CS2030 Lecture 6

Java static/enum/final, Exception Handling and Assertions

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

Semester 1 2021 / 2022

Outline and Learning Outcome

- Understand the use of **`static`**, **`enum`** and **`final`** keywords under different usage contexts
- Be able to employ exception handling to deal with "exceptional" events that are beyond our control such as user mistakes, network connection errors, external database storage errors, etc.
  - Understand the use of **`try`**–**`catch`**–**`finally`** clauses
  - Able to distinguish the different types of exceptions
  - Able to appreciate exception control flow
- Be able to define assertions as pre- and post-conditions in order to to deal with programmer errors

The static Keyword

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

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

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

The `static` Keyword

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

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

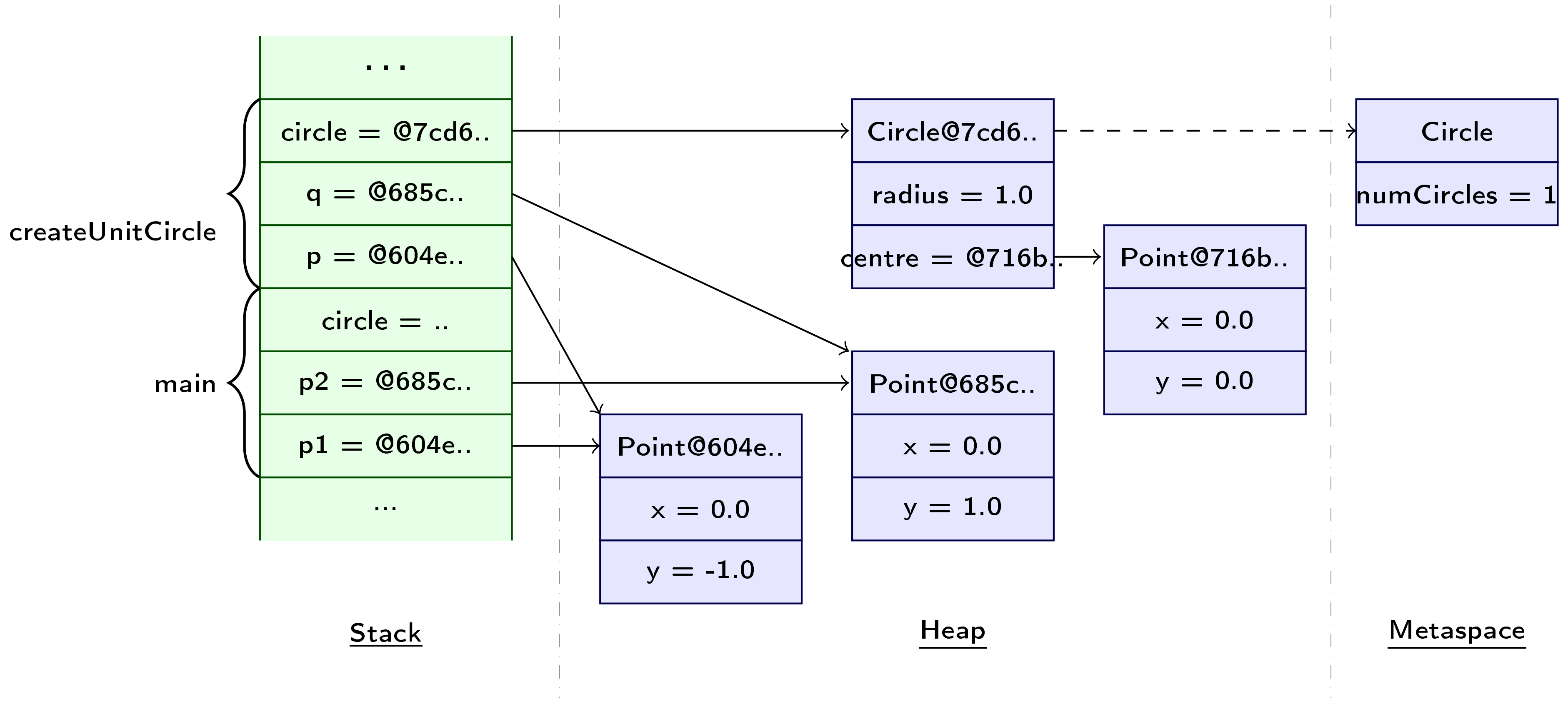
  - main method: `public static void main(String[] args) {`
  - factory method: `static Circle createUnitCircle(Point p, Point q) {`
  - No overriding as `static` methods resolved at compile time
- `static` fields/methods *should* be called through the class

  ```
  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
  ```

- Other uses: static blocks, static nested inner classes

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
- Each constant of an **enum** type is an instance of the **enum** class and is a field declared with **public static final**
- Constructors, methods, and fields can be defined in **enum**s

```
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() {
                    ⋮
          }
}
```

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);
    Point[] points = new Point[sc.nextInt()];
    for (int i = 0; i < points.length; i++) {
        points[i] = 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 double 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

```
public static void main(String[] args) throws FileNotFoundException {
```

