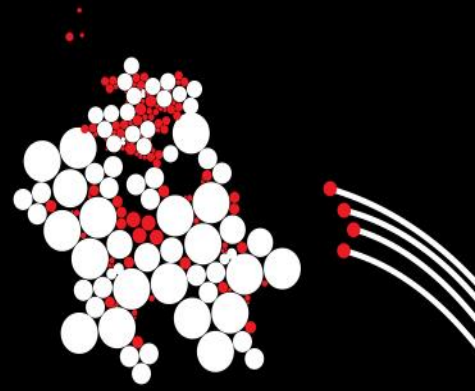
UNIVERSITY OF TWENTE.



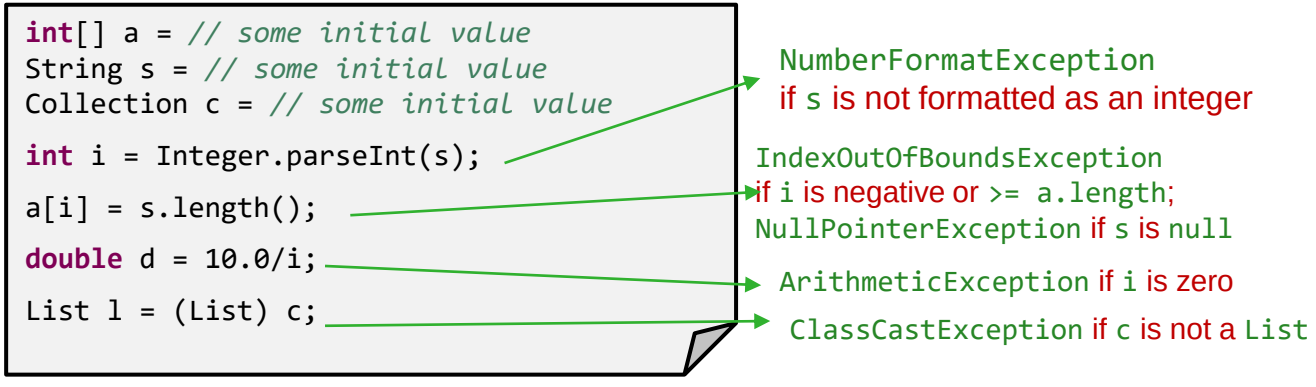
Exceptions

Topic of Software Systems (TCS module 2)

Lecturer: Faizan Ahmed

RUN-TIME EXCEPTIONS

- You have encountered some standard Java exceptions
- E.g., what can go wrong in the following fragment?



- All these are so-called *run-time exceptions*
- Their names (`NullPointerException` etc.) are in fact class names
  - Hence, reference types

HANDLING EXCEPTIONS: THE TRY-STATEMENT

- Surround code with try-block
  - Add catch-block for exceptions you want to handle

```
try {
  int i = Integer.parseInt(s);
  a[i] = s.length();
  double d = 10.0/i;
  List l = (List) c;
} catch (NumberFormatException e) {
  System.out.println(s + "is not a number");
}
```

HANDLING EXCEPTIONS: THE TRY-STATEMENT

- Surround code with try-block
  - Add catch-block for exceptions you want to handle

```
try {
   int i = Integer.parseInt(s);
   a[i] = s.length();
   double d = 10.0/i;
   List l = (List) c;
} catch (NumberFormatException e) {
   System.out.println(s + "is not a number");
} catch (ArithmeticException e) {
   System.out.println("Cannot calculate i, as s is zero");
} catch (IndexOutOfBoundsException | NullPointerException e) {
   e.printStackTrace();
}
```

