Java Exceptions

Università di Modena e Reggio Emilia
Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

The world without exceptions

- If an error happens inside a method developers might be tempted to:
  - return a special value
  - interrupt the execution

```
List<Integer> list = new
ArrayList<Integer>();

public int getItem(int i) {
  if (list.size() <= i) {
    return -1;
  return list.get(i)
}
```

Developers must remember value/meaning of return values

```
List<Integer> list = new
ArrayList<Integer>();

public int getItem(int i) {
  if (list.size() <= i) {
    System.exit();
  return list.get(i)
}
```

Resulting code not reusable!

Never do this!

Real-world problems

- Terminating execution inside a method must be avoided. The resulting code is not reusable
- The use of return codes implies:
  - Code is messy to write and hard to read
  - Only the direct caller can intercept errors (no delegation to any upward method)

```
retval = function();
if (retval == ERROR_CODE_1) {
  // handle error 1
} else if (retval == ERROR_CODE_2) {
  // handle error 2
} …
```

An example, read a file into memory

1. Open a file
2. Determine file size
3. Allocate the needed memory
4. Read the file into memory
5. Close the file

All these operations can fail!

Errors must be checked.

First approach

- Short, readable BUT not reusable nor dependable (errors are not checked at all)

```
void loadFile()  {
   open file;
   determine file size;
   allocate memory;
   read file into memory;
   close file;
}
```

Second approach

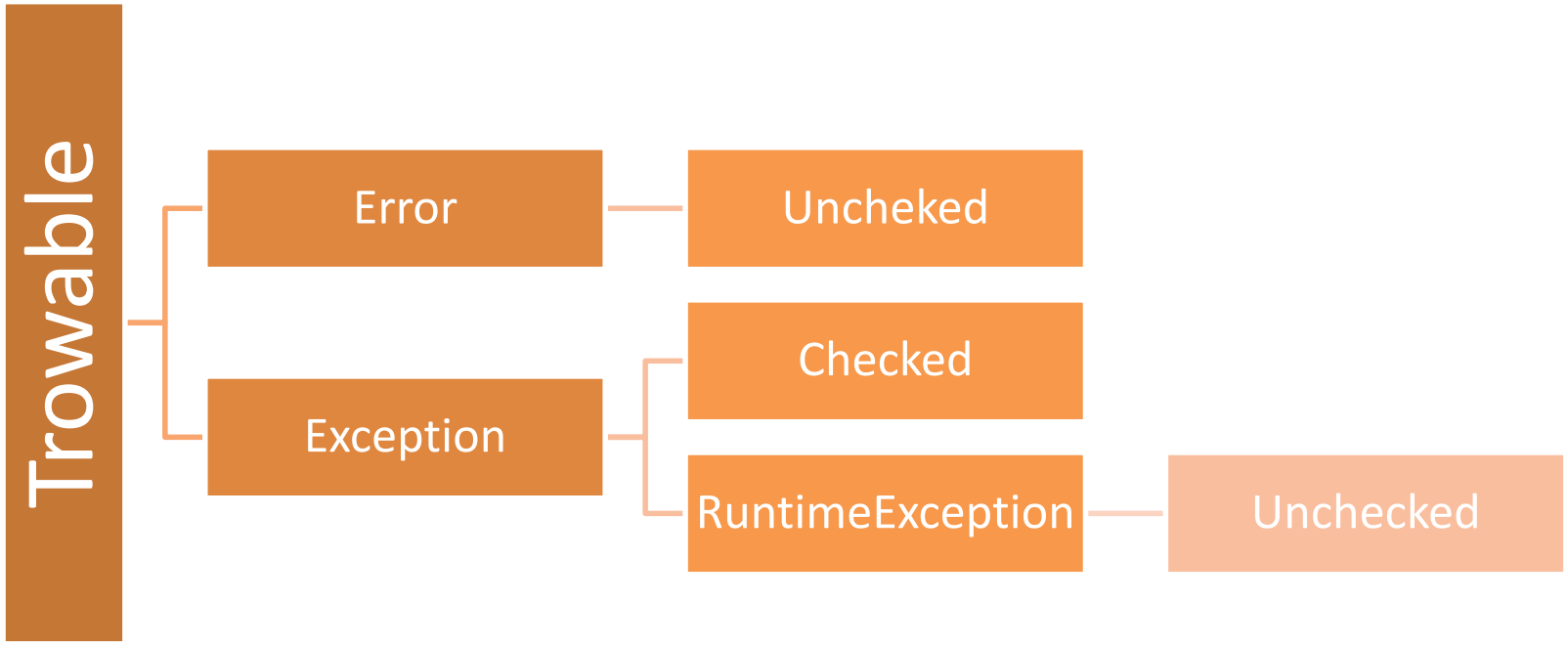
```
open file;
if(operationFailed)
  return -1;
determine file size;
if(operationFailed)
  return -2;
allocate memory;
if(operationFailed) {
  close the file;
  return -3;
}
read file into memory;
if (operationFailed) {
  close the file;
  return -4;
}
close file;
if (operationFailed)
  return  -5;

return 0;
```

