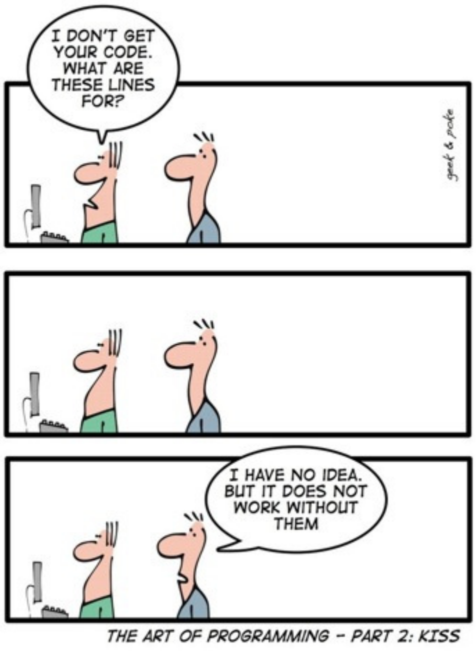

I DON'T GET YOUR CODE. WHAT ARE THESE LINES FOR?
geek & poke
I HAVE NO IDEA. BUT IT DOES NOT WORK WITHOUT THEM
THE ART OF PROGRAMMING - PART 2: KISS

OBJECT-ORIENTED PROGRAMMING

Exception Handling

Lecture #7

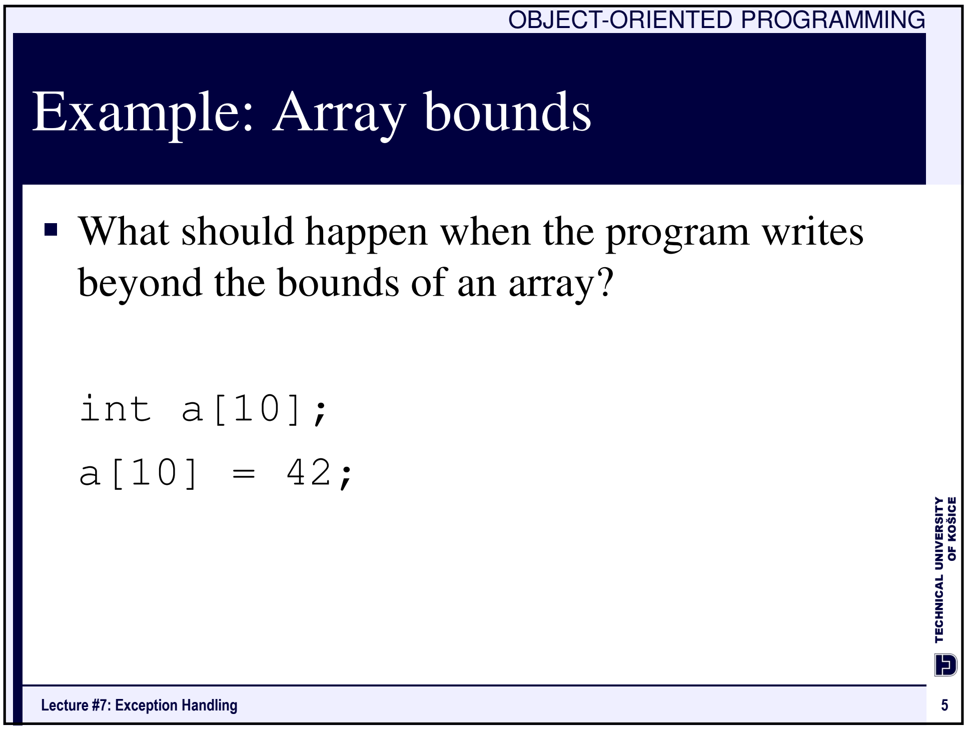
TECHNICAL UNIVERSITY OF KOSICE

Approaches to error checking

- Ignore the possibility
- Handle the error where it occurs
  - Easy to observe the error handling code
  - Clutters the “real” code
- Exception handling in object-oriented languages
  - Makes code clearer, more robust and fault-tolerant

Common errors

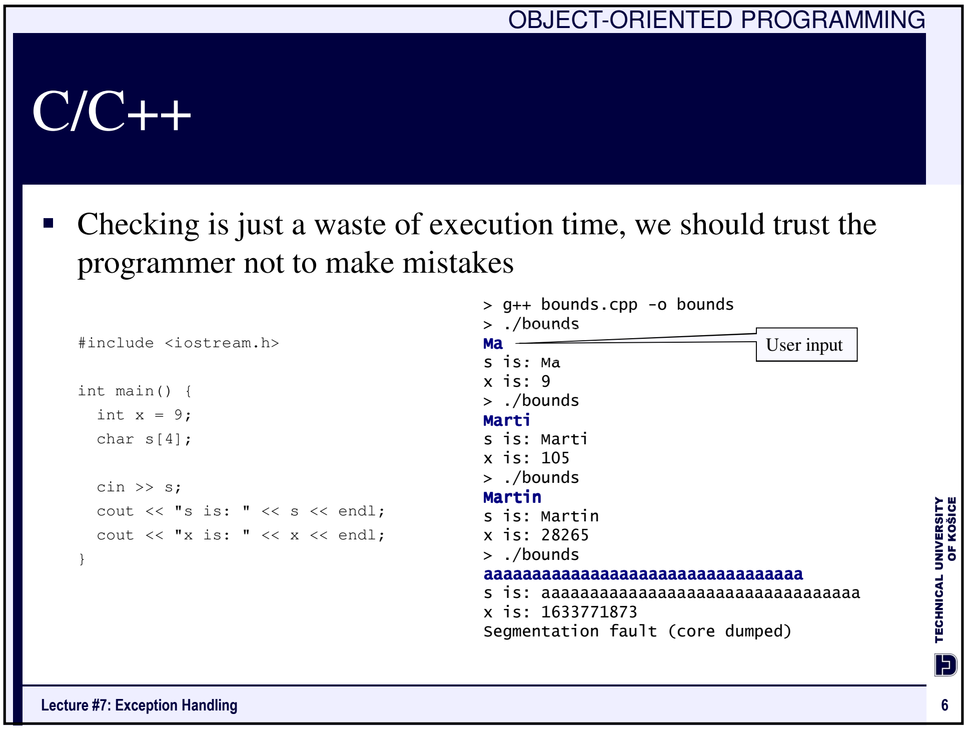
- Failure of **`new`** to allocate requested memory (or other resources)
- Array index out of bounds
- Division by zero
- Function received invalid parameters



