COMP8710 Advanced Java for Programmers

Lecture 6
Enums and Exceptions

Yang He

Topics

- The keywords static and final
- Enums
- Exceptions
- Set Assessment 1

The keywords static and final

Static variables and methods (1)

- We use the keyword static to declare a class member as static
- E.g.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Minimum age: " +
                           Member.MIN_AGE);
        var s1 = new Member("Tom");
        var s2 = new Member("Alice");
        System.out.println(s1);   // #1: Tom
        System.out.println(s2);   // #2: Alice
    }
}
```

```
public class Member {
    public static final int MIN_AGE = 18;
    private static int total = 0;
    private String id;
    private String name;
    public Student(String name) {
        this.id = createId();
        this.name = name;
    }
    private String createId() {
        total++;
        return "#" + total;
    }
    public String toString() {
        return id + ": " + name;
    }
}
```

Static variables and methods (2)

- Static variables and methods are associated with the class itself, rather than any specific instance of the class
  - They are allocated memory space only once during the execution of the program
  - They are shared across *all* instances of the class
  - They can be accessed without the need to create an instance of the class
    - E.g. `Member.MIN_AGE`
- Static methods
  - Can be overloaded, but not overridden
  - Cannot access *non-static* members of the class

They typically require little or no state information, i.e. statics are "this-less"

The final keyword

- The final keyword is a modifier applicable to a
  - Variable
  - Method
  - Class
- It helps improve
  - Performance
    - As the compiler can optimize the code more effectively when it knows that a variable or method cannot be changed
  - Security
    - By preventing malicious code from modifying sensitive data or behaviour

Final variables

- We can create constants with the final modifier
- It ensures that the value of a constant cannot be changed or reassigned after it is initialized
- E.g.

```
public class CD extends Item {
    private static final int MAX_PLAYING_TIME = 80;
    private final String artist;

    public CD(String artist, ...) {
        super(...);
        this.artist = artist;
        ...
    }
    ...
}
```

Final variables must be initialized either on declaration or in the class constructor.

Final methods

- We can declare a method as final
- Final methods cannot be overridden by subclasses
- This is to ensure that the implementation of a final method should be the same for all its subclasses

```
public class Game {
    public enum Player {USER, COMPUTER};
    ...
    public final Player getFirstPlayer() {
        return Player.USER;
    }
    ...
}
```

In general, methods called from constructors should be declared as final.

Final classes

- We can declare a class as final
- A final class cannot be extended by a subclass
- This is useful for classes that are intended to be used as defined and should not be modified or extended
- E.g. an immutable class like the String class

The enum type

The enum type (1)

- Java enum is a special "class" that represents a group of predefined constants
- E.g. There are four seasons in a year

```
public enum Season {
   SPRING, SUMMER, AUTUMN, WINTER;
}
```

Enum constants are separated by commas and should be in UPPERCASE letters

- Each name represents an object of the enumerated type
- We don’t need to instantiate an enum using the keyword new
- E.g.

```
Season currentSeason = Season.AUTUMN;
```

The enum type (2)

- To compare enum values, use the == operator
- e.g.

```
if (currentSeason == Season.AUTUMN) {
    System.out.println("It is Autumn.");
}
```

The enum type (3)

- Java enums implicitly extend the java.lang.Enum class
- They have
  - The same capabilities as a class
  - A set of predefined methods, e.g. `name(), values()`
- We use enum when we know all possible values at compile time
  - Make code more readable
  - Allow for compile-time checking

The enum type (4)

- We can assign values to enums by defining a constructor and a private field variable
- We can add methods in enums
- To iterate all enum values:

```
for (var season : Season.values()) {
    var info = season.name() + " in quarter "
             + season.getQuarter();
    System.out.println(info);
}
```

Outputs:

```
SPRING in quarter 1
SUMMER in quarter 2
AUTUMN in quarter 3
WINTER in quarter 4
```

```
public enum Season {
    SPRING(1),
    SUMMER(2),
    AUTUMN(3),
    WINTER(4);

    private final int quarter;
    Season(int quarter) {
        this.quarter = quarter;
    }

    public int getQuarter() {
        return quarter;
    }
}
```

The enum type (5)

- We can use enum types in a switch statement
- E.g.