- Reusable, dependable BUT long and obscure
- A lot of error-detection and error-handling code
  - To detect errors we have to remember the specification of library calls
  - Each library has its own standards

Third approach (Exceptions)

```
try   {
      open file;
      determine file size;
      allocate memory;
      read file into memory;
      close file;
} catch (fileOpenFailed)    {
     doSomething;
} catch(determineSizeFailed) {
     doSomething;
} catch (memoryAllocationFailed) {
     doSomething;
} catch (readFailed)   {
     doSomething;
} catch (fileCloseFailed)    {
     doSomething;
}
```

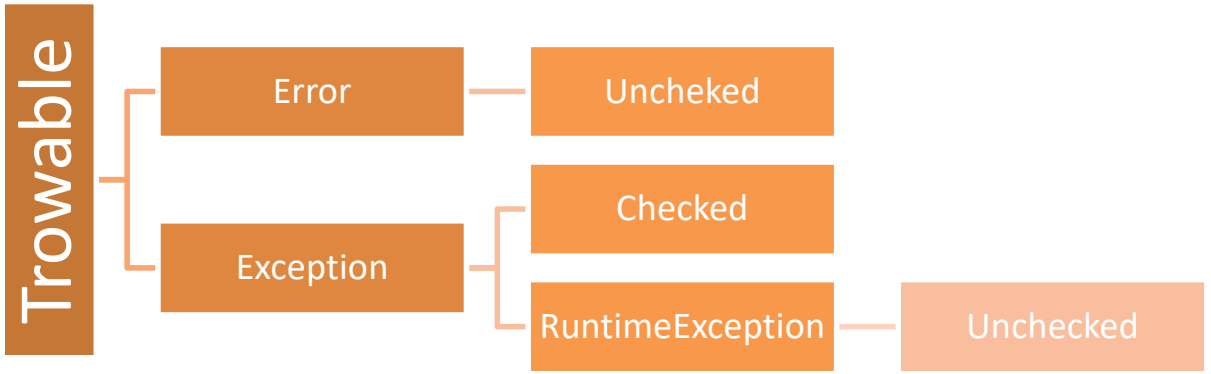
Exceptions and Errors



ING

Errors

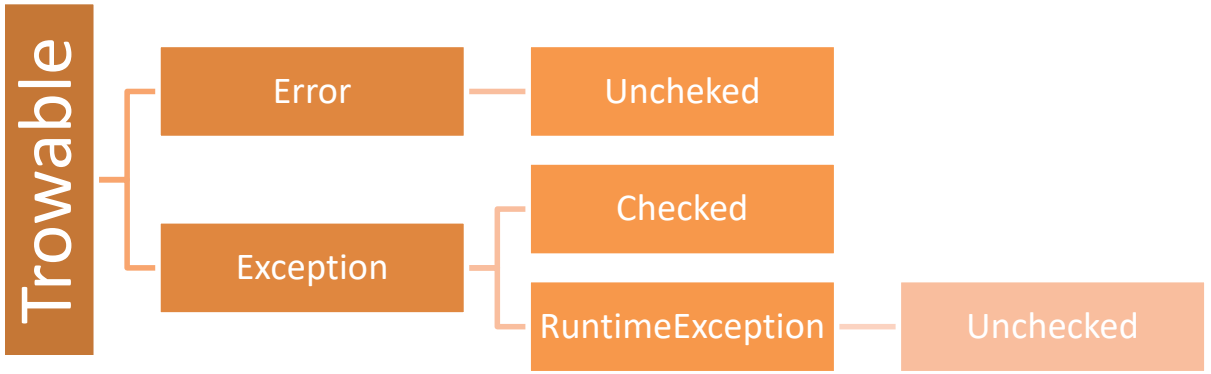
- An Error is a subclass of Throwable referring to serious issues that a reasonable application should not try to catch. Most of errors are truly abnormal conditions.
  - LinkageError indicates that a class has some dependency on another class; however, the latter class has incompatibly changed after the compilation of the former class.
  - VirtualMachineError indicates that the Java Virtual Machine is broken or has run out of resources