Example: Array bounds

- What should happen when the program writes beyond the bounds of an array?

```
int a[10];
a[10] = 42;
```



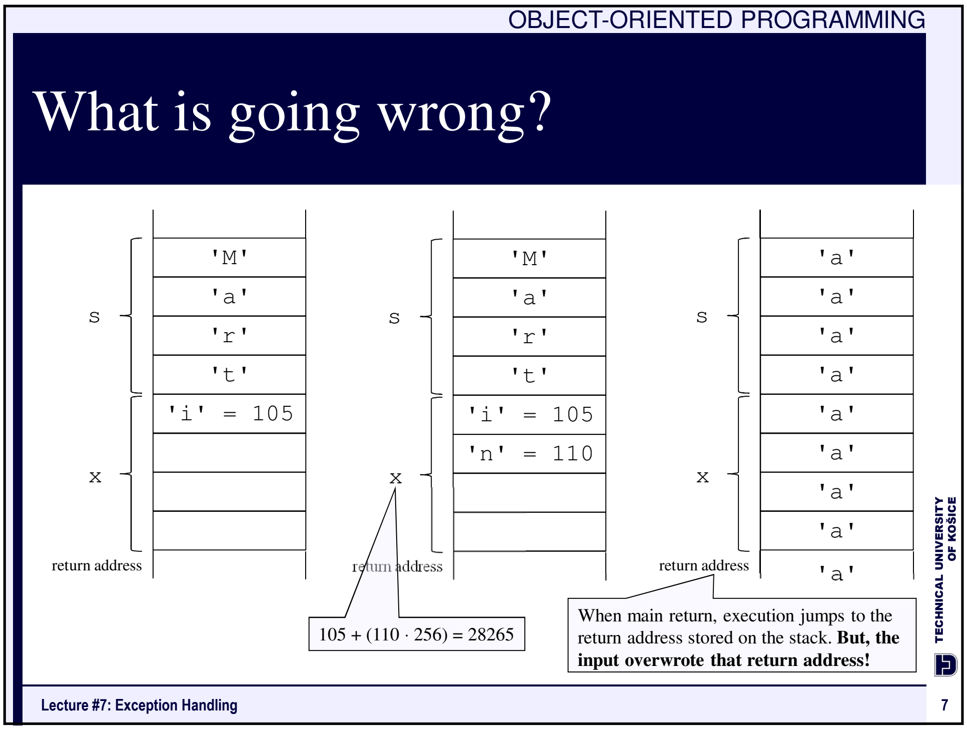
C/C++

- Checking is just a waste of execution time, we should trust the programmer not to make mistakes

```
#include <iostream.h>

int main() {
  int x = 9;
  char s[4];

  cin >> s;
  cout << "s is: " << s << endl;
  cout << "x is: " << x << endl;
}
```

```
> g++ bounds.cpp -o bounds
> ./bounds
Ma
s is: Ma
x is: 9
> ./bounds
Marti
s is: Marti
x is: 105
> ./bounds
Martin
s is: Martin
x is: 28265
> ./bounds
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
s is: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
x is: 1633771873
Segmentation fault (core dumped)
```

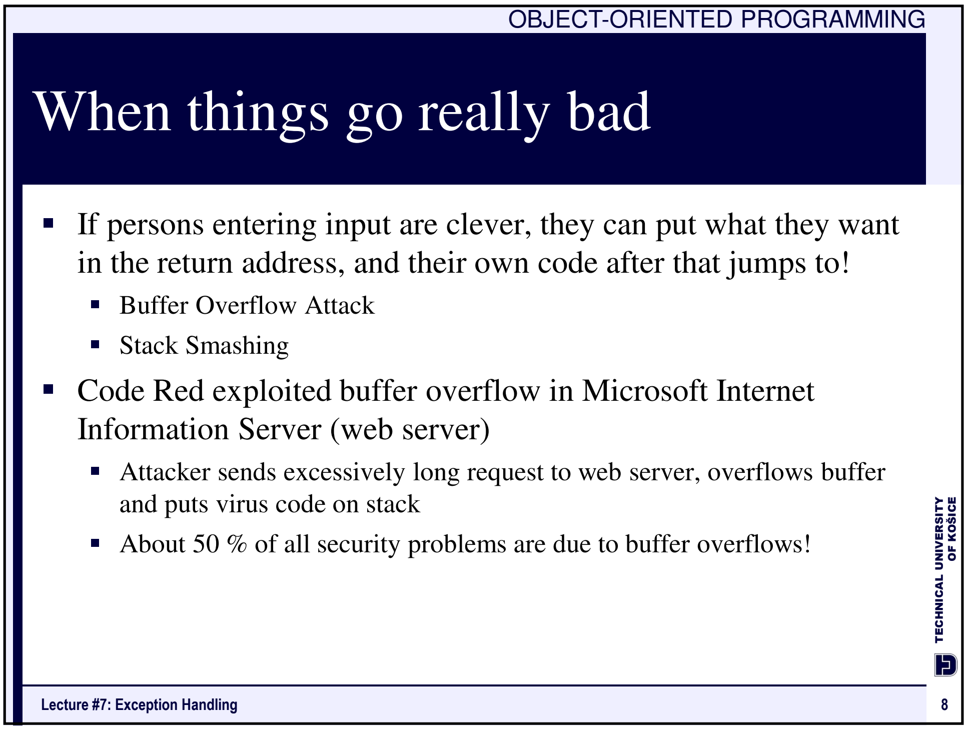
User input



What is going wrong?

