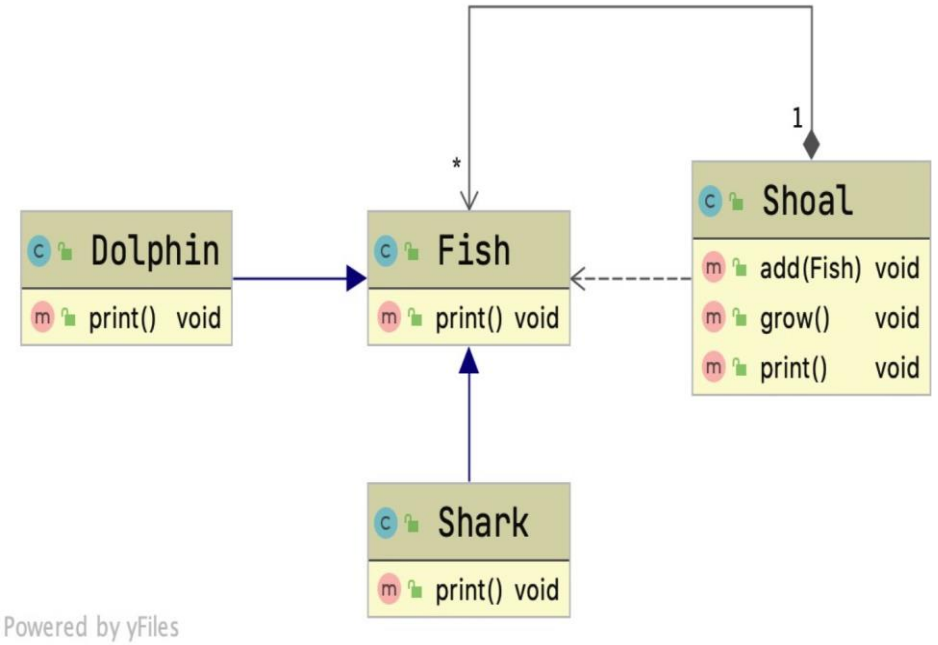
```
private Fish createNewFish(Fish fish) throws
                NoSuchMethodException, InstantiationException,
                InvocationTargetException, IllegalAccessException
{
    return fish.getClass()
               .getDeclaredConstructor()
               .newInstance();
}
```

  - *The getClass call will get the class (of this or the argument)*
  - *The getDeclaredConstructor returns the Constructor object of that class*
  - *The newInstance call will create a new instance of the class, using the constructor with no arguments*

Making new objects (4)

- We don’t need to change anything in the Shoal class when adding other classes
- E.g. grow with more Dolphin objects

```
Shoal shoal2 = new Shoal();
shoal2.add(new Dolphin());
shoal2.grow();
```



Making new objects (5)

- We can also make new instances if we know the fully qualified name of a class
- E.g.

```
public Object factory(String className) throws
                        ClassNotFoundException, NoSuchMethodException,
                        IllegalAccessException, InstantiationException,
                        InvocationTargetException
{
    return Class.forName(className)
                .getDeclaredConstructor()
                .newInstance();
}

var shoal = new Shoal();
var shark = (Shark) shoal.factory("kent.co871.Shark");
shoal.add(shark);
```

Arrays

- Working with arrays in a generic way can be tricky
- Arrays are themselves objects of the same Array class, but different arrays hold data of different types
- Through reflection we can retrieve the component type of an array
- E.g.

```
T[] arr = new T[2];
Class clazz = arr.getClass().getComponentType();
```

Growing an array, generically

- E.g. double the size of an array

```
public static Object doubleArray(Object arr) throws NotAnArray {
    Class clazz = arr.getClass().getComponentType();
    if (clazz == null) throw new NotAnArray();
    int len = Array.getLength(arr);
    Object res = Array.newInstance(clazz, len * 2);
    System.arraycopy(arr, 0, res, 0, len);
    return res;
}
```

  - *The Array.newInstance() call will create a new array*
  - *New array will have component type that is the same as the array we started with*
  - *We could copy the array using the static get/set methods of the class Array (Array.get and Array.set), but System.arraycopy is more efficient*

Object cloning (1)

- To clone an object, i.e. make an exact copy of the object:
  - Create a new instance of the same class, and
  - Copy the contents of all the fields
- In Java we can use Object.clone() method to make a shallow copy of an object
- E.g.

```
Book book = new Book("Java");
Book copy = (Book) book.clone();
```

- To use the method clone(), the class of the object (e.g. Book) must implement Cloneable interface, otherwise it will throw CloneNotSupportedException

Object cloning (2)

- E.g. class Point2D implements Cloneable
- We may want a moveX method that creates a new instance of Point2D with a changed x coordinate
- Problem: we may be in a subclass of Point2D with further fields (e.g. pixel colour)

```
public class Point2D implements Cloneable
{
    private int x, y;

    public Point2D moveY(int newY) {
        try {
            Point2D res = (Point2D) this.clone();
            res.y = newY;
            return res;
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("Clone failed");
        }
    }
    // ...
}
```

Object cloning (3)

- E.g. ColouredPoint is a subclass of Point2D

```
Point2D pt1 = new ColouredPoint(1, 5, Color.RED);
ColouredPoint pt2 = (ColouredPoint) pt1.moveX(2);
System.out.println("pt2: " + pt2);
```

- Note: the method moveX returns an object
  - Its static type is Point2D
  - Its dynamic type is the class of the object that invokes the method

Reflection – Fields

- E.g.

```
// get dynamic type
Class clazz = pt1.getClass();
System.out.println("pt1 dynamic type: " + clazz);

// get all fields declared in the class
Field[] fields = clazz.getDeclaredFields();
for (Field f : fields) {
    System.out.println(f);
}
```

Reflection – Methods

- E.g.

```
// get all public methods (incl. inherited)
Method[] methods = clazz.getMethods();
for (var m : methods) {
    printMethodSignature(m);
}
private static void printMethodSignature(Method m) {
    System.out.print(m.getReturnType().getName() + " ");
    System.out.print(m.getName());
    Class[] params = m.getParameterTypes();
    System.out.print("(");
    for (Class p : params)
        System.out.print(p.getName() + " ");    // print parameter type
    System.out.println(")");
}
```

Reflection – Invoke a method

- E.g. use invoke() of Method

```
try {
    // method without parameter
    var getX = clazz.getMethod("getX");
    var x = (int) getX.invoke(pt1);
    System.out.println("pt1 x=" + x);

    // method with parameter
    var moveX = clazz.getMethod("moveX", int.class);
    var pt3 = (ColouredPoint) moveX.invoke(pt1, 3);
    System.out.println("pt3 x=" + getX.invoke(pt3));
} catch (NoSuchMethodException | IllegalAccessException |
         InvocationTargetException e) {
    e.printStackTrace();
}
```

Summary

- Reflection is powerful but can be dangerous
  - Code is difficult to maintain
  - Can introduce bugs that are difficult for tools to detect
  - Don’t use it extensively unless there is a good reason to do it
- Where to use it
  - Building APIs, libraries, or frameworks
  - Supporting plug-ins for your application
  - More efficient coding
    - E.g. taking advantage of class name patterns (see previous slide)