CS2030 Lecture 6

Exception Handling and Other Java Constructs

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

Semester 1 2022 / 2023

Outline and Learning Outcome

- 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
- Understand the use of **`static`**, **`enum`** and **`final`** keywords under different usage contexts
- Be able to create packages and use the appropriate access modifiers

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]);
                          ^
```

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

Catching Multiple Exceptions

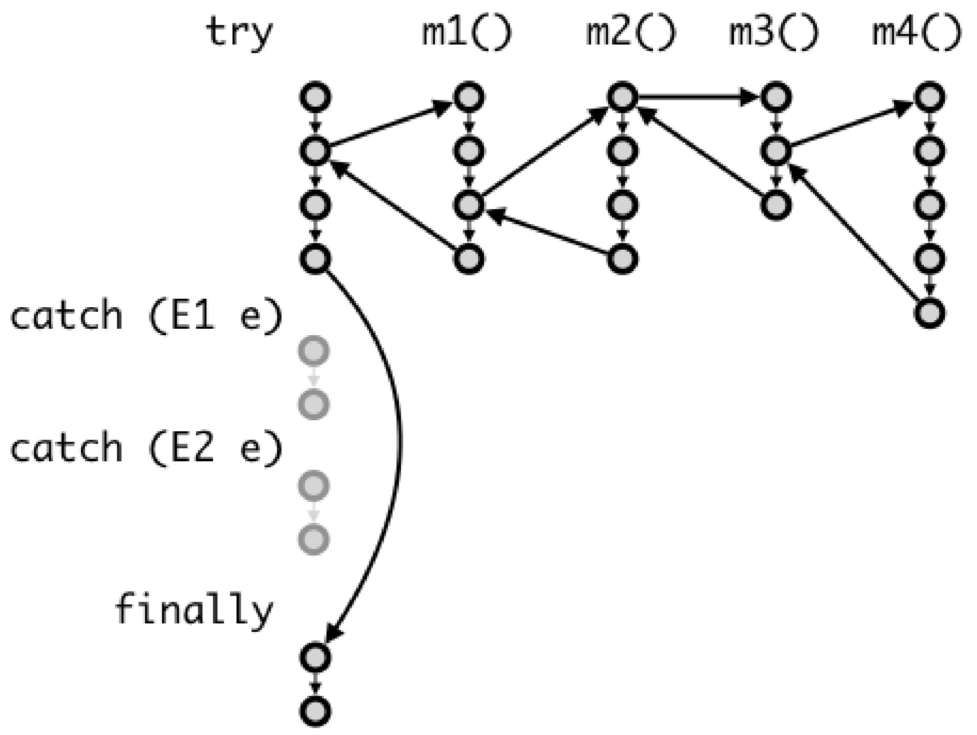
- Multiple catch blocks ordered by *most specific exceptions first*

```
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");
}
```

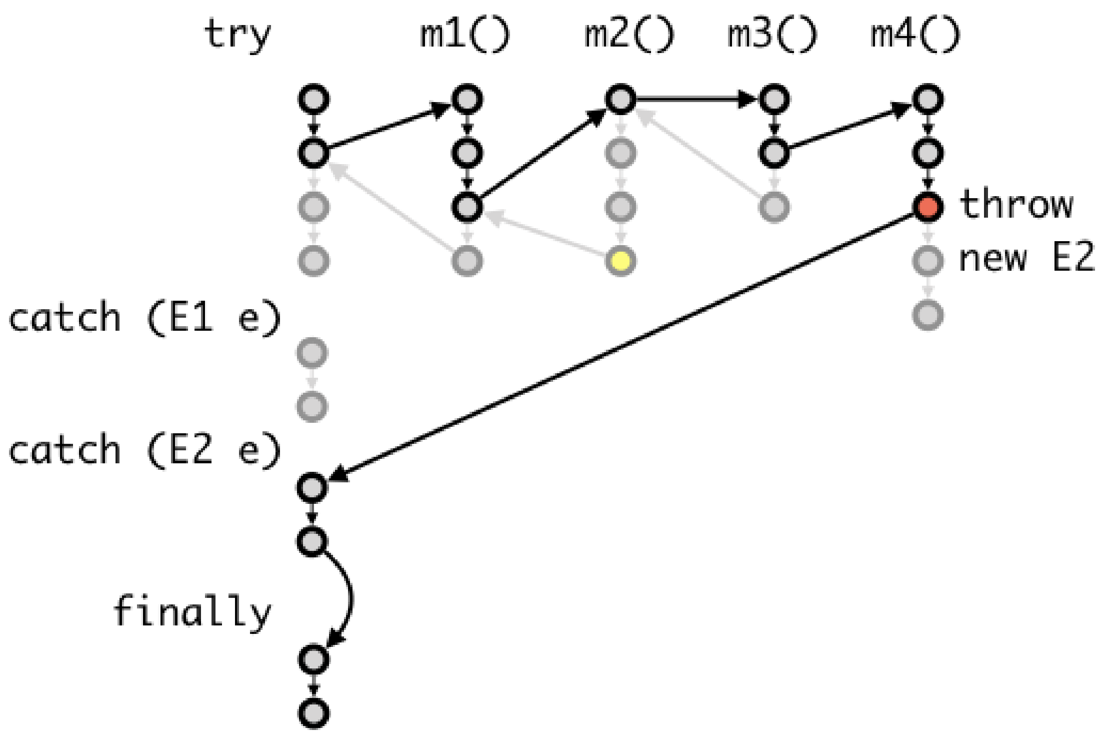
- Optional **`finally`** block used for house-keeping tasks
- Multiple exceptions (no sub-classing) in a single catch using |

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

Throwing an Exception

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