105 + (110 · 256) = 28265

When main return, execution jumps to the return address stored on the stack. **But, the input overwrote that return address!**



When things go really bad

- If persons entering input are clever, they can put what they want in the return address, and their own code after that jumps to!
  - Buffer Overflow Attack
  - Stack Smashing
- Code Red exploited buffer overflow in Microsoft Internet Information Server (web server)
  - Attacker sends excessively long request to web server, overflows buffer and puts virus code on stack
  - About 50 % of all security problems are due to buffer overflows!

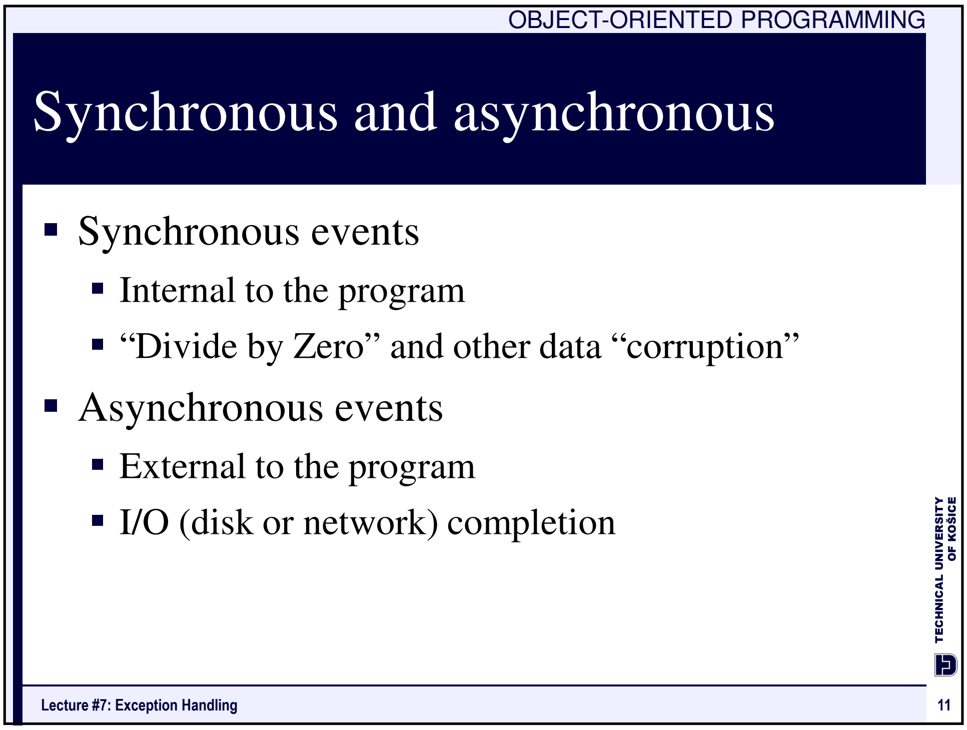
Array bounds in Java

```
public class BoundsExample {
  public static void main(String[] args) {
    String fileName = args[0];
  }
}
```

```
> javac BoundsExample.java
> java BoundsExample
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
     at BoundsExample.main(BoundsExample.java:3)
```

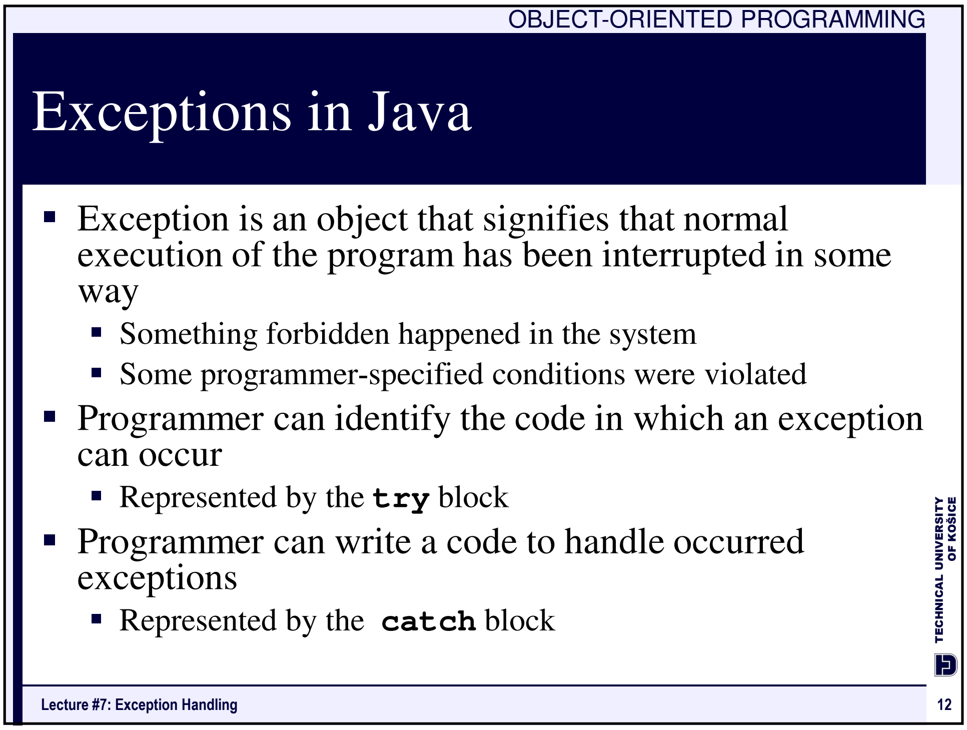
Exception handling

- Catch errors before they occur
- Well-suited for
  - Synchronous events
  - Recoverable errors
  - Fatal errors
- Poorly-suited for
  - Asynchronous events (unscripted events)
  - Normal program control