```
String getWeather(Season season) {
    switch (season) {
        case SPRING:
            return "It's breezy!";
        case SUMMER:
            return "It’s sunny!";
        case AUTUMN:
            return "It’s rainy!";
        case WINTER:
            return "It’s chilly!";
    }
    return "None";
}
```

Homework:
Improve the zuul project by implementing an enum named CommandWord with 4 values: GO("go"), QUIT("quit"), HELP("help"), and UNKNOWN("?")

Exceptions

What do you do if there’s an error?

- Abort?
- Print an error message?
- Ignore it and carry on?
- Return a “special” value?

The need for handling exceptions

- Faced with an unreasonable request, how should an object react?
  - Print an error message. . .
  - Terminate. . .
  - Silently ignore the request. . .
  - Request a correction. . .
- Java's exception handling attempts to resolve this dilemma
  - It is one of the powerful mechanism to handle the runtime errors, so that the normal flow of the application can be maintained

What are an exceptions?

- An exception is an event that disrupts the normal flow of the program
- It is thrown at runtime by an object
- Exception examples:
  - `ArithmeticException` for integer division
  - `ArrayIndexOutOfBoundsException` for array access
  - `IndexOutOfBoundsException` for collections
  - `NumberFormatException` for parsing

Throwing and catching exceptions (1)

- E.g.

```
try {
    var num1 = getNumber();
    var num2 = getNumber();
    System.out.println("Division = " + num1 / num2);
} catch (ArithmeticException e) {
    System.out.println("Error: Divide by zero.");
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

Throwing and catching exceptions (2)

- If an exception is not caught, the program signals an error and exits with the stack trace (most recent method first):

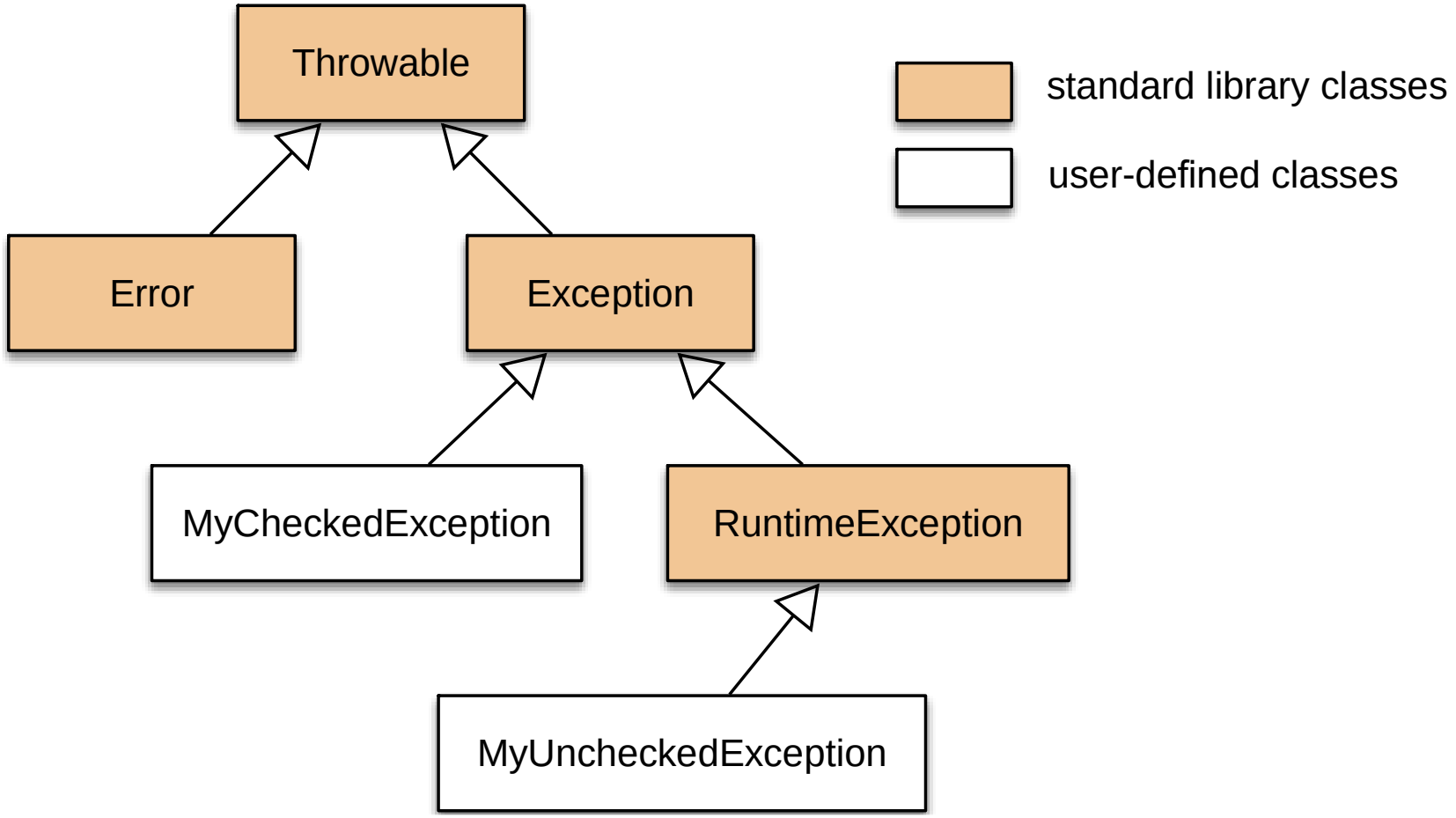
```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at Main.main(Main.java:10)
```