ING

Checked and Unchecked Exceptions

- Checked exceptions
  - Exceptions managed with try/catch and checked by the compiler
  - Examples: *IOException, SQLException, ClassNotFoundException*, …
- Unchecked exceptions
  - Their generation is not foreseen (can happen everywhere). Do not need to be managed with try/catch (not checked by the compiler).
  - Their management would make the code excessively complicated (try/catch everywhere)
  - Examples: NullPointerException, ArrayIndexOutOfBound, …



ING

Basic concepts (stack trace)

```
public class App {
    public void f(int i) {
        g(i);
    }
    public void g(int i) {
        new ArrayList().get(i);
    }
    public static void main(String[] args) {
        new App().f(5);
    }
}
```

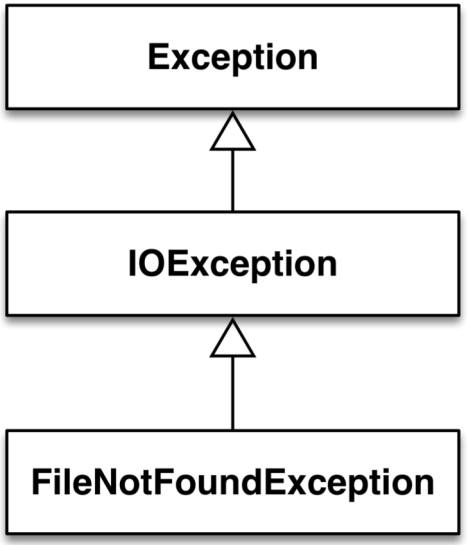
```
Exception in thread "main"
java.lang.IndexOutOfBoundsException: Index: 5, Size: 0
at java.util.ArrayList.rangeCheck(ArrayList.java:657)
at java.util.ArrayList.get(ArrayList.java:433)
at App.g(App.java:9)
at App.f(App.java:6)
at App.main(App.java:12)
```

Basic concepts (delegation)

- The code causing an error generates an exception
- At some point, up in the hierarchy of method invocations, a method might catch and handle the exception
- All methods can
  - Intercept the exception (no delegation)
  - Ignore the exception (complete delegation)
  - Intercept and generate a new exception (partial delegation)

Interception (no delegation)

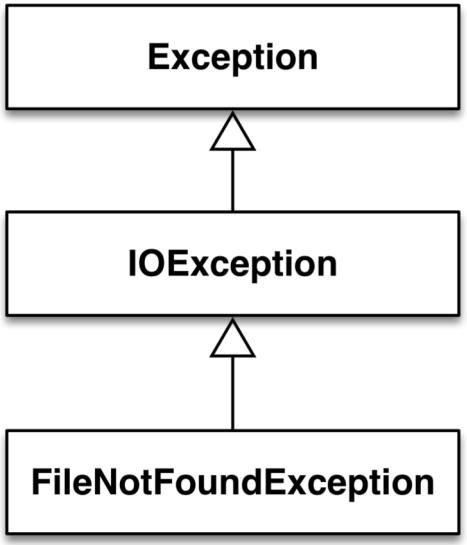
```
public static void main(String[] args) {
    char[] v = new char[256];
    try {
        FileReader f = new FileReader("test.txt");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```



ING

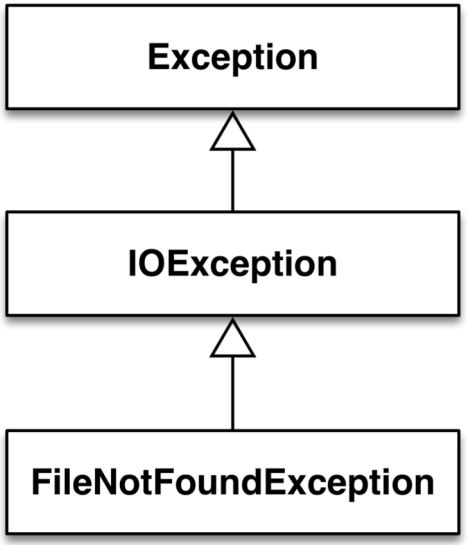
Interception (no delegation)