Synchronous and asynchronous

- Synchronous events
  - Internal to the program
  - “Divide by Zero” and other data “corruption”
- Asynchronous events
  - External to the program
  - I/O (disk or network) completion



Exceptions in Java

- Exception is an object that signifies that normal execution of the program has been interrupted in some way
  - Something forbidden happened in the system
  - Some programmer-specified conditions were violated
- Programmer can identify the code in which an exception can occur
  - Represented by the **`try`** block
- Programmer can write a code to handle occurred exceptions
  - Represented by the **`catch`** block

Exceptions are thrown

- Exception object is “thrown” to the program
  - When an exceptional situation occurs exception object is propagated to the program, so that it can be handled somehow or the execution is stopped
- Implicit throwing
  - The exceptional situation occurs in the system (either OS or virtual machine)
  - The exception object is thrown to the program by the system
- Explicit throwing
  - The programmer within a code can identify an exceptional situation
  - The programmer writes a code to create an exception object and to throw it using **`throw`** command

Exceptions are caught

- Thrown exception are “caught” by the program
  - Caught exceptions can be handled
- The code where an exceptional situation can occur must appear in **`try`** block
  - The code in **`try`** block is interrupted
- The exception is caught and handled using the code in **`catch`** block
  - Interrupted code from **`try`** block continues in **`catch`** block written for the specific type (or supertype) of exception object

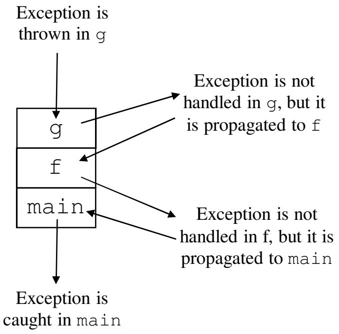
Interruption of the method

- Exceptions are thrown in methods
  - The execution of the method is interrupted
  - If it is in **`try`** block, **`catch`** block can handle it
- What if the method does not like to handle an exception (or do not know how)?
  - The method declares that this exception type (or its supertype) can be thrown further using keyword **`throws`** in its declaration
  - Then the execution of the method is interrupted and the uncaught exception object is thrown further to the caller method
  - Later it can be either caught and handled or thrown further down the call stack

How Java handles exceptions?

- After an exception is thrown, it is propagated down the call stack until a method is found that can handle it

```
public static void main(String[] args) {
  A a = new A();
  try {
    a.f();
  } catch (Exception e) {
    // recovery operations here
  }
}

public class A {
  public void f() throws Exception {
    g();
  }
  public void g() throws Exception {
    throw new Exception();
  }
}
```

