

CS2030 Lecture 7

SOLID Principles

Packaging and Exception Handling

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2023 / 2024

SOLID Principles in OO Design

- Single responsibility principle:
a class should have only one reason to change
— Robert C. Martin (Uncle Bob)
- Liskov substitution principle:
Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T .
— Barbara Liskov
- If S is a subtype of T (denoted $S <: T$), then an object of type T can be replaced by that of type S without changing the desirable property of the program

Outline and Learning Outcome

- Understand the **SOLID principles** and their application in the design of object-oriented software
- Be able to create packages and use the appropriate access modifiers
- Be able to employ exception handling to deal with “exceptional” events
 - Understand the use of **try-catch-finally** clauses
 - Able to distinguish the different types of exceptions
 - Able to appreciate exception control flow

SOLID Principles in OO Design

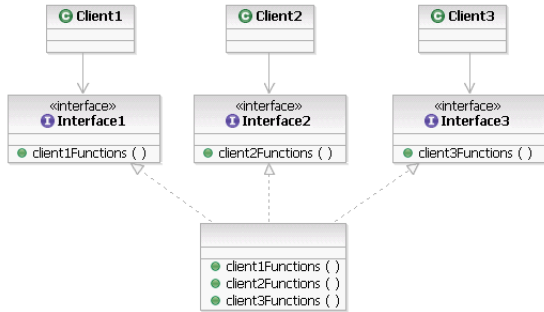
- Open-closed principle:
classes should be open for extension, but closed for modification
— Bertrand Meyer
- ```
jshell> class A { void foo() { } }
| created class A
jshell> void client(A a) { a.foo(); }
| created method client(A)
jshell> client(new A())
jshell> class B extends A { }
| created class B
jshell> class C extends A { @Override void foo() { } }
| created class C
jshell> class D extends B { @Override void foo() { } }
| created class D
jshell> client(new B()) // client does not need modification
jshell> client(new C()) // C:foo() invoked
jshell> client(new D()) // D:foo() invoked
```
-

# SOLID Principles in OO Design

- Interface segregation principle:  
*no client should be forced to depend on methods it does not use.*

— Uncle Bob

```
jshell> Circle circle = new Circle(1)
circle ==> Circle with radius 1
jshell> void client1(Shape s) {
...> s.getArea();
...> }
| created method client1(Shape)
jshell> void client2(Scalable k) {
...> k.scale(2);
...> }
| created method client2(Scalable)
jshell> client1(circle)
jshell> client2(circle)
jshell> void client3(Scalable k) {
...> k.getArea(); // ???
...> }
```



# Creating Packages

- Include the **package** statement at the top of all source files that reside within the package, e.g.  
**package** cs2030.test;
- Include the **import** statement to source files outside the package, e.g.  
**import** cs2030.test.SomeClass;
- Compile the Java files using  
`$ javac -d . *.java`
- cs2030/test directory created with same-package class files stored within

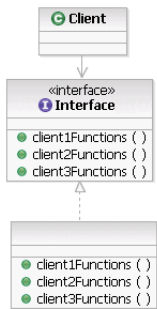
| Access Modifiers ->     | Most Restrictive ← → Least Restrictive |                   |           |        |
|-------------------------|----------------------------------------|-------------------|-----------|--------|
|                         | private                                | Default/no-access | protected | public |
| Inside class            | Y                                      | Y                 | Y         | Y      |
| Same Package Class      | N                                      | Y                 | Y         | Y      |
| Same Package Sub-Class  | N                                      | Y                 | Y         | Y      |
| Other Package Class     | N                                      | N                 | N         | Y      |
| Other Package Sub-Class | N                                      | N                 | Y         | Y      |

# SOLID Principles in OO Design

- Dependency inversion principle:  
*Program to an interface, not an implementation.*

— GoF

```
jshell> /list Shape
1 : interface Shape { // Shape is the contract
 double getArea();
}
jshell> Shape s = new Circle(1)
s ==> Area 3.14 and perimeter 6.28
jshell> class Circle implements Shape { // Circle follows contract specs
...> private final int radius;
...> public double getArea() {
...> return Math.PI * this.radius * this.radius;
...> }
| created class Circle
jshell> void client(Shape s) { // client codes according to contract
...> double area = s.getArea();
...> }
| created method client(Shape)
jshell> client(circle)
```