```java
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]");
    } else {
        ...
    }
}
```

- Creating a user defined exception to be thrown

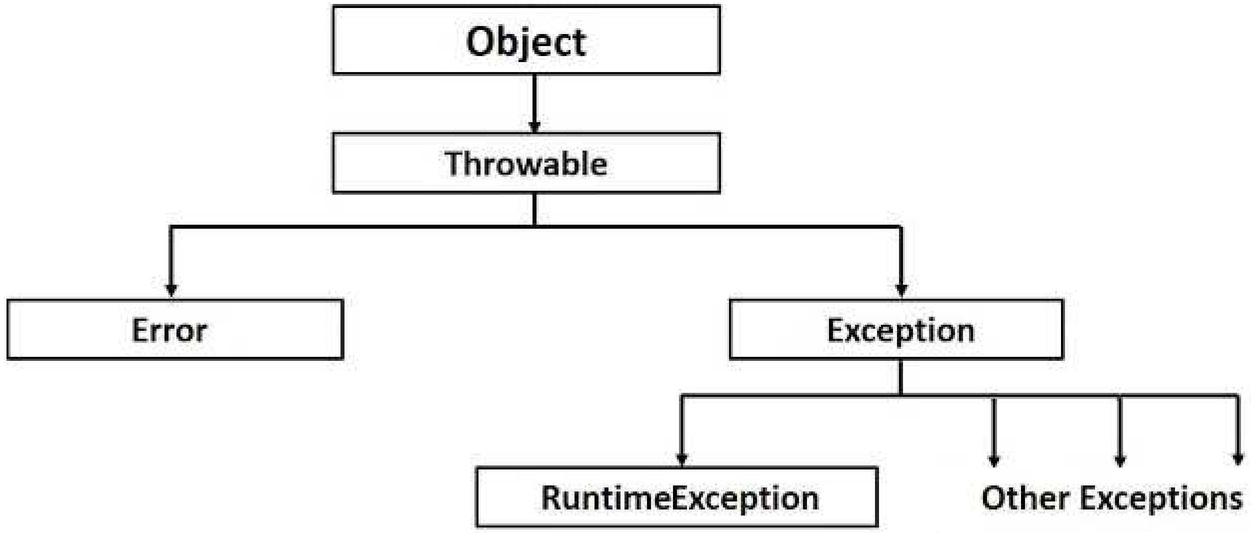
```java
class IllegalCircleException extends IllegalArgumentException {
    IllegalCircleException(String message) {
        super(message);
    }

    @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

Types of Exceptions

- There are two types of exceptions:
  - A **checked exception** is one that the programmer should actively anticipate and handle
    - E.g. when opening a file, it should be anticipated by the programmer that the file cannot be opened and hence `FileNotFoundException` should be explicitly handled
    - All checked exceptions should be caught (**`catch`**) or propagated (**`throw`**)
  - An **unchecked exception** is one that is unanticipated, usually the result of a bug
    - E.g. `ArithmeticException` surfaces when trying to divide by zero

Exception Hierarchy



- Unchecked exceptions are sub-classes of `RuntimeException`
- All `Errors` are also unchecked
- When overriding a method that throws a checked exception, the overriding method cannot throw a more general exception
- Avoid catching `Exception`, *aka Pokemon Exception Handling*
- Handle exceptions at the appropriate abstraction level, do not just throw and break the abstraction barrier

The `static` Keyword

- `static` can be used in the declaration of a field or method
- A `static` **field** is class-level member declared to be shared by all objects of the class
  - Use for defining constants, e.g. EPS

    ```
    private static final double EPS = 1e-15;
    ```

  - Use for defining *aggregated data*, e.g. number of circles

    ```
    class Circle {
        private final Point centre;
        private final double radius;
        private static final double EPS = 1e-15;
        private static int numOfCircles = 0; // mutable!