- Method #2: **handle** the exception

```
try {
    FileReader file = new FileReader(args[0]);
    Scanner sc = new Scanner(file);
    Point[] points = new Point[sc.nextInt()];
    for (int i = 0; i < points.length; i++) {
        points[i] = 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

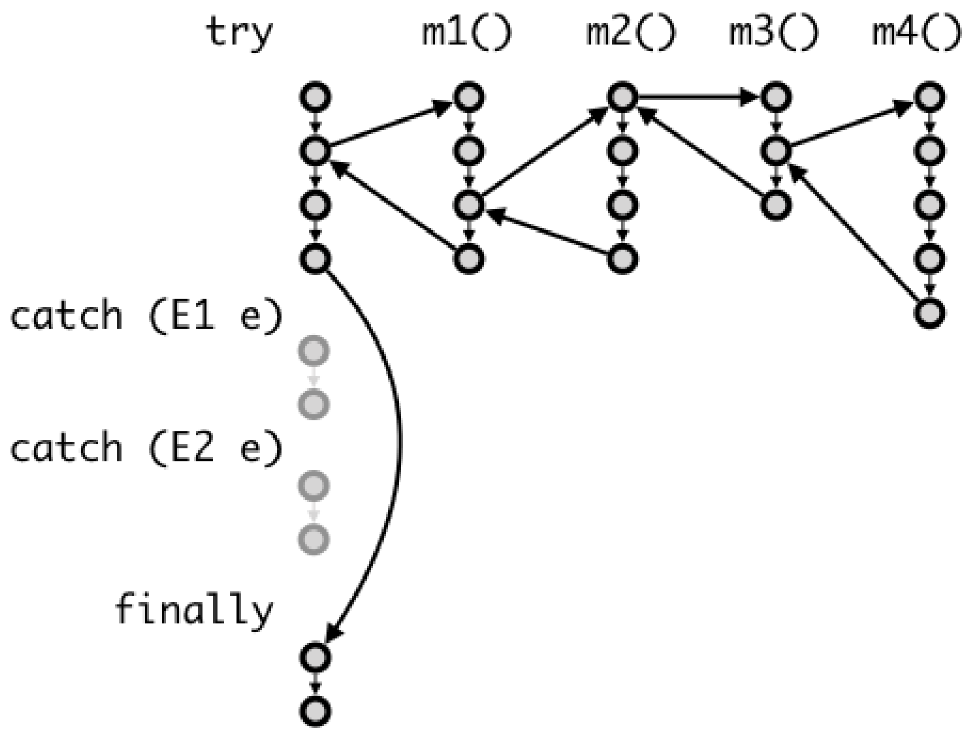
- Multiple catch blocks ordered by *most specific exceptions first*

```
try {
    FileReader file = new FileReader(args[0]);
    Scanner sc = new Scanner(file);
    Point[] points = new Point[sc.nextInt()];
    for (int i = 0; i < points.length; i++) {
        points[i] = 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");
}
```

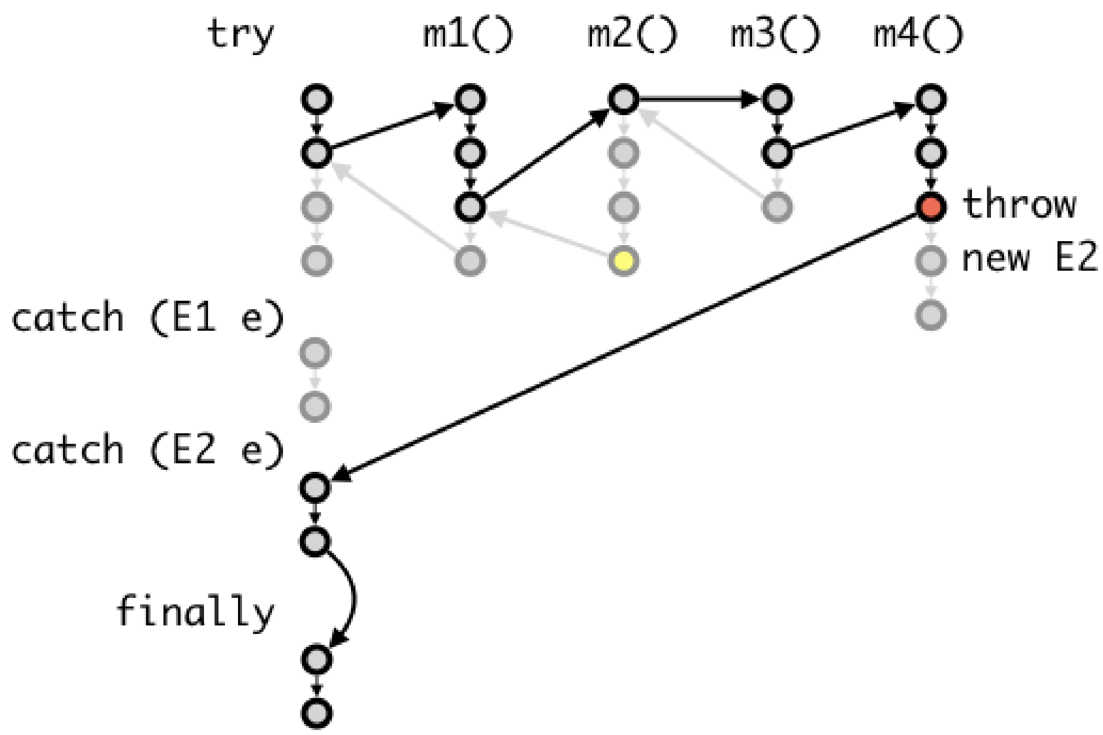
- Optional **`finally`** block used for house-keeping tasks
- Multiple exceptions (no sub-classing) in a single catch using |

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

Create and throw an Exception

- Consider the following `createUnitCircle` method