# Access Modifiers and Their Accessibility

```
==> Base.java <==
package cs2030.test;
public class Base {
 private void foo() { } // -
 protected void bar() { } // #
 void baz() { } // ~
 public void qux() { } // +
 private void test() {
 this.foo();
 this.bar();
 this.baz();
 this.qux();
 }
}

==> InsidePackageClient.java <==
package cs2030.test;
class InsidePackageClient {
 private void test() {
 Base b = new Base();
 b.bar();
 b.baz();
 b.qux();
 }
}

==> InsidePackageSubClass.java <==
package cs2030.test;
class InsidePackageSubClass extends Base {
 private void test() {
 super.bar();
 super.baz();
 super.qux();
 }
}

==> OutsidePackageClient.java <==
import cs2030.test.Base;
class OutsidePackageClient {
 private void test() {
 Base b = new Base();
 b.qux();
 }
}

==> OutsidePackageSubClass.java <==
import cs2030.test.Base;
class OutsidePackageSubClass extends Base {
 private void test() {
 super.bar();
 super.qux();
 }
}
```

## Preventing Inheritance and Overriding

- The **final** keyword can be applied to methods or classes
  - Use the **final** keyword to explicitly prevent inheritance

```
final class Circle {
 :
}
```
  - To allow inheritance but prevent overriding

```
class Circle implements Shape {
 :
 @Override
 final double getArea() {
 :
 }
 :
 @Override
 final double getPerimeter() {
 :
 }
}
```

9 / 16

## Handling Exceptions

- Method #1: **throws** the exception out of the method

```
public static void main(String[] args) throws FileNotFoundException {
```
- Method #2: **handle** the exception within the method

```
try {
 FileReader file = new FileReader(args[0]);
 Scanner sc = new Scanner(file);
 List<Point> points = new ArrayList<Point>();
 while (sc.hasNext()) {
 points.add(new Point(sc.nextDouble(), sc.nextDouble()));
 }
 DiscCoverage maxCoverage = new DiscCoverage(points);
 System.out.println(maxCoverage);
} catch (FileNotFoundException ex) {
 System.err.println("Unable to open file " + args[0] + "\n" + ex);
}
```

  - **try** block encompasses the business logic
  - **catch** block encompasses exception handling logic

11 / 16

## Error Handling

- Use exceptions to track reasons for program failure, e.g.

```
public static void main(String[] args) {
 FileReader file = new FileReader(args[0]);
 Scanner sc = new Scanner(file);
 List<Point> points = new ArrayList<Point>();
 while (sc.hasNext()) {
 points.add(new Point(sc.nextDouble(), sc.nextDouble()));
 }
 DiscCoverage maxCoverage = new DiscCoverage(points);
 System.out.println(maxCoverage);
}
```

  - Filename missing or misspelt
  - The file contains a non-numerical value
  - The file provided contains insufficient numerical values
- Compiling the above gives the following compilation error:  
Main1.java:12: error: unreported exception FileNotFoundException;  
must be caught or declared to be thrown  
 FileReader file = new FileReader(args[0]);  
 ^