        Circle(Point centre, double radius) {
            this.centre = centre;
            this.radius = radius;
            Circle.numOfCircles = Circle.numOfCircles + 1;
        }
    ```

Other uses (*out of scope*): static blocks, static nested inner classes

The **static** Keyword

- **static** methods belong to the class instead of an object
  - methods that access/mutate static fields:

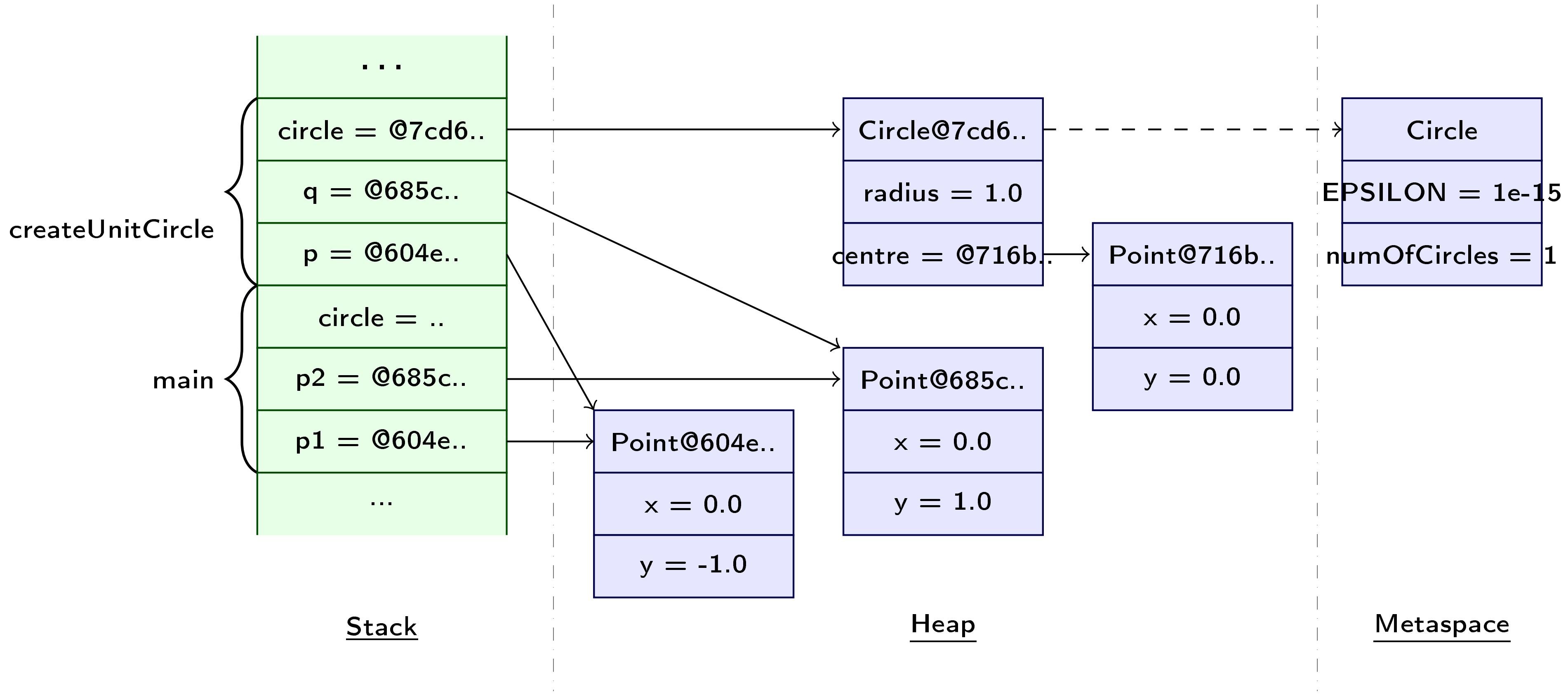
    ```
    static int getNumOfCircles() {
        return Circle.numOfCircles;
    }
    ```

