CS2030 Programming Methodology II

Lecture IV

Shengdong Zhao
Spring 2019

Acknowledge: slides are adapted from Henry Chia

Lecture Outline

- **File input**
- **Exception handling**
  - Throwing exceptions
  - try–catch–finally
  - Handling multiple exceptions
  - Exception control flow
  - Checked and unchecked exceptions
  - Generating exceptions
- **Assertions**
  - Preconditions and postconditions
- **Java `enum` types**

File Input

- Suppose reading via file input: `$ java Main data.in`
- How can a user misuse our program?
  - User does not specify a file: `$ java Main`
  - User misspells the filename: `$ java Main in.data`
  - The file provided contains an odd number of double values

```
0.46958466 -0.929214594
-2.798873326 3.000093839
-0.427611837
```

- The file contains a non-numerical value

```
A -0.929214594
-2.798873326 3.000093839
-O.427611837 3.101891969
```

Injecting Error Handling Code

```c
if (argc < 2) {
    fprintf(stderr, "Missing filename\n", argc);
} else {
    filename = argv[1];
    fd = fopen(filename, "r");
    if (fd == NULL) {
        fprintf(stderr, "Unable to open file %s.\n", filename);
    } else {
        numOfPoints = 0;
        while ((errno = fscanf(fd, "%lf %lf", &point.x, &point.y)) == 2) {
            points[numOfPoints] = point;
        }
        if (errno != EOF) {
            fprintf(stderr, "File format error\n");
        }
        fclose(fd);
    }
}
```

- Where is the main “business logic” in the program fragment?

Separate Business Logic from Error Handling

- What we desire is to separate the “business logic” part from the error handling part of the code.

```
public static void main(String[] args) {
       FileReader file = new FileReader(args[0]);
       Scanner scanner = new Scanner(file);
       Point[] points = new Point[100];
       int numOfPoints = 0;
       while (scanner.hasNextDouble()) {
              double x = Double.parseDouble(scanner.next());
              double y = Double.parseDouble(scanner.next());
              points[numOfPoints] = new Point(x, y);
              numOfPoints++;
       }
       DiscCoverage maxCoverage = new DiscCoverage(points,
              numOfPoints);
       System.out.println(maxCoverage);
}
```

Throwing Exceptions

- Compiling the program 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]);
                      ^
1 error
```

- From the Java API Specifications for FileReader the following constructor is specified:

```
public FileReader(String fileName)
       throws FileNotFoundException
```

- This means that the `FileNotFoundException` must be handled (or thrown)

throws Exception Out of a Method

- One way is to just throw the exception out from the main method in order to make it compile

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

- When the file cannot be found, the exception will be thrown at the user of the program

```
$ javac Main.java
$ java Main in.data
Exception in thread "main" java.io.FileNotFoundException: in.data (No such file
or directory)
         at java.base/java.io.FileInputStream.open0(Native Method)
         at java.base/java.io.FileInputStream.open(FileInputStream.java:196)
         at java.base/java.io.FileInputStream.<init>(FileInputStream.java:139)
         at java.base/java.io.FileInputStream.<init>(FileInputStream.java:94)
         at java.base/java.io.FileReader.<init>(FileReader.java:58)
         at Main.main(Main1.java:12)
```

- The reserved word used here is **throws** and not to be confused with **throw** as discussed later

Handling Exceptions

- The more responsible way is to handle the exception:

```
try {
        FileReader file = new FileReader(args[0]);
        Scanner scanner = new Scanner(file);
        Point[] points = new Point[100];
        int numOfPoints = 0;
        while (scanner.hasNextDouble()) {
                double x = Double.parseDouble(scanner.next());
                double y = Double.parseDouble(scanner.next());
                points[numOfPoints] = new Point(x, y);
                numOfPoints++;
        }
        DiscCoverage maxCoverage = new DiscCoverage(points,
numOfPoints);
        System.out.println(maxCoverage);
}      catch (FileNotFoundException ex) {
        System.err.println("Unable to open file " + args[0] +
                "\n" + ex + "\n");
}
```

try and catch Blocks

- Notice that while error (exception) handling is performed, the business logic of the program does not change
- This is made possible because of separate **try** and **catch** blocks; specifically
  - The **try** block encompasses the business logic of the program
  - Exception handling is dealt with in separate **catch** blocks, typically one for each exception
  - In addition, there is an optional **finally** block which can be used for house-keeping tasks
- Exceptions provide us a way to keep track of the reason for program failure, without which we would then have to rely on error numbers stored in normal variables

Catching Multiple Exceptions

- Multiple catch blocks can be defined to handle individual exceptions
- More than one exception can be handled in a single catch block using |
- An exception (just like an object) can be printed, typically through `System.err.println`

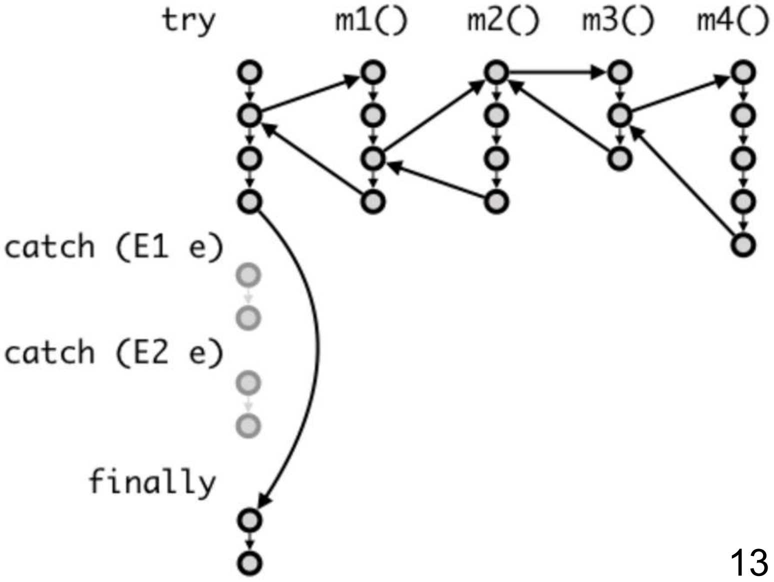
```
try {
        FileReader file = new FileReader(args[0]);
        Scanner scanner = new Scanner(file);
        Point[] points = new Point[100];
        int numOfPoints = 0;
        while (scanner.hasNext()) {
                double x = Double.parseDouble(scanner.next());
                double y = Double.parseDouble(scanner.next());
