CS2030/S Lecture 5

Interlude...

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

Semester 1 2020 / 2021

Lecture Outline

- Java **`static`** keyword
- Java **`enum`** types
- Java packages and access modifiers
- Exception handling
  - Throwing exceptions
  - **`try`**–**`catch`**–**`finally`**
  - Exception control flow
  - Types of exceptions
- From Java's **`null`** value to Java's `Optional` class
  - A glimpse into the *very near* future...

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

```
class Circle {
    private final Point centre;
    private final double radius;
    private static int numOfCircles = 0; // mutable!

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

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

The `static` Keyword

- **`static`** methods belong to the class instead of an object
  - For factory methods

```
static Circle getCircle(Point centre, double radius) {
    return new Circle(centre, radius);
}
```

  - For methods that access/mutate static fields

```
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);
        ...
    }
}
```

Nested Classes

- Encapsulation: nested class only useful in its enclosing class
- Non-static nested inner classes
  - can access all (even private) members of enclosing class

```
class Circle {
    private double radius;
    ...
    Circle() {
        new UnitCircle().create(this);
    }

    private class UnitCircle {
        private void create(Circle circle) {
            circle.radius = 1.0;
        }
    }
```

- **static** nested inner classes
  - can only access static members of enclosing class
  - top-level class cannot be made static

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 + ")";
    }
}
```

Enum's Fields and Methods

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

```
jshell> for (Color color : Color.values()) {
   ...> System.out.println(color.luminance());
   ...> }
0.0
1.0
0.2126
0.0722
0.7152
0.9278
0.2848

jshell> Color.valueOf("BLUE")
$.. ==> (0.0, 0.0, 1.0)
```

Access Modifiers

- When discussing the abstraction barrier, we have been using **`private`** and default modifiers
- Other than these, there are **`protected`** and **`public`** modifiers
- Java adopts a **`package`** abstraction mechanism that allows the grouping of relevant classes/interfaces together under a *namespace*, just like `java.lang`
- 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)

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.Base;
  ```

- Compile the four Java files using

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

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

```
==> 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();
    }
}
```

Java Memory Model Revisited

- The Java **memory model** comprising three areas:
  - Stack
    - 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)
    - for storing loaded classes, and other meta data
    - **`static`** fields are stored here

Error Handling Code

- 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 contains a non-numerical value
  - The file provided contains insufficient double values

```c
// example code fragment in 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);
    }
}
```

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 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);
}
```

- 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

- 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`** (not to be confused with **`throw`** to be 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 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 + "\n");
}
```

Catching Multiple Exceptions

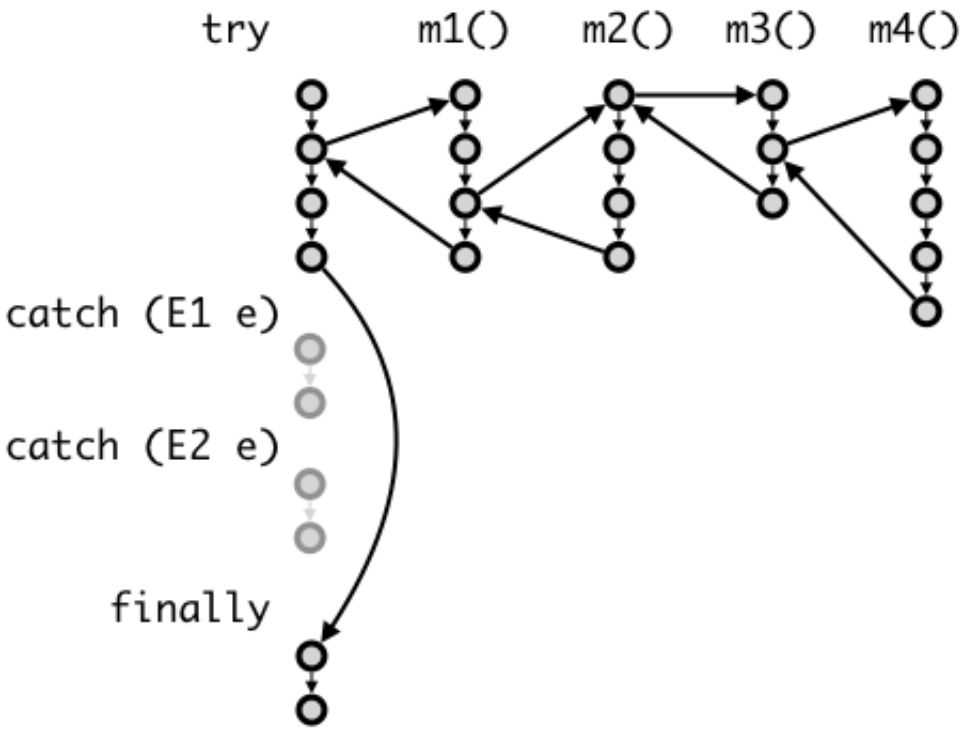
- Multiple catch blocks ordered by most specific exceptions first