```
static Circle createUnitCircle(Point p, Point q) {
    double distPQ = p.distanceTo(q);
    if (distPQ > 0.0 && distPQ < 2.0 + EPSILON) {
        ...
    } else {
        throw new IllegalArgumentException("Distance pq not within (0, 2]");
    }
}
```

- User defined exception by inheriting from existing ones

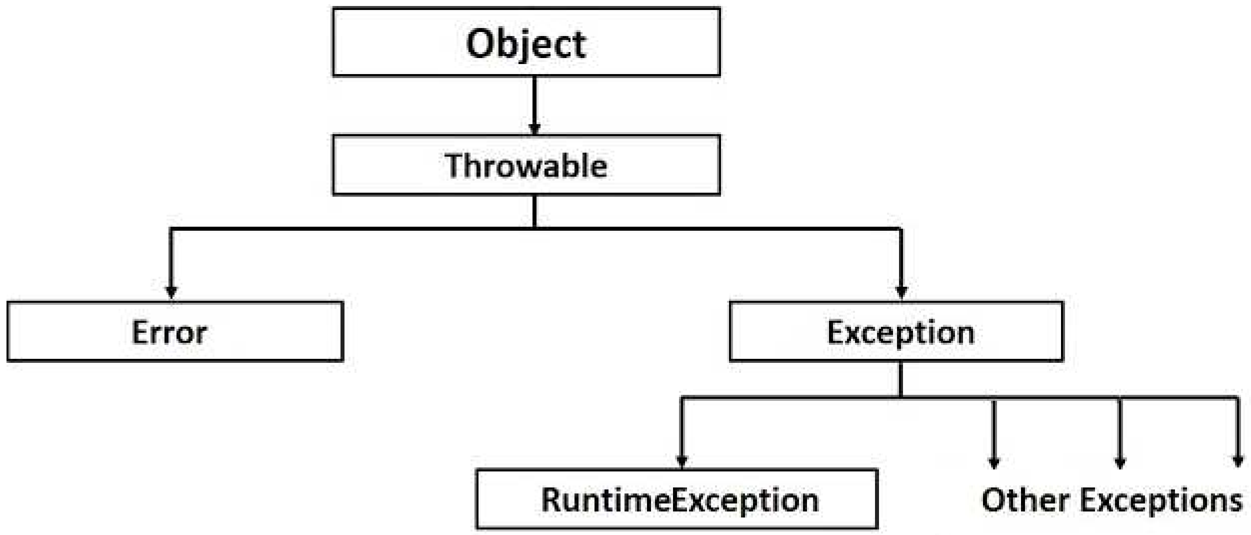
```
class IllegalCircleException extends IllegalArgumentException {
    IllegalCircleException(String message) {
        super(message);
    }

    @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

Assertions

- While exceptions are used to handle user mishaps, **assertions** are used to identify bugs during program development
- When implementing a program, it is useful to state conditions that should be true at a particular point, say in a method
- These conditions are called **assertions**; there are two types:
  - **Preconditions** are assertions about a program's state when a program is invoked
  - **Postconditions** are assertions about a program's state after a method finishes
- There are two forms of `assert` statement
  - `assert boolean_expression;`
  - `assert boolean_expression : string_expression;`

Assertions

- Suppose invocation of `createUnitCircle` is pre-conditioned on the distance of the two points to be within $(0, 2]$
  - Any violation of this precondition within the method is deemed as a bug!
- Define an assertion within `createUnitCircle` as follows:

```
static Circle createUnitCircle(Point p, Point q) {
    double distPQ = p.distanceTo(q);
    assert (distPQ > 0.0 && distPQ < 2.0 + EPSILON);
    ...
```

- The `-ea` flag tells the JVM to enable assertions
  - Using Jshell, e.g. `jshell -R -ea ...`
  - Running the program, e.g. `java -ea ...`
- For a more meaningful message, replace the assertion with

```
assert (distPQ > 0.0 && distPQ < 2.0 + EPSILON) : "Error with distPQ!";
```