                points[numOfPoints] = new Point(x, y);
                numOfPoints++;
        }
        DiscCoverage maxCoverage = new DiscCoverage(points, numOfPoi
        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 (NumberFormatException | NoSuchElementException ex) {
        System.err.println("Incorrect file format\n");
} finally {
        System.err.println("Program Terminates\n");
}
```

do try -> if exception -> execute catch -> finally

exception order:specified -> generic

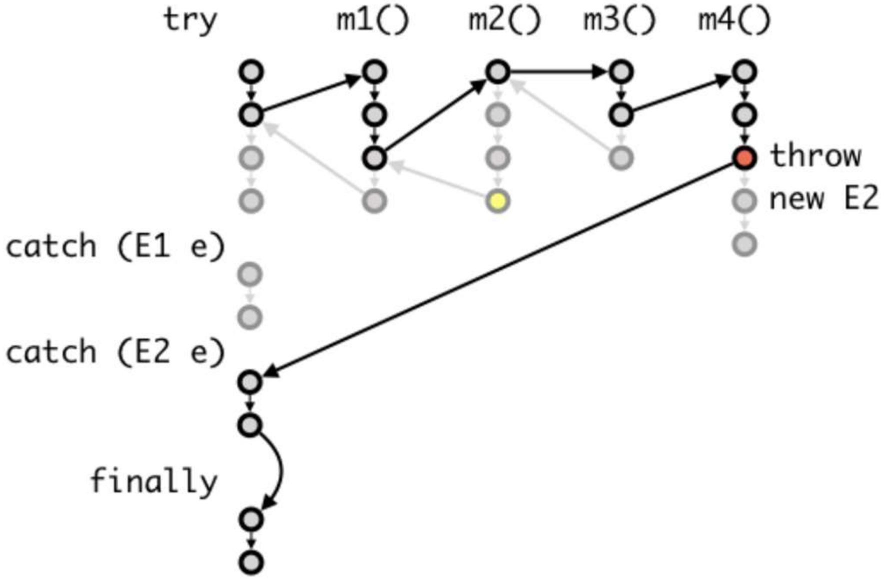
Exception Control Flow

- Consider a **try**–**catch**–**finally** block that catches two exceptions E1 and E2.
- Within the **try** block
  - method m1() is called;
  - m1() calls method m2();
  - m2() calls method m3(); and
  - m3() calls method m4().
- The control flow for the normal (i.e. no exception) situation, looks like this:



Exception Control Flow

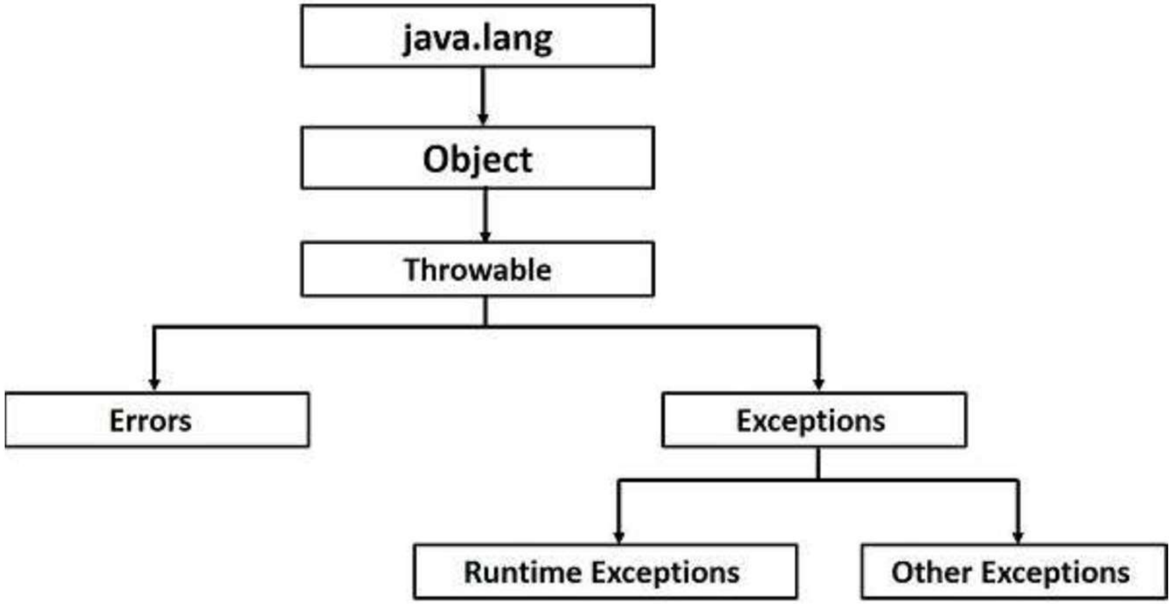
- Suppose an exception E2 is thrown in m4(), and causes the execution in m4 to stop prematurely
- The block of code that catches E2 is searched, beginning at m4(), then back to it’s caller m3(), then m2(), then m1()
- Notice that none of the methods m1() to m4() catches the exception; hence the code that handles E2 in the initial caller is executed before executing the **finally** block



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
  - An unchecked exception is one that is unanticipated, usually the result of a bug
    - E.g. a `NullPointerException` surfaces when trying to call p.distanceTo(q), with p being `null`
- All checked exceptions should be caught (**catch**) or propagated (**throw**)

Exception Hierarchy



- Unchecked exceptions are sub-classes of `RuntimeException`
- All `Errors` are also unchecked.

throw an Exception

- Given the constraints of a problem, if our program does not meet these constraints, then we have an exception scenario
- For example, given two points p and q, if their distance is more than 2, then they cannot form the boundary of a unit circle

