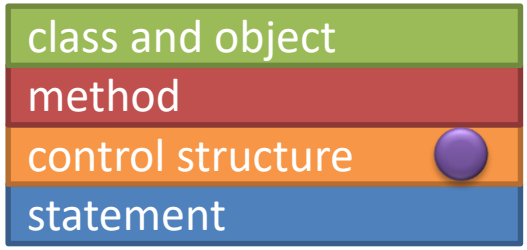
Runtime Errors and Exceptions

Week 10



16 Error Handling with Exceptions

Dr James Montgomery, james.montgomery@utas.edu.au

Types of errors in programs

Syntactical

- Compiler cannot interpret what we mean
- Discovered at **compile time**

Logical

- Wrong behaviour but *is* what the code says to do
- Discovered at **runtime**

Runtime errors

- Errors that cause program execution to terminate unexpectedly
- Represented by Exception and Error objects
- Result in a long, unfriendly list of method names and line numbers

Exceptions interrupt normal execution

```
public class Divide {
   public static void main(String[] args) {
      int num, quotient;
      num = 4;
      quotient = num / 0;
      System.out.println("Answer: " + quotient);
   }
}
```

Ugly error message a normal user should never have to see

*When an exception is **thrown**, by the system or by your code, the normal flow of execution is interrupted and the instructions after the exception are not executed*