  - factory method: `static Circle createUnitCircle(Point p, Point q) {`
  - `main` method: `public static void main(String[] args) {`
  - No overriding as **static** methods resolved at compile time
- **static** fields/methods *should* be called through the class

```
jshell> Circle c = new Circle(new Point(0.0, 0.0), 1.0)
c ==> Circle at (0.0, 0.0) with radius 1.0

jshell> Circle.getNumOfCircles()
$.. ==> 1

jshell> c.getNumOfCircles() // possible, but to be avoided
$.. ==> 1
```

Java Memory Model Revisited

- Other than the stack and heap, a non-heap (metaspace since Java 8) is used for storing loaded classes, and other meta data
  - **static** fields are stored here



Enumeration

- An **enum** is a special type of class used for defining constants

```
enum Color {
   BLACK, WHITE, RED, BLUE, GREEN, YELLOW, PURPLE
}
...
Color color = Color.BLUE;
```

- **enum** is type-safe; `color = 1` is invalid
- Can also define *constant-specific class body* — custom methods for each **enum**'s constant

```
enum Color {
    BLACK(0, 0, 0),
    WHITE(1, 1, 1),
    RED(1, 0, 0),
    BLUE(0, 0, 1),
    GREEN(0, 1, 0),
    YELLOW(1, 1, 0),
    PURPLE(1, 0, 1);

    private final double r;
    private final double g;
    private final double b;

    Color(double r, double g, double b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public double luminance() {
        return (0.2126 * r) + (0.7152 * g) + (0.0722 * b);
    }

    public String toString() {
        return "(" + r + ", " + g + ", " + b + ")";
    }
}
```

Preventing Inheritance and Overriding

- The **final** keyword can also 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 {
    ⋮
    @Override
    final double getArea() {
        ⋮
    }
    ⋮
    @Override
    final double getPerimeter() {
        ⋮
    }
}
```

Creating Packages

- Include the **`package`** statement at the top of all source files that reside within the package, e.g.

  ```
  package cs2030.test;
  ```

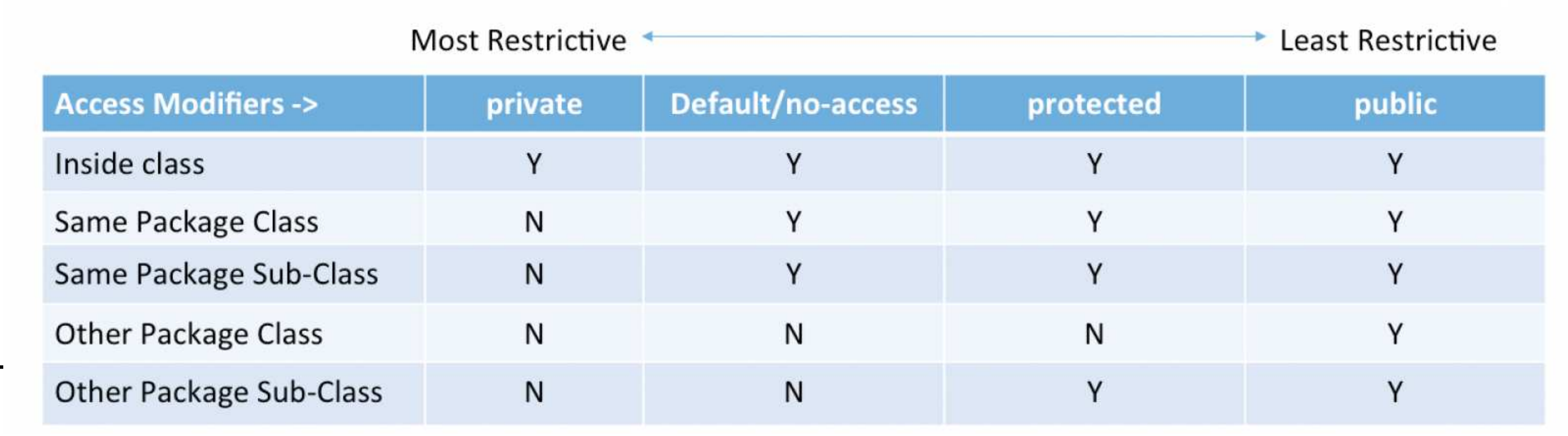
- Include the **`import`** statement to source files outside the package, e.g.

  ```
  import cs2030.test.SomeClass;
  ```

- Compile the Java files using

  ```
  $ javac -d . *.java
  ```

- `cs2030/test` directory created with same-package class files stored within



Most Restrictive ←→ Least Restrictive

Access Modifiers ->	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

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();
    }
}
```