WEEK FOURTEEN

Acknowledgements: Slides created based off material provided by Dr. Travis Doom

EXCEPTIONS

EXCEPTIONS & ERRORS

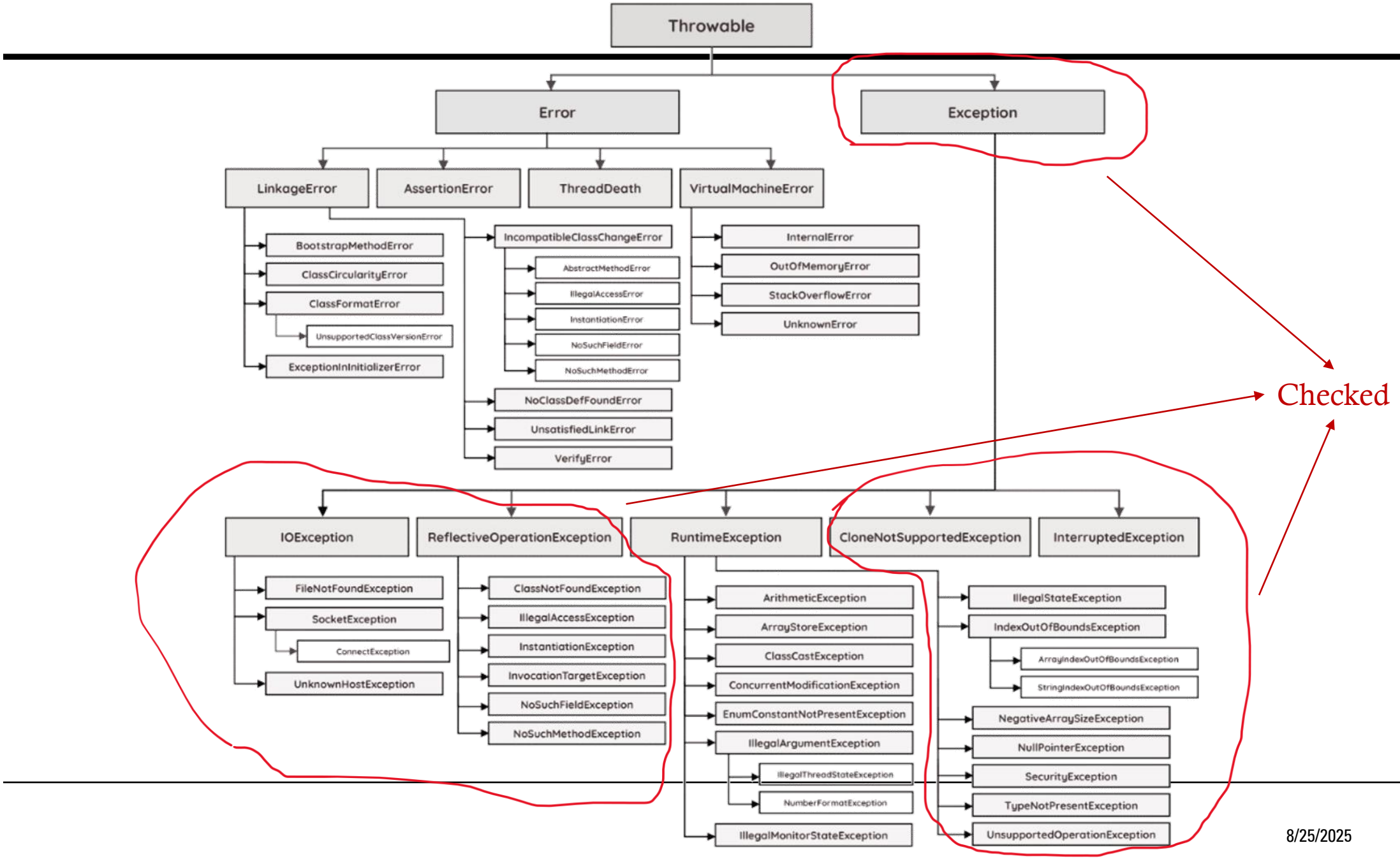
- Errors indicate problems that are not recoverable
- Exceptions can be handled (checked exceptions *must* be handled)
- Throws keyword
  - Acknowledges that the exception can occur
  - Does not explicitly handle the exception
  - “ducking responsibility”
- Try/Catch
  - Allows us to *do* something if the exception occurs rather than immediately crashing
  - “taking responsibility”