Exceptions and Errors

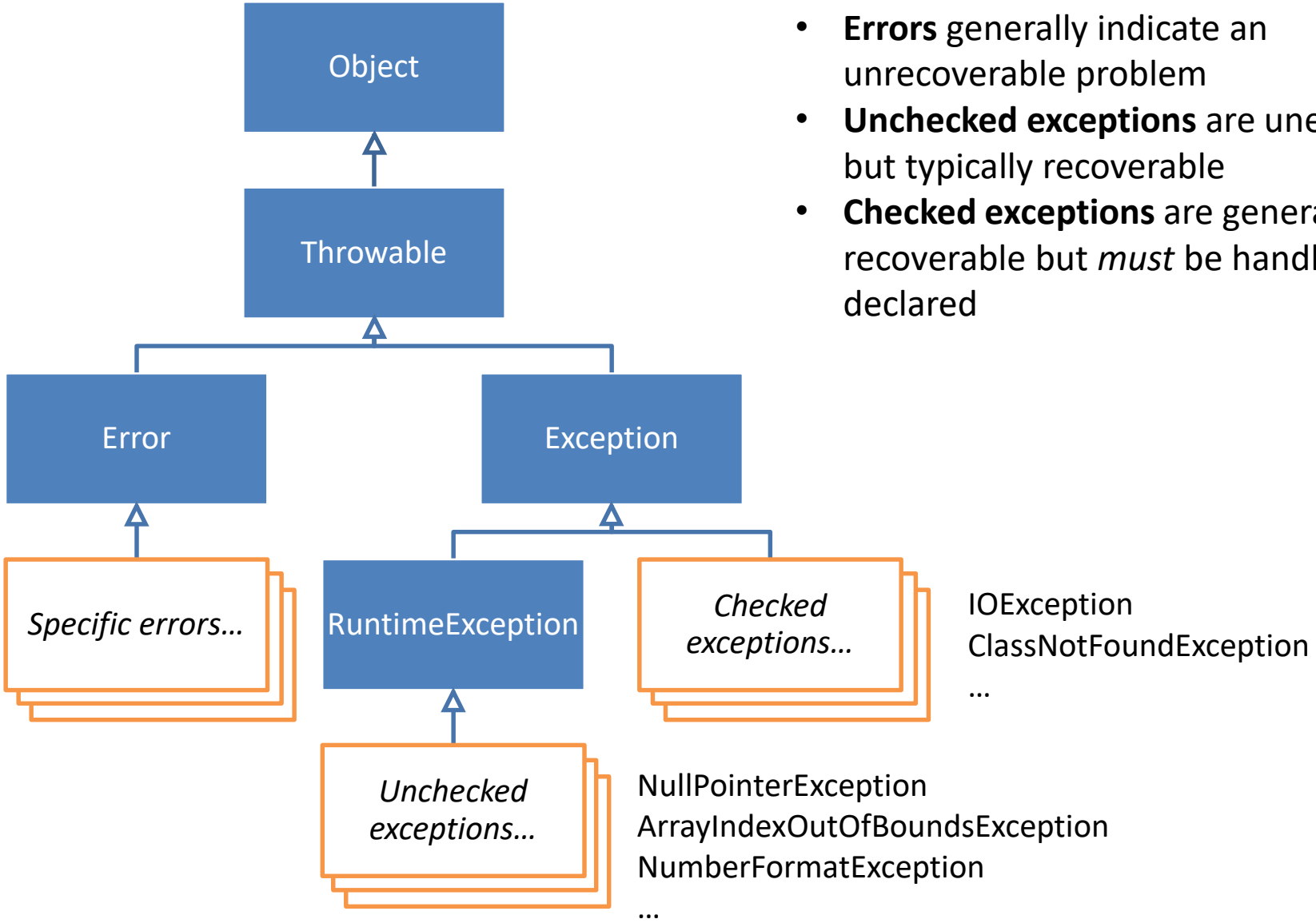
Exceptions

- indicate either an unusual or erroneous condition
- may be “caught” and “handled” by another part of the program
- Program can be separated into
  - normal execution flow
  - exception execution flow

Errors

- indicate an unrecoverable situation
- should not be caught

Different kinds of exception



- **Errors** generally indicate an unrecoverable problem
- **Unchecked exceptions** are unexpected but typically recoverable
- **Checked exceptions** are generally recoverable but *must* be handled or declared

Unchecked v Checked exceptions

Unchecked

- Compiler *doesn't* check if your program handles these
- Does not require explicit handling
- If no handling then exception automatically propagated to next level (caller)
- Examples:
  - `ArrayIndexOutOfBounds Exception`
  - `ArithmeticException`
  - `NumberFormatException`

Checked

- Compiler *does* check if your program handles these
  - Syntax error if not handled
- Example:
  - `IOException`, can be thrown when reading from "standard input" (keyboard) or a file (outside the scope of the unit)
  - `Scanner` class deals with this (so we do not need to)
  - You will deal with these in your *future* programming

Exception Handling: 3 ways

ignore it

- Exception propagated to top level (**main()**)
- program will terminate and produce an appropriate (but ugly) message

```
public class Divide {
  public static void main(String[] args) {
    int num, q;
    num = 4;
    q = num / 0;
    System.out.println("Answer: " + q);
  }
}
```

handle it where it occurs

- **try** the risky code
- **catch** the exception (if it occurs)

```
try {
  //risky code
} catch (Exception ex) {
  //handle the exception
}
```

handle it another place in the program

- a variant of the above

```
public int risky() throws Exception {
  //risky code
}
```

The manner in which an exception is processed is an important design consideration when programming

Handling exceptions

Syntax:

```
try {
    //Code that might
    //produce an exception
} catch (ExceptionClass ex) {
    //Code to handle
    //the exception
}
```

Optimistically *try* the code that might produce an exception

If exception of class *ExceptionClass* occurs, jump immediately to the catch clause and execute that block of code

This code can:

- do nothing
- throw/re-throw exception
- report error to the user
- execute ‘correcting’ code
- abort program

Demonstrations: **ignore** or handle

Divide.java has no exception handling. Sample run:

```
Enter number to divide
3
Enter number to divide by
0
Exception in thread "main" java.lang.ArithmeticException:
/ by zero
    at Divide.main(Divide.java:16)
```

Line where the error occurred

Type of exception that occurred

Demonstrations: ignore or **handle**

DivideAndCatch.java handles the exception

1. Must identify likely exception: divide by zero
2. Must decide on action to take: report the error

Sample run:

```
Enter number to divide
3
Enter number to divide by
0
Cannot do that division
Do another? (y/n)
n
program over
```