CS2030 Lecture 6

Other Java Constructs

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

Semester 2 2019 / 2020

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 *aggregated data*, e.g. number of circles

```
public class Circle {
    private double radius;
    private static int numOfCircles = 0;

    public Circle(double radius) {
        this.radius = radius;
        Circle.numOfCircles++; // or this.numCircles++
    }
```

  - Use for constants, **`static double final`** `PI = 3.146;`

properties of the class is object-level member -> properties is shared within the object
static field is class-level member -> shared within the whole class

The `static` Keyword

- **`static`** methods belong to the class instead of an object
  - For factory methods **`static`** `Circle createCircle(...)`
  - For methods that access/mutate static fields

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

  - No overriding as **`static`** methods resolved at compile time
- **`static`** block to initialize static fields that can't be done via `=`

```
class MyColors {
    static List<Color> colors = new ArrayList<>();
    static {
        colors.add(Color.BLUE);
        ...
    }
}
```

The `static` Keyword

- Nested classes: classes that are defined within another class

đóng gói
  - Encapsulation: nested class only useful in its enclosing class
  - Non-static (anonymous) inner classes
    - can access all (even private) members of enclosing class
  - `static` nested classes
    - can only access static members of enclosing class
    - top-level class cannot be made static

```
class Circle {
    private double radius;
    public Circle() {
        new CircleCreator().create(this);
    }
    private static class CircleCreator {
        private void create(Circle circle) {
            ...
        }
    }
```

Java Memory Model Revisited

- The Java **memory model** comprising three areas:
  - Stack — handles method calls
    - LIFO stack for storing activation records of method calls
    - method local variables are stored here
  - Heap
    - for storing Java objects upon invoking **`new`**
    - *garbage collection* is done here
  - Non-heap (*Metaspace* since Java 8) — class information
    - for storing loaded classes, and other meta data
    - **`static`** fields are stored here

Error Handling Code

String[] args: a string of file names
To read file:
import java.io.FileReader
FileReader file = new FileReader(args[0])
Scanner sc = new Scanner(file)

- Suppose reading via file input: `$ java Main data.in`
  - 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
  - The file contains a non-numerical value

```
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);
    }
}
```

Throwing Exceptions

- Use exceptions to track reasons for program failure, rather than to rely on error numbers stored in variables

```
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.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, numOfPoints);
    System.out.println(maxCoverage);
}
```

- 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]);
                          ^
```

`throws` Exception Out of a Method

throws new ExceptionType('Error Detailed message')

- 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
- The more responsible way is to handle the exception

Handling Exceptions

- Notice that while error (exception) handling is performed, the business logic of the program does not change
  - **try** block encompasses the business logic
  - **catch** block handles exceptions

```
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, numOfPoints);
    System.out.println(maxCoverage);
} catch (FileNotFoundException ex) {
    System.err.println("Unable to open file " + args[0] +
            "\n" + ex + "\n");
}
```

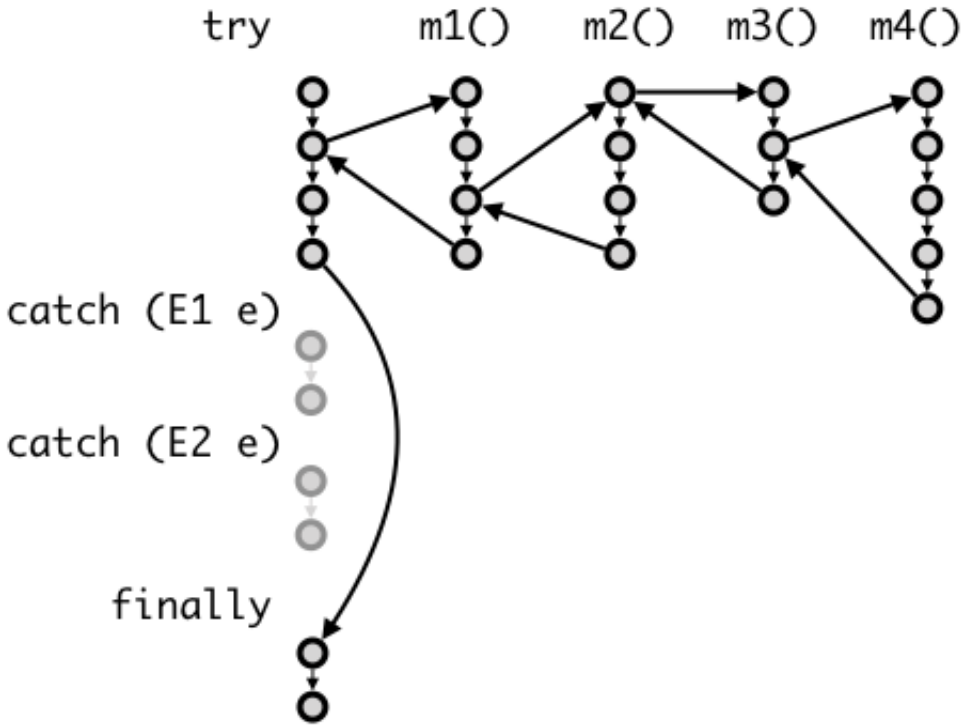
Catching Multiple Exceptions

- Multiple catch blocks defined to handle each exception
- Handling multiple exceptions in a single catch using |
- Optional **finally** block used for house-keeping tasks