One `try` can have multiple `catch`-blocks

Each `catch` can do its own thing

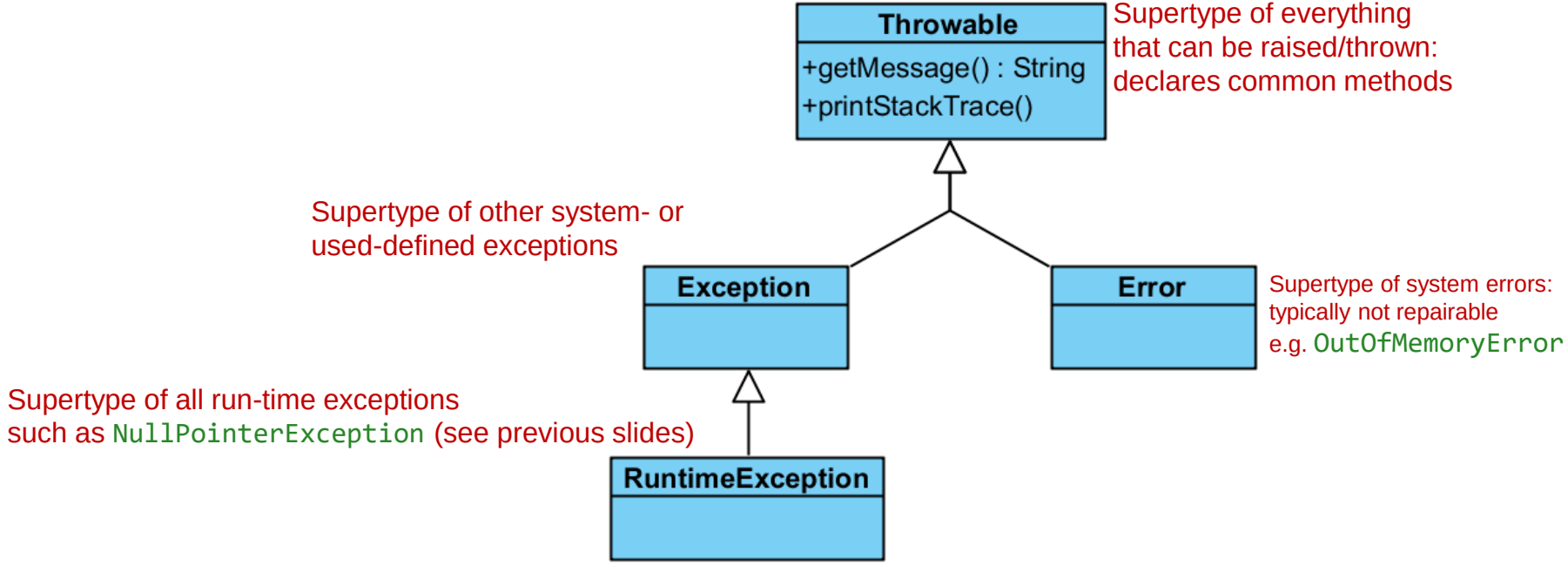
You can also combine exceptions

`e` is an object, you can call methods on it

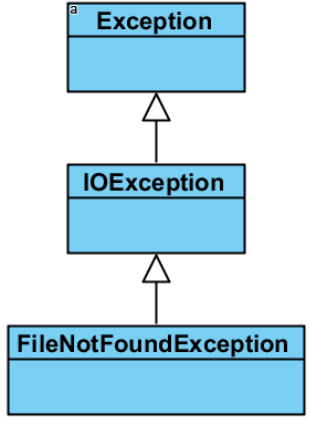
(`printStackTrace` prints a message on `System.err` showing what went wrong and where)

EXCEPTION HIERARCHY

- There are multiple levels of exceptions



CHECKED EXCEPTIONS



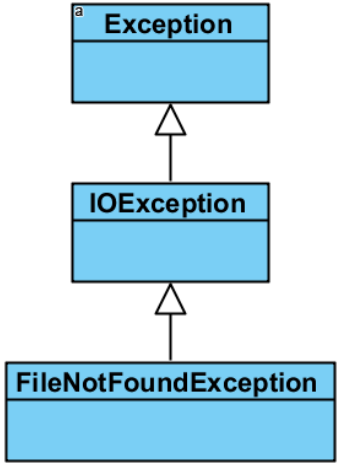
- Exceptions that are not run-time exceptions
  - The compiler does not allow you to ignore them
  - In standard Java, they typically occur when doing I/O
- E.g., the following does not compile:

```
String name = System.console().readLine();   throws FileNotFoundException
BufferedReader r = new BufferedReader(new FileReader(name));
System.out.println("First line: "+r.readLine());  throws IOException
```