```
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");
}
```

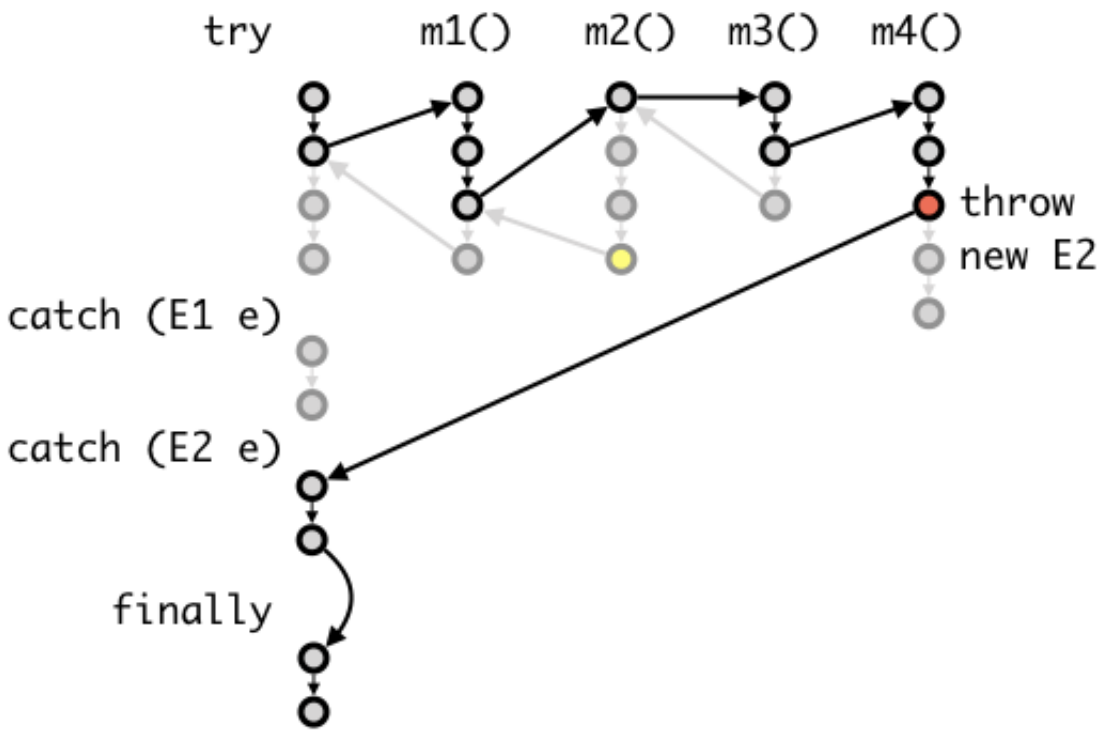
- Optional **`finally`** block used for house-keeping tasks
- Multiple exceptions (no sub-classing) in a single catch using `|`

Normal vs Exception Control Flow

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



Normal Control Flow

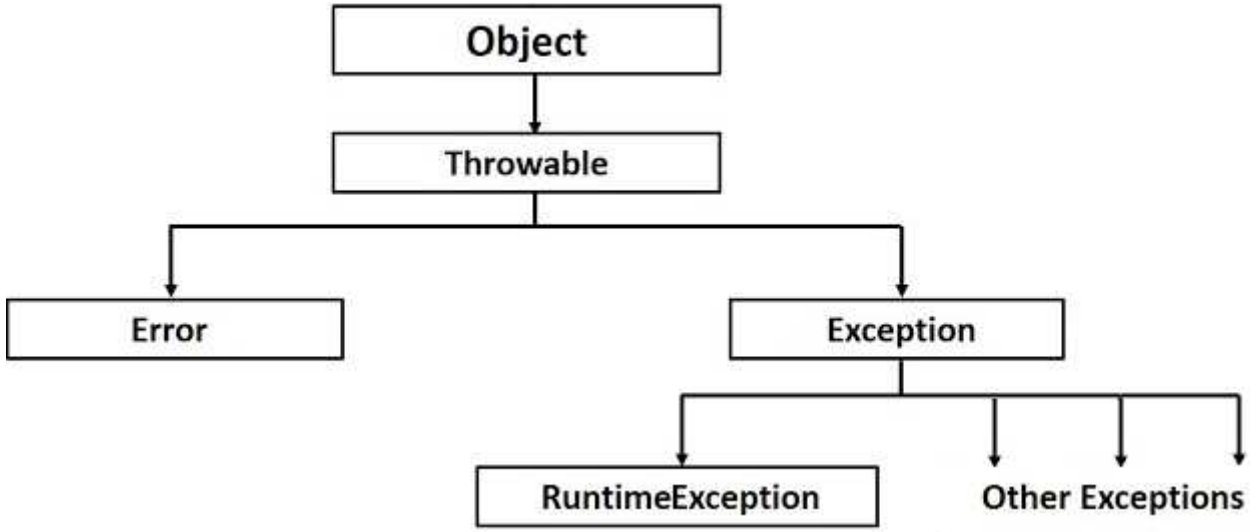


Exception Control Flow

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

`throw` an Exception

- Consider the following `getCircle` method