```
public static void main(String[] args) {
    char[] v = new char[256];
    try {
        FileReader f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```



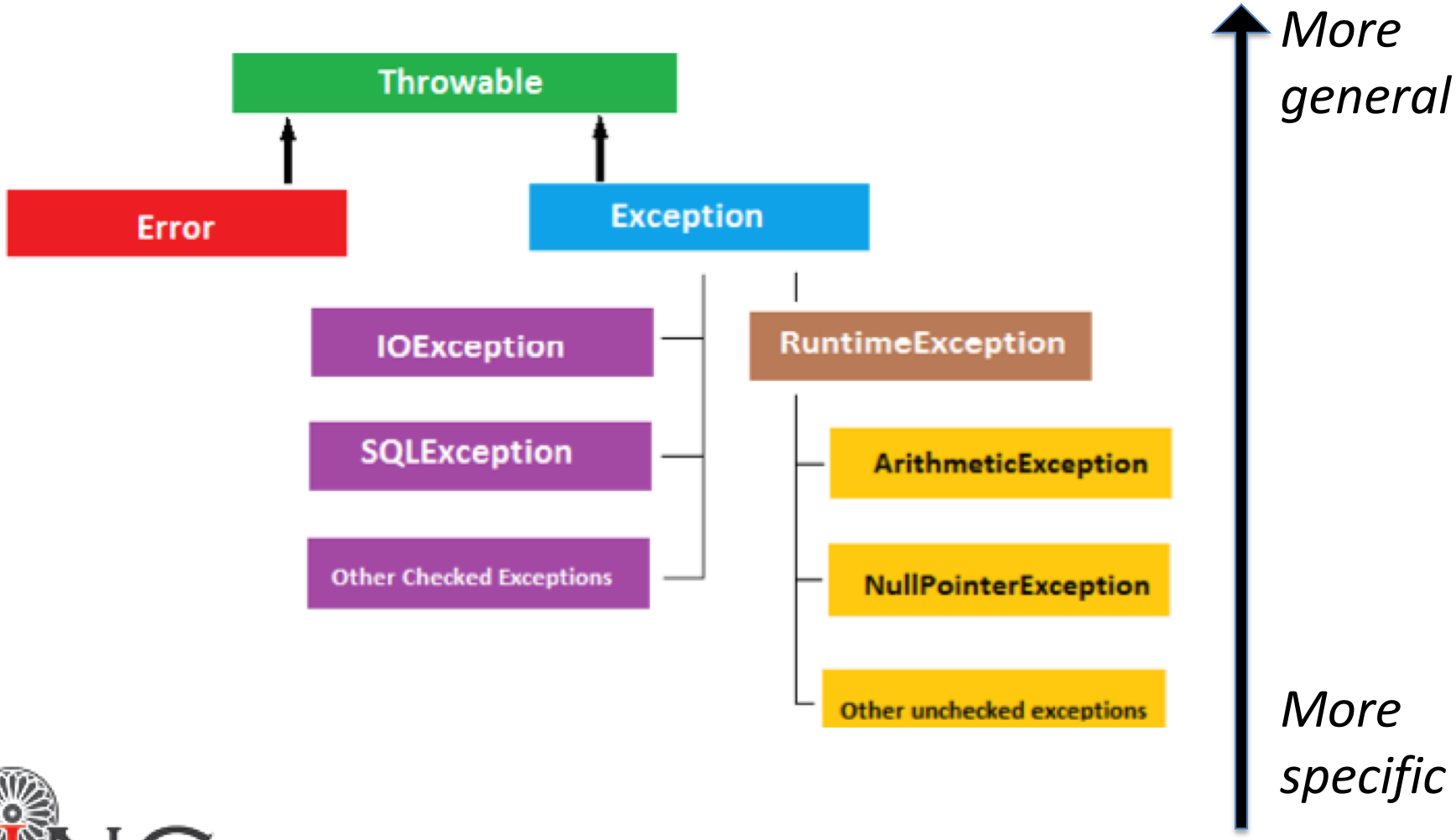
Interception (no delegation)

```
public static void main(String[] args) {
    char[] v = new char[256];
    try {
        FileReader f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```



ING

Matching Rules



ING

Complete delegation

- For enabling delegation, methods must declare exception type(s) generated within their implementation using the keyword throws (only for checked exceptions, see later)
- Exceptions can be either generated:
  - by the method itself
  - by other methods called within the method and not caught