TRY/CATCH

- Using a try/catch block allows us to handle issues without our program crashes
- Try block
  - Should surround any of the code that could generate an exception
  - Any code that is dependent on the code above also needs to be in the try block
  - Once an exception is thrown from the try block the execution halts and jumps to the corresponding catch block
- Catch block
  - Can be more than one for different exceptions
  - Order from most specific to least specific
- Finally block
  - Executes last regardless of whether an exception is thrown

TRY/CATCH/FINALLY

```
try {
	// code that can cause an exception
} catch (Exception e) {
	// code that happens if that exception occurs
} finally {
	// code that happens regardless of what happens above
}
```


Throwable
Error
Exception
LinkageError
AssertionError
ThreadDeath
VirtualMachineError
BootstrapMethodError
ClassCircularityError
ClassFormatError
UnsupportedClassVersionError
ExceptionInInitializerError
IncompatibleClassChangeError
AbstractMethodError
IllegalAccessError
InstantiationError
NoSuchFieldError
NoSuchMethodError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
InternalError
OutOfMemoryError
StackOverflowError
UnknownError
Checked
IOException
ReflectiveOperationException
RuntimeException
CloneNotSupportedException
InterruptedException
FileNotFoundException
SocketException
ConnectException
UnknownHostException
ClassNotFoundException
IllegalAccessException
InstantiationException
InvocationTargetException
NoSuchFieldException
NoSuchMethodException
ArithmeticException
ArrayStoreException
ClassCastException
ConcurrentModificationException
EnumConstantNotPresentException
IllegalArgumentException
IllegalThreadStateException
NumberFormatException
IllegalMonitorStateException
IllegalStateException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
TypeNotPresentException
UnsupportedOperationException

GENERATING EXCEPTIONS

- We can extend the Exception class to write our own exceptions
- Does not require much code

```
public class MyException extends Exception {
	public MyException () {
		super(“This is a big problem!!!”);
	}
}
```

THROWING A CUSTOM EXCEPTION

- Use the ‘throw’ keyword

```
public static void main(String[] args) {
	Scanner scnr = new Scanner(System.in);
	int num = scnr.nextInt();
	if (num < 0) {
		throw new MyException();
	}
}
```

MISCELLANEOUS TOPICS

- None of the following are required, and you will not be directly tested on them:
  - Enumerated types
  - Encapsulation note*
  - Copy constructors

ENUMERATED TYPES (ENUM)

- Pair a number (value) with a word (identifier)
- Very useful for encoding
- Makes code easier to read (good style)
- Each identifier in an enumerated type is an object of the type declared after the enum keyword
- Each identifier is ordered from 0 upwards

```
enum Color {RED, ORANGE, YELLOW,
GREEN, BLUE, INDIGO, VIOLET};
// RED is 0
// ORANGE is 1
// YELLOW is 2 …
// VIOLET IS 6
Color favColor = Color.BLUE;
if (favColor == Color.VIOLET)
        …
```

ENUM CONTINUED

```
enum Color {RED, ORANGE, YELLOW, GREEN,
BLUE, INDIGO, VIOLET};

Color favColor = Color.BLUE;
```

Output:

```
System.out.println(favColor);                    BLUE

System.out.println(favColor.ordinal());          4

System.out.println(Color.INDIGO.ordinal());      5
```

ENCAPSULATION NOTE

- Remember, we want to control access to our class fields
- If we write a getter for an object (not primitive type), what do we return?
  - If we return the reference to the actual field object, it can be modified even if it is private
  - Thus, we should return a *copy* of the object
  - This ensures that all the information is provided without the ability to change the class field
  - Essentially, we need to create a *new* object

COPY CONSTRUCTORS

- A constructor that has an object of the same class as a parameter
- Makes an identical copy or clone of the object

```
public class Course {
        private String name = “”;
        private int creditHours = 0;

        public Course(Course originalCourse) {
                this.setName(originalCourse.getName());
                this.setCreditHours(originalCourse.getCreditHours());
        }
}
```

SHALLOW COPY

```
public class Student {
        private ArrayList<Course> classes = new ArrayList<>();

        public Student(Student originalStudent) {
                for (Course c : originalStudent.getClasses()) {
                        classes.add(c);
                } // SHALLOW COPY: a reference to the Course is added, not a new separate object
        }         // If we modify the Course objects of the originalStudent, our new Student's
 }                // Course objects would also change
```

DEEP COPY

```
public class Student {
        private ArrayList<Course> classes = new ArrayList<>();

        public Student(Student originalStudent) {
                for (Course c : originalStudent.getClasses()) {
                        classes.add(new Course(c));
                } // DEEP COPY: a new object is created and added to classes
        }         // If we modify the Course objects of the originalStudent, our new Student's
 }                // Course objects would NOT change
```