```
static Circle getCircle(Point centre, double radius) {
    if (radius > 0) {
        return new Circle(centre, radius);
    } else {
        throw new IllegalArgumentException("radius must be positive");
    }
}
```

- 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

`null` — The Billion-Dollar Mistake

```
static Circle getCircle(Point centre, double radius) {
    if (radius > 0) {
        return new Circle(centre, radius)
    } else {
        return null;
    }
}
```

- What happens to the following test?

```
Circle.getCircle(new Point(0, 0), -1).contains(new Point(0, 0))
```

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement."

– Sir Charles Antony Richard Hoare *aka Tony Hoare*

Java's `Optional<T>` Class

- **`static`** methods that create `Optional` objects

  `Optional.of(T value)`, `Optional.empty()`, `Optional.ofNullable(T value)`

- Some useful methods of the `Optional` class:

  ```
  public void ifPresent(Consumer<? super T> action)
  ```

  - `Consumer<T>` with abstract method `accept(T t)`

  ```
  public Optional<T> filter(Predicate<? super T> predicate)
  ```

  - `Predicate<T>` with abstract method `boolean test(T t)`

  ```
  public <U> Optional<U> map(Function<? super T, ? extends U> mapper)
  ```

  - `Function<T,R>` with abstract method `R apply(T t)`

  ...

Redefining the `getCircle` Method

- Creating a circle via `getCircle` may return a circle or nothing

```
import java.util.Optional;
...
    static Optional<Circle> getCircle(Point centre, double radius) {
        if (radius > 0)
            return Optional.of(new Circle(centre, radius));
        else
            return Optional.empty();
    }
```

```
jshell> Circle.getCircle(new Point(0, 0), 1)
$.. ==> Optional[Circle at (0.0, 0.0) with radius 1.0]

jshell> Circle.getCircle(new Point(0, 0), -1)
$.. ==> Optional.empty
```

- Chaining with `contains` method still gives a compilation error:

```
jshell> Circle.getCircle(new Point(0, 0), 1).contains(new Point(0, 0))
|  Error:
|  cannot find symbol
|    symbol:   method contains(Point)
|  Circle.getCircle(new Point(0, 0), 1).contains(new Point(0, 0))
|  ^-------------------------------------^
```

A Glimpse into the Future..

- `Consumer<T>` specifies an abstract method **`void`** `accept(T)`
- Just like `Comparator<T>`, we can define a class that implements the interface, or use an *anonymous inner class*

```
jshell> class MyConsumer implements Consumer<Circle> {
   ...> @Override
   ...> public void accept(Circle c) {
   ...> System.out.println(c.contains(new Point(0, 0)));
   ...> }
   ...> }
|  created class MyConsumer

jshell> MyConsumer consumer = new MyConsumer()
consumer ==> MyConsumer@4c70fda8

jshell> Circle.getCircle(new Point(0, 0), 1).ifPresent(consumer)
true

jshell> Consumer<Circle> consumer = new Consumer<Circle>() { // anonymous inner class
   ...> public void accept(Circle c) {
   ...> System.out.println(c.contains(new Point(0, 0)));
   ...> }}
consumer ==> 1@14acaea5

jshell> Circle.getCircle(new Point(0, 0), 1).ifPresent(consumer)
true

jshell> Circle.getCircle(new Point(0, 0), -1).ifPresent(consumer)

jshell>
```

A Glimpse into the Future..

- `Predicate<T>` specifies an abstract method **`boolean`** `test(T)`

```
jshell> Predicate<Circle> pred = new Predicate<Circle>() {
   ...> public boolean test(Circle c) {
   ...> return c.contains(new Point(0, 0)); }}
pred ==> 1@4c70fda8

jshell> Circle.getCircle(new Point(0, 0), 1).filter(pred)
$.. ==> Optional[Circle at (0.0, 0.0) with radius 1.0]

jshell> Circle.getCircle(new Point(0, 0), -1).filter(pred)
$.. ==> Optional.empty
```

- Compute the area only if a valid circle contains the origin
  - `Function<T,R>` specifies an abstract method `R apply(T)`

```
jshell> Function<Circle,Double> f = new Function<Circle,Double>() {
   ...> public Double apply(Circle c) {
   ...> return c.getArea(); }}
f ==> 1@26be92ad

jshell> Circle.getCircle(new Point(0, 0), 1).filter(pred).map(f)
$.. ==> Optional[3.141592653589793]

jshell> Circle.getCircle(new Point(0, 0), -1).filter(pred).map(f)
$.. ==> Optional.empty
```

to infinity... and beyond...