```
public static void main(String[] args)
    throws IOException {
        char[] v = new char[256];
        FileReader f = new FileReader("test.txt");
        f.read(v);
        f.close();
}
```

Complete delegation

```
/* foo() delegates IOException */
class Dummy {
     public void foo() throws IOException {
          char[] v = new char[256];
          FileReader f = new FileReader("test.txt");
          f.read(v);
          f.close();
     }
}

/* main() intercepts IOException */
class App {
     public static void main (String args[]) {
          try {
               new Dummy().foo();
          } catch (IOException e) { /* do something */ }
     }
}
```

Complete delegation

```
class Dummy {
     public void foo() throws IOException {
          char[] v = new char[256];
          FileReader f = new FileReader("test.txt");
          f.read(v);
          f.close();
     }
}

/* main() intercepts IOException and differentiate among subclasses */
/* more specific subclasses first! */
class App {
     public static void main (String args[]) {
          try {
               new Dummy().foo();
          } catch (FileNotFoundException e) {
               System.out.println("FileNotFoundException");
          } catch (IOException e) {
               System.out.println("IOException");
          }
     }
}
```

Complete delegation

```
Class Dummy {
    public void foo() throws IOException {
        char[] v = new char[256];
        FileReader f = new FileReader("test.txt");
        f.read(v);
        f.close();
    }
}

/* main() delegates IOException as well */
Class App {
    public static void main (String args[]) throws IOException {
        new Dummy().foo();
    }
}
```

Generation (throw)

- (Eventually) Declare a user defined Exception subclass
- Throw upward a new exception object from a method marked with the throws keyword

Generation (throw)

```
public class App {
  public static void main(String[] args) {
    ArrayList<String> l = new ArrayList<String>();
    l.get(0);
  }
}
```

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
at java.util.ArrayList.rangeCheck(ArrayList.java:657)
at java.util.ArrayList.get(ArrayList.java:433)
at App.main(App.java:12)
```

Generation (throw)

```
public class ArrayList<E> extends AbstractList<E> {
…
  public E get(int index) {
    rangeCheck(index);
    return elementData(index);
  }

  private void rangeCheck(int index) {
    if (index >= size) throw new
IndexOutOfBoundsException(outOfBoundsMsg(index));
  }
  …
}
```

Partial delegation

- Methods can intercept and exception, handle it partially, and throw a new exception to be managed by callers.
- The new thrown exception can be either:
  - of the the same type (of the intercepted exception)
  - of a different type (of the intercepted exception)

Partial delegation (same exception)

```
public static void main(String[] args)
throws IOException {
    char[] v = new char[256];
    FileReader f;
    try {
        f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (IOException e) {
        // do something
        throw (new IOException());
    }
}
```

Partial delegation (custom exception)

- It is possible to define new types of exceptions if the ones provided by the system are not enough
- Developers have to define a new subclass of Exception
  - `public class MyException extends Exception {}`
- In most cases, there is no need of writing addition code
- To make subclasses completely compatible with java.lang.Exception, the Exception constructors must be explicitly written. Otherwise, only the default constructor will be available in subclasses.

Partial delegation (custom exception)

```
public class MyException extends Exception {}

public static void main(String[] args)
throws MyException {
    char[] v = new char[256];
    FileReader f;
    try {
        f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (IOException e) {
        // do something
        throw (new MyException());
    }
}
```

Partial delegation (standard exception)

```
public static void main(String[] args) {
    char[] v = new char[256];
    FileReader f;
    try {
        f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (IOException e) {
        // do something
        throw (new RuntimeException());
    }
}
```

RuntimeException is unchecked. It is frequently used for notifying the upper layers about errors that cannot be recovered. Unchecked exceptions do not required the throws mark on the method declaration

Nested exceptions

```
public static void main(String[] args) {
      char[] v = new char[256];
      FileReader f = null;
      try {
            f = new FileReader("test.txt");
            f.read(v);
            f.close();
      } catch (IOException e) {
            FileWriter fw = null;
            try {
                  fw = new FileWriter("log");
                  fw.write("Error in opening test.txt");
                  fw.close();
            } catch (IOException e1) {
                  /* the operation has failed, writing the log has failed
                   * nothing can be done anymore */
                  throw new RuntimeException();
            }
      }
}
```