- Exceptions are objects, which means that throwing an exception involves creating an object

Principles of exceptions

- Use exceptions rather than special return values
  - Constructors can only throw exceptions as they have no return value
- Only use exceptions and exception handlers for exceptional behaviour
  - Don’t use exceptions to handle normal behaviour, like reaching the end of an array
- Handle errors as locally as possible
  - Listen for errors while they occur
  - Encourage recovery
  - Do not worry about what can go wrong at each line

The exception class hierarchy



Checked exceptions (1)

- Checked exceptions
  - Subclass of Exception
  - Use for anticipated failures
    - Problems which are outside the immediate control of the program, e.g. network issue
  - Where recovery may be possible
  - Checked at compile time

Checked exceptions (2)

- Checked exceptions are meant to be caught and responded to
- The compiler ensures that their use is tightly controlled
- We must either catch the exception or declare that our method may throw the exception
- E.g.

```
private static void checkedExceptionWithThrows() throws FileNotFoundException  {
   File file = new File("fake_file");
   FileInputStream stream = new FileInputStream(file);
}
```

Unchecked exceptions

- Unchecked exceptions
  - Subclass of RuntimeException
  - Use for unanticipated failures
    - Programming error that should be fixed, e.g. dividing by 0
  - Where recovery is unlikely
  - Not checked at compile time
- We do not need a `throws` declaration for these exceptions
- If not handled, unchecked exceptions will propagate up to the calling methods

User-defined exceptions

- E.g. in Cat class constructor, throw a checked exception

```
public Cat(String name, int age) throws InValidAgeException {

     if (age <= 0 || age > 25)
         throw new InValidAgeException("Invalid age");

     this.name = name;
     this.age = age;
}

public class InValidAgeException extends Exception {
     public InValidAgeException(String msg) {
        super(msg);
     }
}
```

Checked or unchecked exception? (1)

- From the Java documentation:
  - If a client can reasonably be expected to recover from an exception, make it a checked exception
  - If a client cannot do anything to recover from the exception, make it an unchecked exception

Checked or unchecked exception? (2)

- If we have some code that validates file names, and a user inputs an invalid name, we may throw a custom checked exception

```
if (!isCorrectFileName(fileName)) {
    throw new IncorrectFileNameException(fileName  );
}
```

- On the other hand, if the input file name string is empty or null, we have some internal logic errors and should throw an unchecked exception

```
if (fileName == null || fileName.isEmpty()) {
   throw new NullOrEmptyException("The filename is null or empty.");
}
```

Attempting recovery

- E.g. restrict the number of attempts

```
static int getNumber() {
    final int MAX_ATTEMPTS = 3;
    var successful = false;
    var number = 0;
    var attempts = 0;
    do {
        try {
            var scanner = new Scanner(System.in);
            System.out.print("Enter an integer: ");
            number = scanner.nextInt();
            successful = true;
        } catch (InputMismatchException e) {
            attempts++;
            System.out.println("Attempt #" + attempts);
            if (attempts == MAX_ATTEMPTS) {
                throw new RuntimeException("Max attempts reached.");
            }
        }
    } while (!successful);
    return number;
}
```

The complete form of try statement (1)

```
try {
   // protected statements
}
catch (SomeException e1) {
   // ...
}
catch (AnotherException e2) {
   // ...
}
finally {
   // always execute finally block, i.e.
   // (1) if no exception is thrown;
   // (2) if an exception is caught with a catch clause;
   // (3) if an exception is not caught with a catch clause.
}
```

The complete form of try statement (2)

```
try {
   // protected statements
}
catch (SomeException | AnotherException e) {
   // ...
}
finally {
   // always execute finally block, i.e.
   // (1) if no exception is thrown;
   // (2) if an exception is caught with a catch clause;
   // (3) if an exception is not caught with a catch clause.
}
```

Set Assessment 1