```
try {
    FileReader file = new FileReader(args[0]);
    Scanner scanner = new Scanner(file);
    Point[] points = new Point[100];
    int numOfPoints = 0;
    while (scanner.hasNext()) {
        points[numOfPoints] = new Point(
                Double.parseDouble(scanner.next()),
                Double.parseDouble(scanner.next()));
        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);
} 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");
}
```

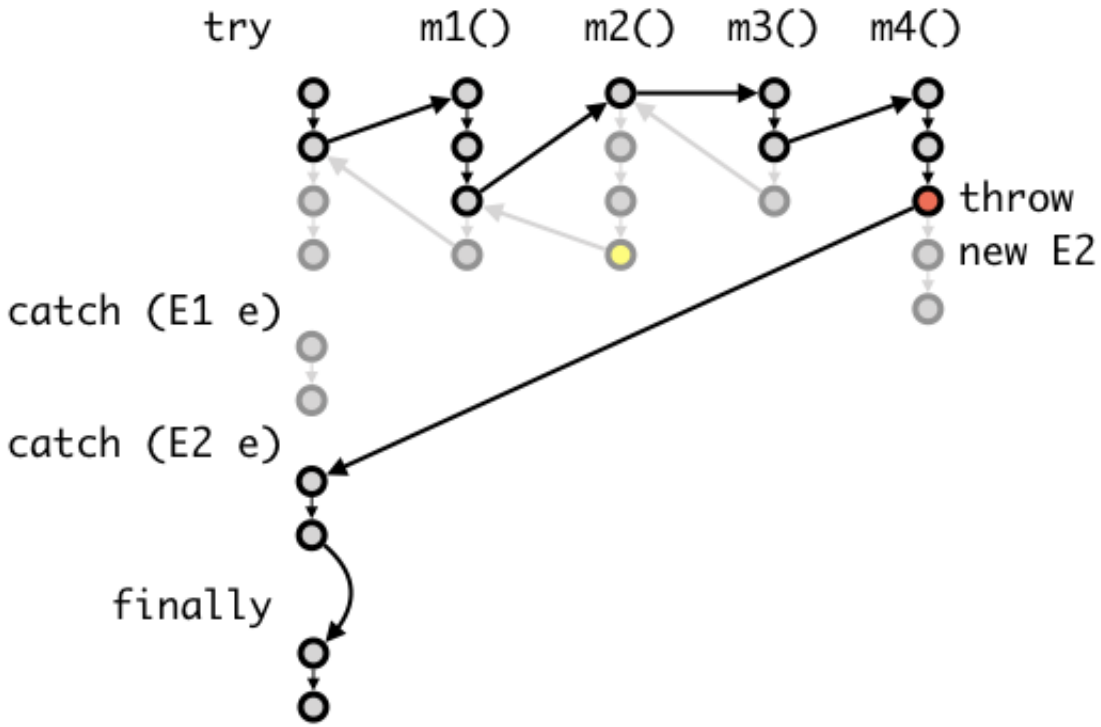
Exception Control Flow

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



Exception Control Flow

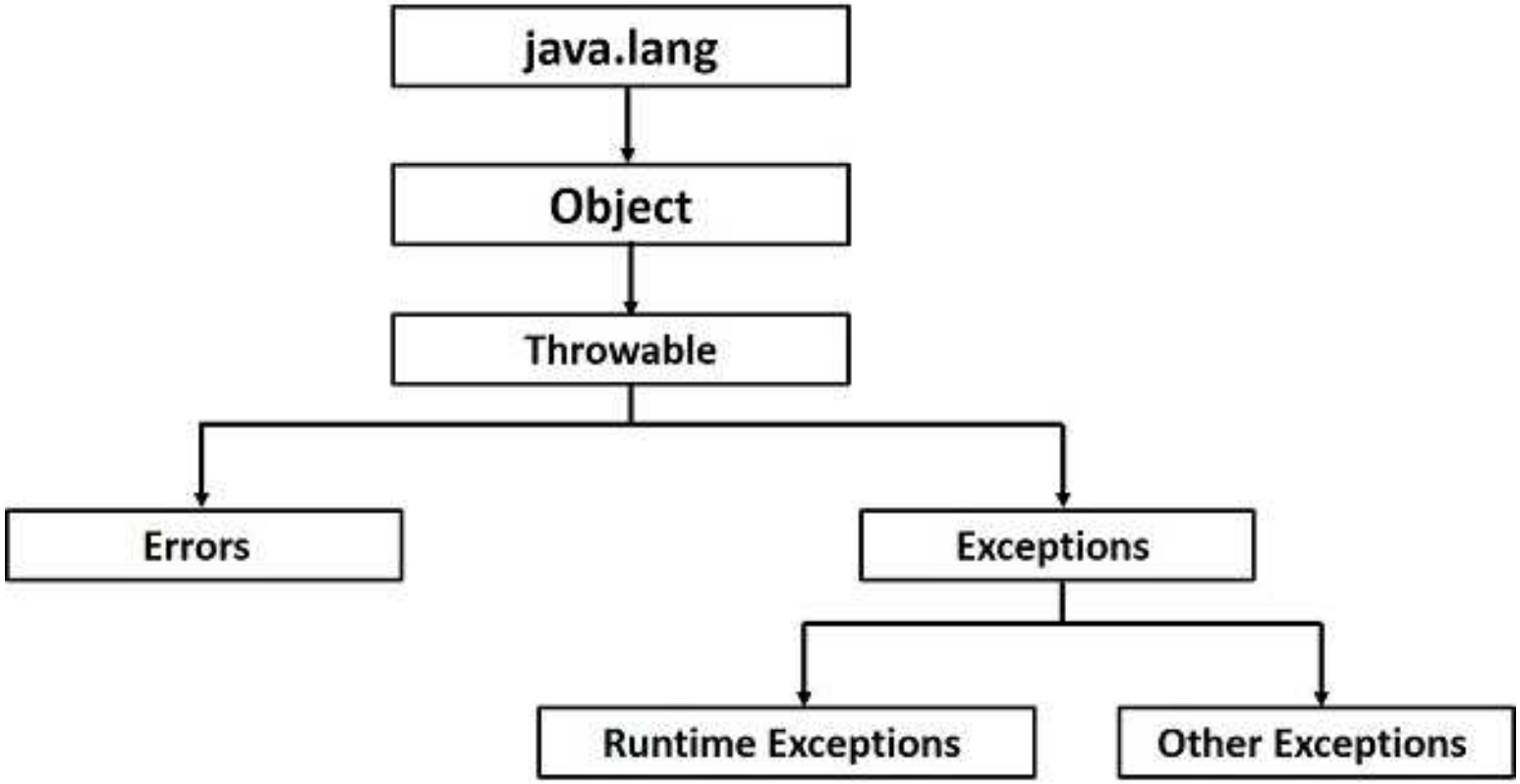
- Suppose an exception `E2` is thrown in `m4()`, and causes the execution in `m4` to stop prematurely
- The block of code that catches `E2` is searched, beginning at `m4()`, then back to it's caller `m3()`, then `m2()`, then `m1()`
- Notice that none of the methods `m1()` to `m4()` catches the exception; 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 two points p and q, if their distance is more than twice the radius of the circle, then the circle cannot be formed