Finally

- The JVM always executes the ***finally*** block regardless the outcome of try/catch. It is used for cleaning things up (e.g., closing files, database connections, etc)
- Errors inside the finally block, usually cannot be recovered. It makes sense to use RuntimeException() (unchecked)

```
public static void main(String[] args) {
      char[] v = new char[256];
      FileReader f = null;
      try {
            f = new FileReader("test.txt");
            f.read(v);
            f.close();
      } catch (IOException e) {
            // do something
      } finally {
            try {
                  if (f != null) f.close();
            } catch (IOException e) {
                  throw new
RuntimeException();
            }
      }
}
```

Exceptions and loops

- For errors affecting a single iteration (or items of a collection!), the try-catch blocks is nested in the loop. In case of exception the execution goes to the catch block and then proceed with the next iteration.

```
while(something){
    try{
        // potential exceptions
    } catch(Exception e){
        // discard iteration
    }
}
```

Exceptions and loops

- For errors compromising the whole loop, the loop is nested within the try block. In case of exception the execution goes to the catch block, thus exiting the loop.

```
try{
  while(something){
    // potential exceptions
  }
} catch(Exception e){
  // discard whole loop
}
```

Summary

- Java provides four keywords related to exceptions
  - Try
    - Contains code that may generate exceptions
  - Catch
    - Defines the error handler
  - Finally
    - Code block being executed regardless the catching phase
  - Throws
    - Mark a method as able to delegate exceptions
  - Throw
    - Generates a new exception

- There is also a class for representing exceptions
  - java.lang.Exception

Summary

- Exceptions separate error handling from functional code
  - Functional code is compact, readable and well-separated from error handling code
- Exceptions allows delegate error handling to higher levels
  - Callee might not know how to recover from an error
  - Moving upwards along the stack trace implies a larger “field of view”
  - Caller of a method can handle error in a more appropriate way than the callee

Exceptions - Best Practices

Preserve encapsulation

- Never let implementation-specific checked exceptions rise to the higher layers. For example, do not propagate SQLException from data access code to the business objects layer. Business objects layer do not need to know about SQLException. You have two options:·
  - Convert SQLException into another checked exception, if the client code is expected to recover from the exception.
  - Convert SQLException into an unchecked exception, if the client code cannot do anything about it.

Close or release resource in finally block

- Closing resources in finally block guarantees that precious and scarce resource are released properly both in case of normal and aborted execution.
- From Java 7, language has a more interesting automatic resource management or ARM blocks, which can do this for you.

Provide meaningful message on Exception

- The message of Exception is the most important place where you can point out the cause of a problem because it is the first place programmers looks upon.
- Always try to provide precise and factual information here. For example, compare:
  - new IllegalArgumentException("Incorrect argument for method”);
  - new IllegalArgumentException("Illegal value for ${argument}: ${value}”);

- The first one just says that argument is illegal or incorrect, but second one include both name of argument and its illegal value which is important to point out cause of error.
- Always follow this *Java best practice*, when writing code for handling exceptions and errors in Java.

Convert Checked Exception into RuntimeException

- This is one of the technique used to limit use of checked Exception in many of frameworks like Spring ,where most of checked Exception, which are raised from JDBC is wrapped into DataAccessException, an unchecked Exception.
- This Java best practice provides benefits, in terms of restricting specific exception into specific modules, like SQLException into DAO layer and throwing meaningful RuntimeException to client layer.

Avoid empty catch blocks

- Nothing is worse than empty catch block, because it not just hides the Errors and Exception, but also may leave your object in unusable or corrupt state.
- Empty catch block only make sense, if you absolutely sure that Exception is not going to affect object state on any ways, but still its better to log any error comes during program execution.

Use Standard Exceptions

- Using standard Exception instead of creating own Exception is better in terms of maintenance and consistency.
- Reusing standard exception makes code more readable, because most of Java developers are familiar with standard RuntimeException from JDK like, IllegalStateException, IllegalArgumentException etc. and they will immediately be able to know purpose of Exception, instead of looking out another place on code or docs to find out purpose of user defined Exceptions.

Document Exception thrown by any method

- Java provides throw and throws keyword to throw exception and in javadoc you have @throw to document possible Exception thrown by any method. This becomes increasingly important if you are writing API or public interface. With proper documentation of Exception thrown by any method you can potentially alert anyone who is using it.