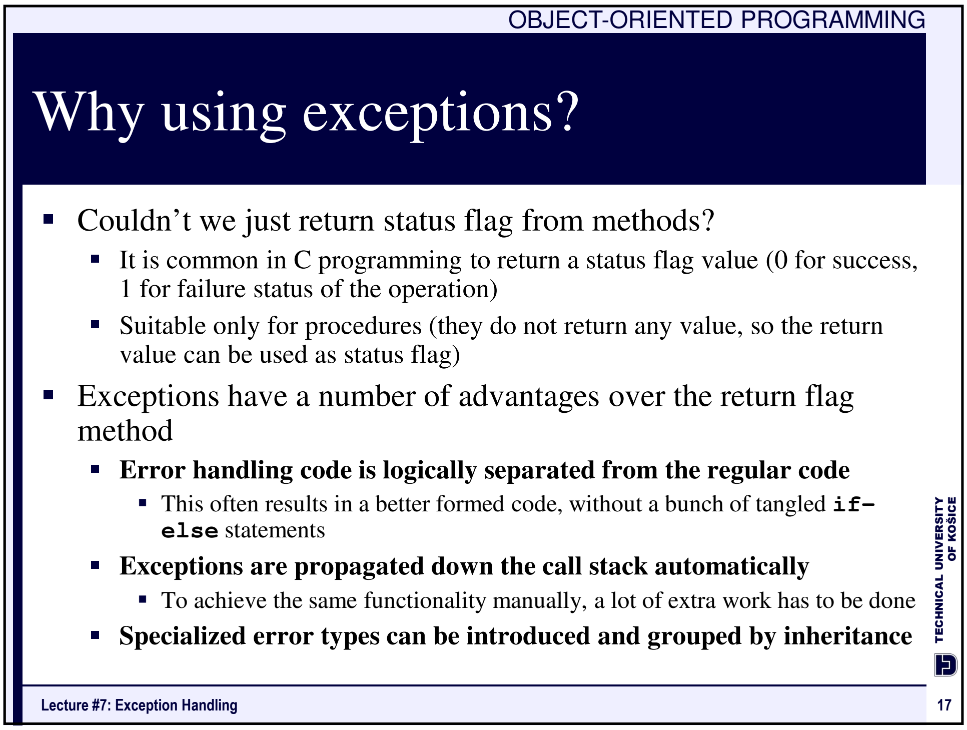


Exception is thrown in `g`

Exception is not handled in `g`, but it is propagated to `f`

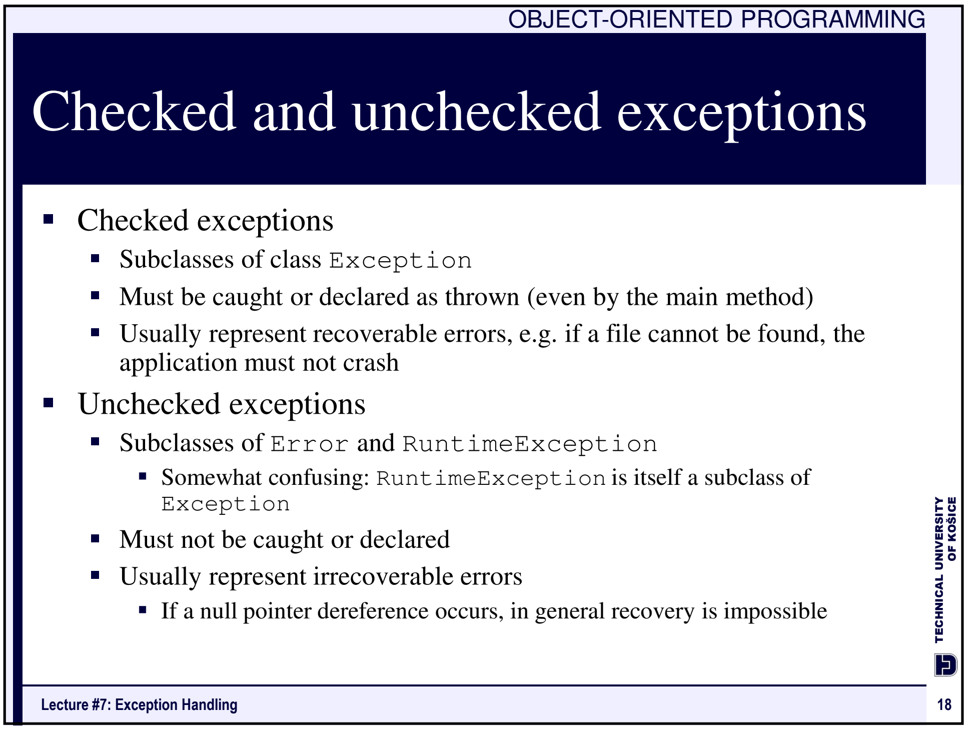
Exception is not handled in f, but it is propagated to `main`

Exception is caught in `main`



Why using exceptions?

- Couldn’t we just return status flag from methods?
  - It is common in C programming to return a status flag value (0 for success, 1 for failure status of the operation)
  - Suitable only for procedures (they do not return any value, so the return value can be used as status flag)
- Exceptions have a number of advantages over the return flag method
  - **Error handling code is logically separated from the regular code**
    - This often results in a better formed code, without a bunch of tangled **`if-else`** statements
  - **Exceptions are propagated down the call stack automatically**
    - To achieve the same functionality manually, a lot of extra work has to be done
  - **Specialized error types can be introduced and grouped by inheritance**



Checked and unchecked exceptions

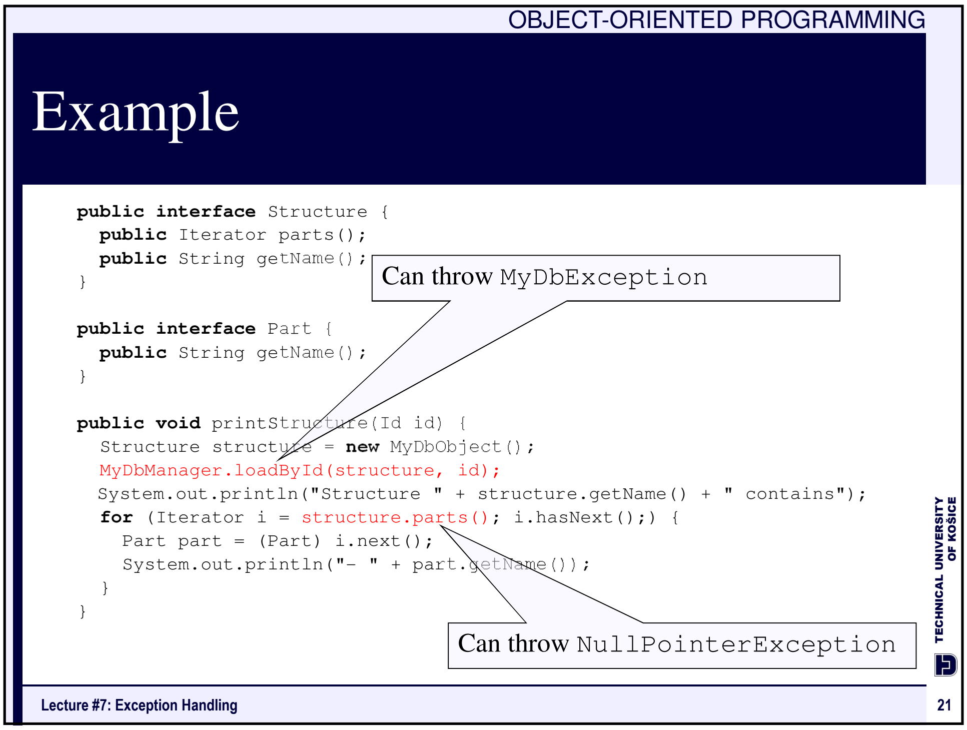
- Checked exceptions
  - Subclasses of class `Exception`
  - Must be caught or declared as thrown (even by the main method)
  - Usually represent recoverable errors, e.g. if a file cannot be found, the application must not crash
- Unchecked exceptions
  - Subclasses of `Error` and `RuntimeException`
    - Somewhat confusing: `RuntimeException` is itself a subclass of `Exception`
  - Must not be caught or declared
  - Usually represent irrecoverable errors
    - If a null pointer dereference occurs, in general recovery is impossible