```
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);
}
```

```
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

- User defined 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?)
- Avoid catching `Exception`, *aka Pokemon Exception Handling*
- Handle exceptions at the appropriate abstraction level, do not just throw and break the abstraction barrier

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

bad because there are time where you cannot clean up resources

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

good because you can 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:
  - **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 expression;`
  - `assert expression1 : expression2;`

Assertions

- Consider the following program fragment

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

```
$ 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
- For a more meaningful message, replace the assertion with

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

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** are 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 + ")";
    }
}
```

Enum's Fields and Methods

- All **enum**s inherit from the class `Enum<E>` implicitly
- Two useful implicitly declared static methods are:
  - `public static E[] values();`

```
for (Color color : Color.values()) {
    System.out.println(color.luminance());
}
```

  - `public static E valueOf(String name);`

```
Scanner sc = new Scanner(System.in);
while (sc.hasNext()) {
    System.out.println(Color.valueOf(sc.next()));
}
```

Access Modifiers

- In the discussion of an abstraction barrier, we have seen the use of the **`public`**, **`private`** and **`protected`** modifiers
- Other than these three, there is a default modifier
- Java adopts an additional **`package`** abstraction mechanism that allows the grouping of relevant classes/interfaces together under a *namespace*, just like `java.lang`
- In particular, a **`protected`** field can be accessed by other classes within the same package
- The access level (most restrictive first) is given as follows:
  - private (visible to the class only)
  - default (visible to the package)
  - protected (visible to the package and all sub classes)
  - public (visible to the world)

Access Modifiers

to declare a package, at the top of the file, write
package package_directory

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

https://www.comp.nus.edu.sg/~cs2030/javadoc/

Creating Packages

- Suppose `Shape`, `Scalable`, `Circle` and `Rectangle` resides in the `cs2030.shapes` package
- Include the following line at the top of the java files

```
package cs2030.shapes;
```

- Compile the four Java files using `javac -d . *.java`
- `cs2030/shapes` directory created with class files stored within
- The client, say `Main.java`, imports `cs2030.shapes` package

```
import cs2030.shapes.Shape;
import cs2030.shapes.Scalable;
import cs2030.shapes.Circle;
import cs2030.shapes.Rectangle;

class Main {
    ...
```

- Javadoc: e.g. invoke `javadoc Circle.java`