```
public Circle(Point p, Point q, double radius) {
        if (p.distanceTo(q) > 2 * radius) {
                throw new IllegalArgumentException(
                        "Input points are too far apart");
        }
        if (p.equals(q)) {
                throw new IllegalArgumentException(
                        "Input points coincide");
}
        this.radius = radius;
        this.centre = findCentre(p, q, radius);
}
```

throw and Exception

- The exception can be thrown to the caller which in this case is ignored

```
for (int i = 0; i < points.length - 1; i++) {
        for (int j = i + 1; j < points.length; j++) {
                try {
                        Circle c = new Circle(points[i], points[j], 1.0);
                        int numOfPoints = findCoverage(c, points);
                        if (numOfPoints > maxDiscCoverage) {
                                maxDiscCoverage = numOfPoints;
                                this.maxCircle = c;
                        }
                } catch (IllegalArgumentException ex) {
                        // System.out.println(ex);
                }
        }
}
```

Generating Exception

- Create your own exception by inheriting from existing ones

```
class IllegalCircleException extends IllegalArgumentException {
        Point p;
        Point q;

        IllegalCircleException(String message) {
                super(message);
        }

        IllegalCircleException(Point p, Point q, String message) {
                super(message);
                this.p = p;
                this.q = q;
        }

        @Override
        public String toString() {
                return p + ", " + q + ": " + getMessage();
        }
}
```

Notes on Exceptions

- Only create your own exceptions if there is a good reason to do so, else just find one that suits your needs
- When overriding a method that throws a checked exception, the overriding method must throw only the same or more specific exception (why?)
- Although convenient, do not catch the “mother” Exception
- Handle exceptions at the appropriate abstraction level, do not just throw and break the abstraction barrier

```
public void m2() throws E2 { // Bad
        // setup resources
        m3();
        // clean up resources
}
```

```
public void m2() throws E2 { // Good
        try {
                // setup resources
                m3();
}
catch (E2 e) {
throw e;
        }
finally {
// clean up resources
}
}
```

Assertions

- While exceptions are usually used to handle user mishaps, assertions are used to prevent bugs
- 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:
  - **Pre-conditions** are assertions about a program's state when a program is invoked
  - **Post-conditions** are assertions about a program's state after a method finishes
- There are two forms of assert statement
  - assert expression;
  - assert expression1 : expression2;

Assertions

```
public double distanceTo(Point q) {
        double distance = Math.sqrt(Math.pow(dx(q),2) +
                Math.pow(dy(q),2));
        assert distance >= 0;
        return distance;
}
```

- Run the program with

```
$ java -ea Main data.in
Program Terminates

Exception in thread "main" java.lang.AssertionError
        at Point.distanceTo(Point.java:21)
        at Main.findMaxDiscCoverage(Main.java:38)
        at Main.main(Main.java:67)
```

- The `-ea` flag tells the JVM to enable assertions

Assertions

- For a more meaningful message, replace the assertion with

```
assert distance >= 0 :
        this.toString() + " " + q.toString() + " = " + distance;
```

- Run the program

```
$ java -ea Main data.in
Program Terminates

Exception in thread "main" java.lang.AssertionError: (0.470, -0.929)
(-2.799, 3.000) = -5.110996220688496
        at Point.distanceTo(Point.java:22)
        at Main.findMaxDiscCoverage(Main.java:38)
        at Main.main(Main.java:67)
```

- Notice the **finally** block still executes since assertions are just normal exceptions

Enumeration

- An **enum** is a special type of class used for defining constants
- To define an **enum**,

```
enum EventType {
        ARRIVE,
        SERVE,
        WAIT,
        LEAVE,
        DONE
}
```

- Declare say, `eventType` with type `EventType` instead of **int**
- **enum** are type-safe since `eventType = 1` no longer works, but `eventType = EventType.ARRIVE` does

Enum’s Fields and Methods

- 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 **enums**

```
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 + ")";
    }
}
```

Lecture Summary

- Exceptions are meant to deal with "exceptional" events beyond our control such as user mistakes, network connection errors, external database storage errors, etc.
  - These need to be handled elegantly
- Assertions, on the other hand, are meant to deal with programmer errors
  - Use them liberally to provide an assurance that conditions at certain points of the program are met
  - Letting the program "crash" when a condition is not met is still better than carrying on executing with the error