  - Example solution:

```
BufferedReader r = null;
while (r == null) {
    String name = System.console().readLine("Enter filename: ");
    try {
        r = new BufferedReader(new FileReader(name));
        System.out.println("First line: "+r.readLine());
    } catch (FileNotFoundException e) {   //IOException still uncaught
        System.console().printf("File %s does not exist%n", name);
    }
}
```

CHECKED EXCEPTIONS



- Exceptions that are not run-time exceptions
  - The compiler does not allow you to ignore them
  - In standard Java, they typically occur when doing I/O
- E.g., the following does not compile:

```
String name = System.console().readLine();
BufferedReader r = new BufferedReader(new FileReader(name));
System.out.println("First line: "+r.readLine());
```

  - Example solution:

```
BufferedReader r = null;
while (r == null) {
   String name = System.console().readLine("Enter filename: ");
   try {
      r = new BufferedReader(new FileReader(name));
      System.out.println("First line: "+r.readLine());
   } catch (IOException e) {//also catches subclasses of IOException
      System.console().printf("File %s not suitable%n", name);
   }                                   message is now less specific
}
```

PASSING UP EXCEPTIONS

- Rather than catching exceptions, you can also pass them up
  - Add **`throws`** clause to method declaration

Supertype of both thrown exception

```
public BufferedReader getLine() throws IOException {
   String name = System.console().readLine("Enter filename: ");
   BufferedReader r = new BufferedReader(new FileReader(name));
   return r.readLine();
}
```

- Now the exception must be handled by the caller

```
BufferedReader r = null;
while (r == null) {
   try {
      System.out.println(getLine());
   } catch (IOException e) {
      System.console().printf("File %s not suitable%n", name);
   }
}
```

EXCEPTIONS: THROWING YOUR OWN

```
void setWord(String old, String newWord) throws Exception {
  if (!testWord(old)) {
    throw new Exception("Old password wrong");
  }
  if (!acceptable(newWord)) {
    throw new Exception("New password not acceptable");
  }
  setWord(newWord);
}
```

`throw` statement

`Exception` has constructor with `String`-parameter: string can be retrieved by `getMessage()`

EXCEPTIONS: DEFINING YOUR OWN

- Rather than using generic `Exception` class
  - You can define a subclass

```
public class PasswordException extends Exception {
   public PasswordException(String message) {
      super(message);
   }
}
```

  - This makes your code better understandable

```
void setWord(String old, String newWord) throws PasswordException {
   if (!testWord(old)) {
      throw new PasswordException("Old password wrong");
   }
   if (!acceptable(newWord)) {
      throw new PasswordException("New password not acceptable");
   }
   setWord(newWord);
}
```

TRY WITH RESOURCES

- Some classes need to be *closed*
  - In particular, I/O-based classes such as `Reader`
  - Not closing them runs the risk of resource leakage
    - E.g., the same file may not be renamed or deleted while open

```
try (declare AutoClosable objects) {
   // do some stuff
} catch (exceptions) {
   // this part is optional
}
```

- Afterwards, `close()` will be called on all objects thus declared

```
String name = System.console().readLine("Enter filename: ");
try (BufferedReader r = new BufferedReader(new FileReader(name))) {
   System.out.println("First line: " + r.readLine());
} catch (IOException e) {
   System.console().printf("File %s has a problem%n", name);
}
```

FINALLY STATEMENT

- Used for clean-up whether or not an exception is thrown

```
String name = System.console().readLine("Enter filename: ");
try {
  BufferedReader r = new BufferedReader(new FileReader(name));
  System.out.println("First line: " + r.readLine());
} catch (IOException e) {
  System.console().printf("File %s has a problem%n", name);
}finally {
  r.close();
}
```

DON'TS

- Do not construct instances of `Exception`
  - It makes your code less understandable
  - Always use your own subclass
- Do not throw your own `RuntimeException`
  - It bypasses the checking mechanism
  - Eventually you will regret this
- Never catch `Exception`
  - This is too generic and also catches ones you didn't expect
  - Among others, all `RuntimeExceptions`