Information carried by an exception

- The **type** of an exception
  - Good design practice to use different exception classes for different kinds of exceptional situations
  - **Polymorphic class hierarchy of exceptions**
- **Stack trace** – the state of the call stack at the moment the exception was thrown
  - Chain of methods and their “active” instructions
  - The `printStackTrace()` method of `Exception`
- Any **additional information**
  - E.g. a verbal explanatory message
    - The `getMessage()` method of `Exception`
- Security implication: Can an attacker use exception type and stack trace information to gain insight in the part of the application code not accessible to the attacker directly?

Rules for secure class design: validity checks

- Many methods have restrictions on validity of parameters
  - E.g., object references often must not be null
- Many intermediate results can be checked for validity (sanity checks)
- If no validity checks are present, bad things can happen
  - Execution of a method may fail unexpectedly
  - A method may terminate without failure, but cause failure at some later point in the execution
  - Etc.
- Use exceptions to implement validity checks
  - Often `IllegalArgumentException`, `IndexOutOfBoundsException`, and `NullPointerException`



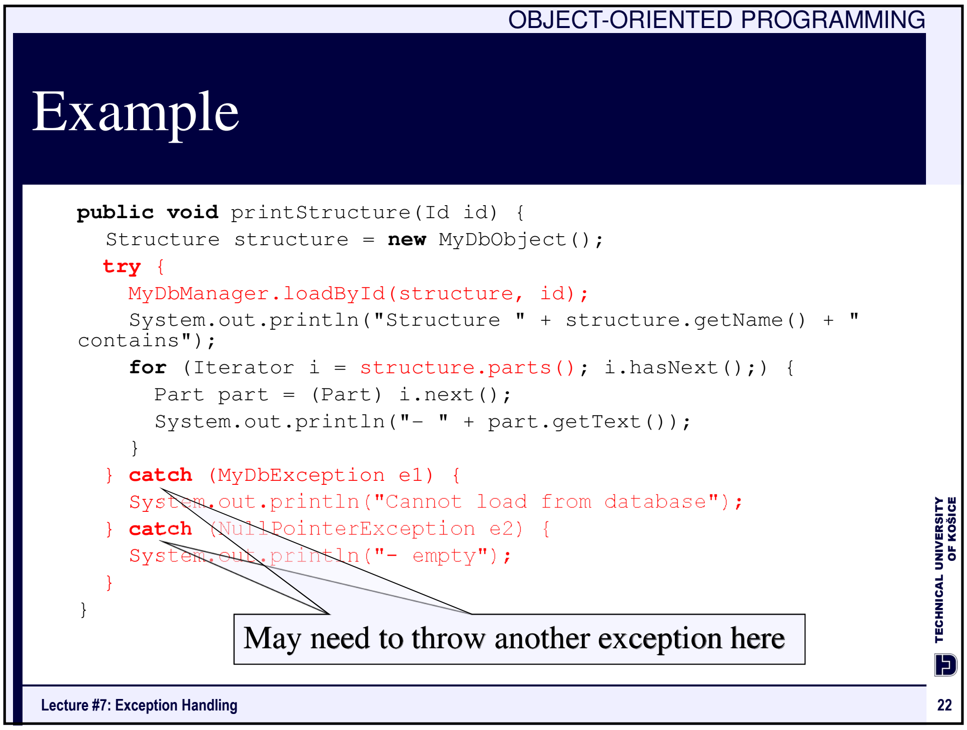
Example

```
public interface Structure {
  public Iterator parts();
  public String getName();
}

public interface Part {
  public String getName();
}

public void printStructure(Id id) {
  Structure structure = new MyDbObject();
  MyDbManager.loadById(structure, id);
  System.out.println("Structure " + structure.getName() + " contains");
  for (Iterator i = structure.parts(); i.hasNext();) {
    Part part = (Part) i.next();
    System.out.println("- " + part.getName());
  }
}
```