10 / 16

## Catching Multiple Exceptions

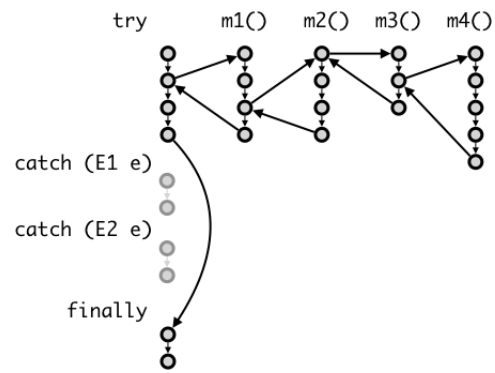
- Multiple catch blocks ordered by *most specific exceptions first*

```
try {
 FileReader file = new FileReader(args[0]);
 Scanner sc = new Scanner(file);
 List<Point> points = new ArrayList<Point>();
 while (sc.hasNext()) {
 points.add(new Point(sc.nextDouble(), sc.nextDouble()));
 }
 DiscCoverage maxCoverage = new DiscCoverage(points);
 System.out.println(maxCoverage);
} catch (FileNotFoundException ex) {
 System.err.println("Unable to open file " + args[0] + "\n" + ex);
} catch (ArrayIndexOutOfBoundsException ex) {
 System.err.println("Missing filename");
} catch (NoSuchElementException ex) { // includes InputMismatchException
 System.err.println("Incorrect file format\n");
} finally {
 System.out.println("Program Terminated\n");
}
```
- Optional **finally** block used for house-keeping tasks
- Multiple exceptions (no sub-classing) in a single catch using |

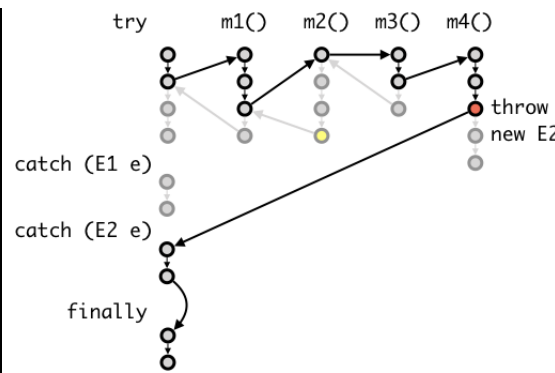
12 / 16

## Normal vs Exception Control Flow

- E.g. **try-catch-finally** block (m1 is called, m1 calls m2, m2 calls m3, m3 calls m4), and catching two exceptions E1, E2



Normal Control Flow



Exception Control Flow

13 / 16

## Types of Exceptions

- There are two types of exceptions:
  - A **checked exception** is one that the programmer is expected to actively anticipate and handle
    - all checked exceptions should be caught (**catch**) or propagated (**throw**)
    - e.g. when opening a file, `FileNotFoundException` should be explicitly handled
  - An **unchecked exception** is one that is unanticipated, usually the result of a bug in the program
    - e.g. `ArithmeticException` surfaces when trying to divide by zero

15 / 16

## Throwing an Exception

- An exception can be created and thrown using **throw**

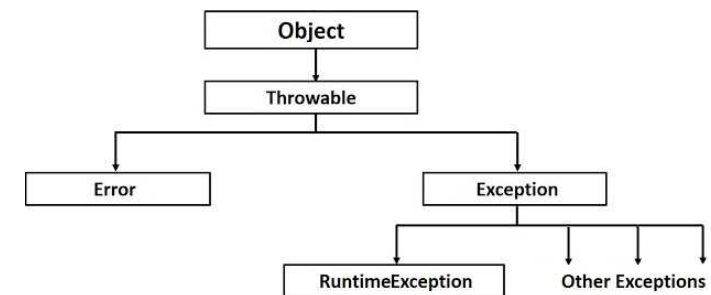
```
Circle createUnitCircle(Point p, Point q) {
 double distPQ = p.distanceTo(q);
 if (distPQ < EPSILON || distPQ > 2.0 + EPSILON) {
 throw new IllegalArgumentException("Distance pq not within (0, 2]");
 }
 ...
 return new Circle(...);
}
```
- Creating a user defined exception to be thrown

```
class IllegalCircleException extends IllegalArgumentException {
 IllegalCircleException(String message) {
 super(message);
 }
 @Override
 public String toString() {
 return "IllegalCircleException: " + getMessage();
 }
}
```
- Only create your own exceptions if there is a good reason to do so, else just find one that suits your needs

14 / 16

## Exception Hierarchy

- Unchecked exceptions are sub-classes of `RuntimeException`



- When overriding a method that throws a checked exception, the overriding method cannot throw a more general exception
- Avoid *Pokemon Exception Handling*, **catch** (`Exception ex`)
- Handle exceptions at the appropriate abstraction level, do not just throw and break the abstraction barrier

16 / 16