Can throw MyDbException

Can throw NullPointerException

TECHNICAL UNIVERSITY OF KOŠICE

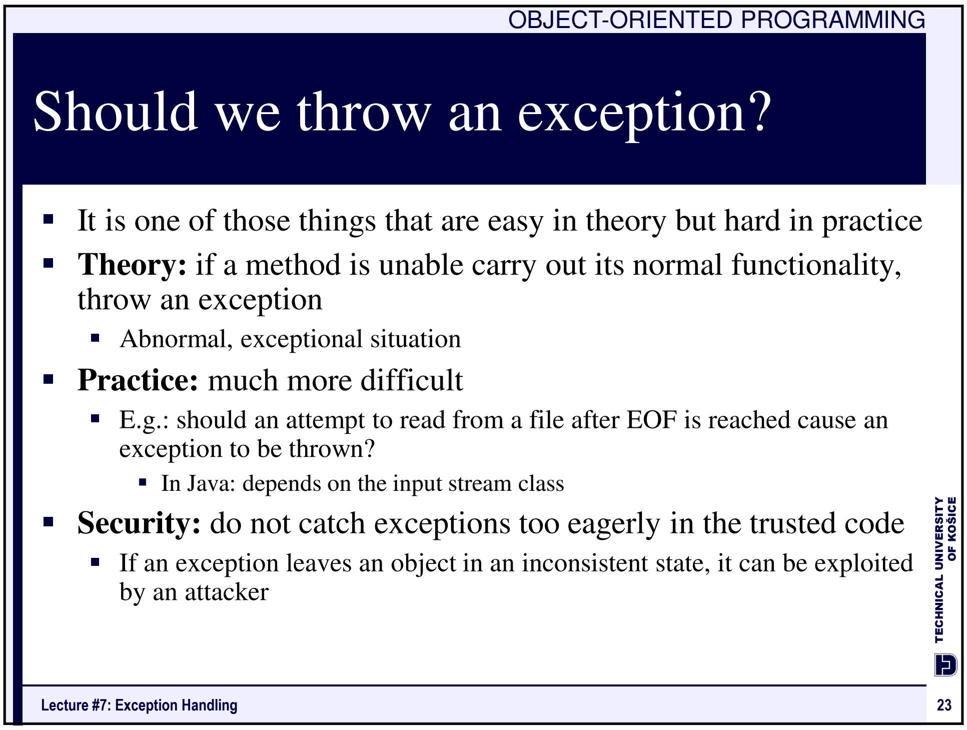


Example

```
public void printStructure(Id id) {
  Structure structure = new MyDbObject();
  try {
    MyDbManager.loadById(structure, id);
    System.out.println("Structure " + structure.getName() + "
contains");
    for (Iterator i = structure.parts(); i.hasNext();) {
      Part part = (Part) i.next();
      System.out.println("- " + part.getText());
    }
  } catch (MyDbException e1) {
    System.out.println("Cannot load from database");
  } catch (NullPointerException e2) {
    System.out.println("- empty");
  }
}
```

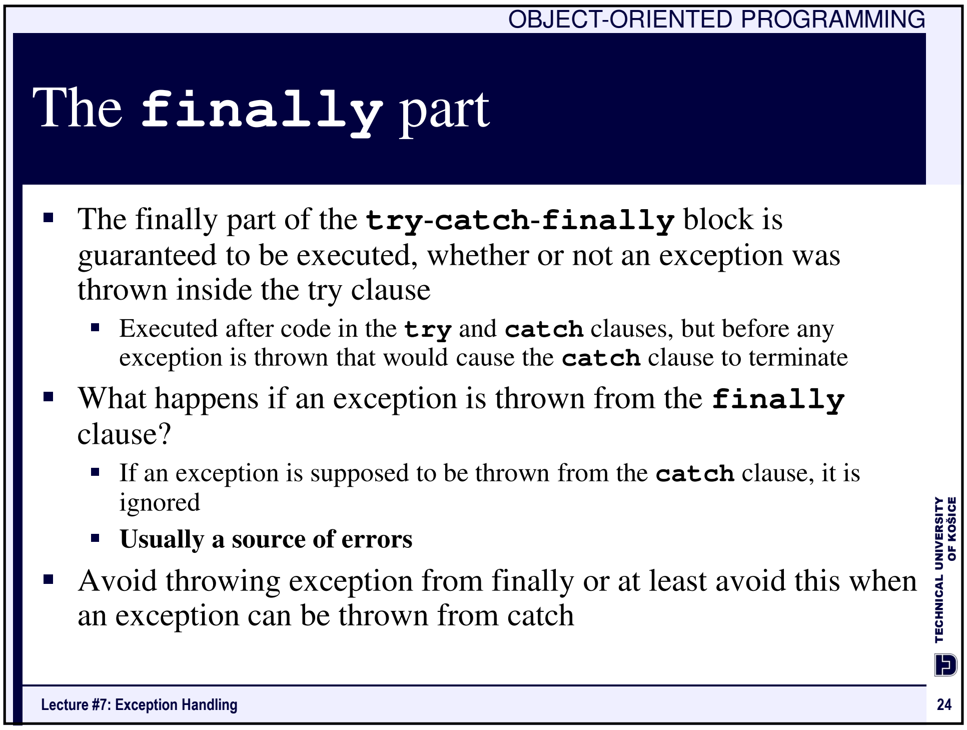
May need to throw another exception here

TECHNICAL UNIVERSITY OF KOŠICE



Should we throw an exception?

- It is one of those things that are easy in theory but hard in practice
- **Theory:** if a method is unable carry out its normal functionality, throw an exception
  - Abnormal, exceptional situation
- **Practice:** much more difficult
  - E.g.: should an attempt to read from a file after EOF is reached cause an exception to be thrown?
    - In Java: depends on the input stream class
- **Security:** do not catch exceptions too eagerly in the trusted code
  - If an exception leaves an object in an inconsistent state, it can be exploited by an attacker



The `finally` part

- The finally part of the **`try`**-**`catch`**-**`finally`** block is guaranteed to be executed, whether or not an exception was thrown inside the try clause
  - Executed after code in the **`try`** and **`catch`** clauses, but before any exception is thrown that would cause the **`catch`** clause to terminate
- What happens if an exception is thrown from the **`finally`** clause?
  - If an exception is supposed to be thrown from the **`catch`** clause, it is ignored
  - **Usually a source of errors**
- Avoid throwing exception from finally or at least avoid this when an exception can be thrown from catch

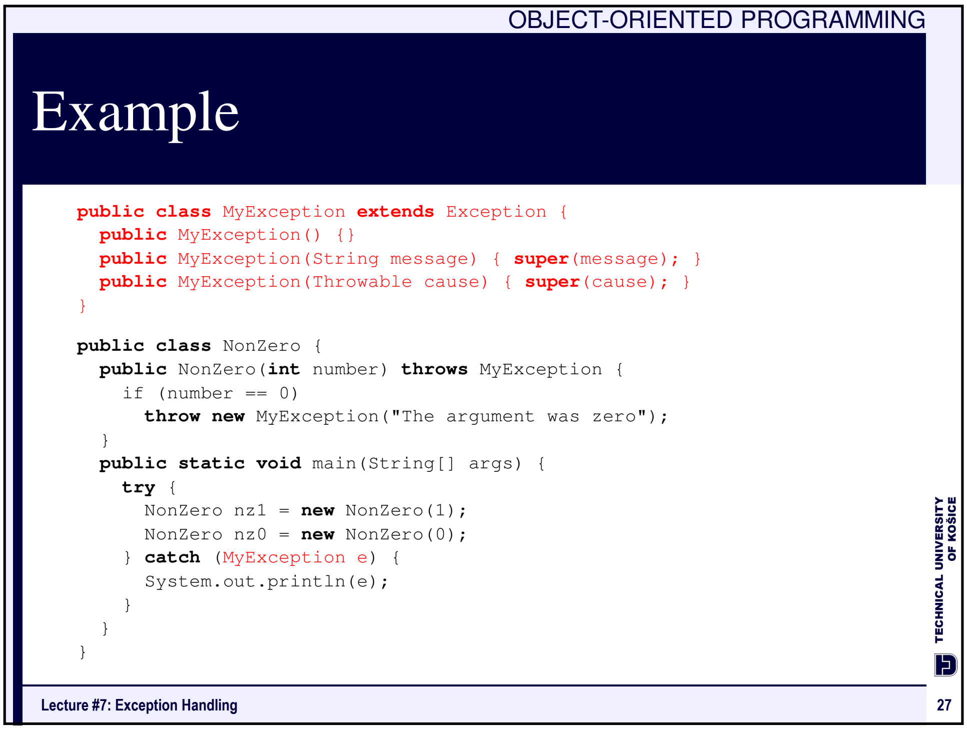
Example

```
public void createOrUpdate(Session session) {
  Id id = session.getObjectId();
  RunData data = session.getRunData();
  MyDbObject object = new MyDbObject();
  try {
    MyDbManager.loadById(object, id);
    MyLogManager.debug("Loaded from DB");
  } catch (MyDbException e) {
    object.setId(id);
    MyLogManager.debug("New created");
  } finally {
    object.update(data);
    MyDbManager.save(obj);
    MyLogManager.debug("Updated and saved to DB");
  }
}
```

Object `obj` is updated and saved to the database even if it is loaded in **`try`** block or not

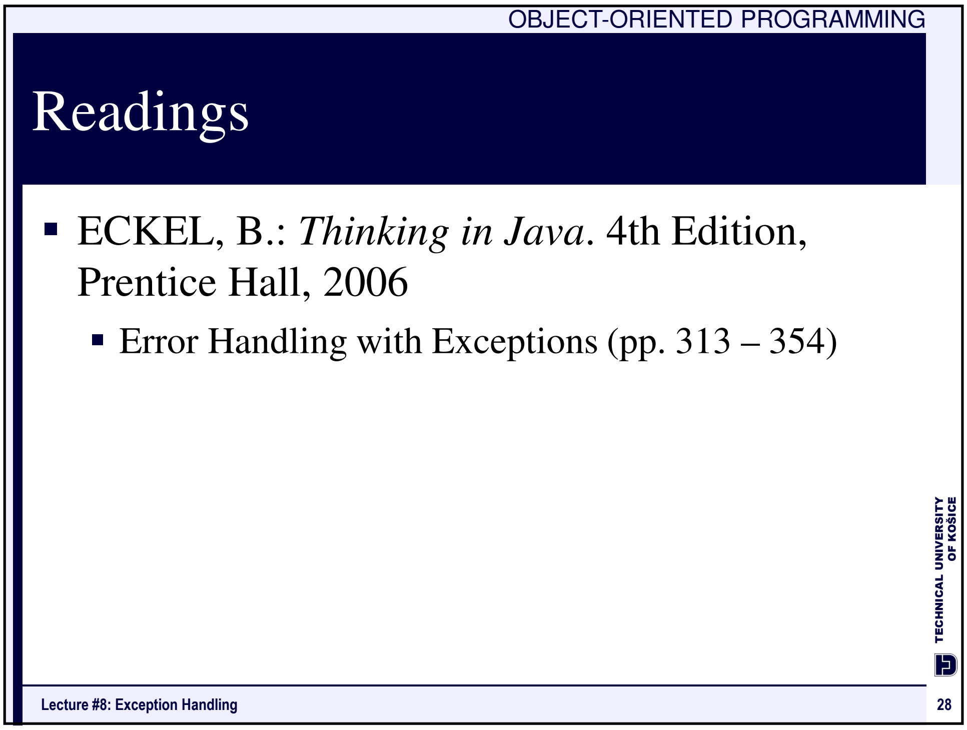
Declaring new exception type

- Most programmers use existing exception classes from the Java API or from the third party vendors
- If you do need to create an exception class, then you should extend the class `Exception` and specify three constructors



Example

```
public class MyException extends Exception {
  public MyException() {}
  public MyException(String message) { super(message); }
  public MyException(Throwable cause) { super(cause); }
}

public class NonZero {
  public NonZero(int number) throws MyException {
    if (number == 0)
      throw new MyException("The argument was zero");
  }
  public static void main(String[] args) {
    try {
      NonZero nz1 = new NonZero(1);
      NonZero nz0 = new NonZero(0);
    } catch (MyException e) {
      System.out.println(e);
    }
  }
}
```



Readings

- ECKEL, B.: *Thinking in Java*. 4th Edition, Prentice Hall, 2006
  - Error Handling with Exceptions (pp. 313 – 354)