



# Programming in Java – Exceptions



*Paolo Vercesi*  
*Technical Program Manager*

# How to report error conditions

*When implementing a method there are three main approaches to report error conditions*

## Error codes

*The method returns an error code.*

*E. g. 0 if everything is ok, -1 if an error happens, etc.*

## Error flags

*The method set/reset an error flag in its class to report an error condition.*

## Exception

*The method throws an Exception to interrupt its execution and to inform the caller that an exceptional condition arose.*



# Error codes

```
public class FixedSizeDisplay {  
  
    public static final int OK = 0;  
    public static final int TEXT_LENGTH_TOO_BIG = 1;  
  
    private final int size = 10;  
  
    public int display(String text) {  
        if (text.length() > size) {  
            System.out.println(text.substring(0, 10));  
            return TEXT_LENGTH_TOO_BIG;  
        } else {  
            System.out.println(text);  
            return OK;  
        }  
    }  
}
```

The method cannot return any value, it must return an **error code**

The caller must always **check the returned error code**



# Error flags

```
public class FixedSizeDisplay {  
  
    private final int size = 10;  
    private boolean error;  
  
    public void display(String text) {  
        if (text.length() > size) {  
            System.out.println(text.substring(0, 10));  
            error = true;  
        } else {  
            System.out.println(text);  
            error = false;  
        }  
    }  
  
    public boolean checkError() {  
        return error;  
    }  
}
```

The method can return any value

The caller must always **check the error status**, usually by using a method of the same class

The **PrintStream** class (the class of **System.out**) uses this approach.



# Throwing exceptions

```
public class FixedSizeDisplay {  
    private final int size = 10;  
  
    public void display(String text) throws Exception {  
        if (text.length() > size) {  
            System.out.println(text.substring(0, 10));  
            throw new Exception("Text length: " + text.length() + " exceeds display size");  
        }  
        System.out.println(text);  
    }  
}
```

*The caller shouldn't check all the invocations, but it need to deal with exceptional cases only.*

*When we throw an exception, the method execution is interrupted at the point where the exception is thrown, and the exception is propagated to the caller hierarchy until it is caught by an appropriate **try-catch** block.*



# Example

```
public class ThrowException1 {  
    public static void main(String[] args) {  
        String s = "Hello";  
  
        System.out.println(s.charAt(10));  
    }  
}
```

An exception is an **Object** that describes an exceptional condition. It brings with itself a **stack trace** and usually an **explanatory message**.

```
$ java.exe it.units.sdm.exceptions.ThrowException1  
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 10  
    at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:48)  
    at java.base/java.lang.String.charAt(String.java:1512)  
    at it.units.sdm.exceptions.ThrowException1.main(ThrowException1.java:8)
```

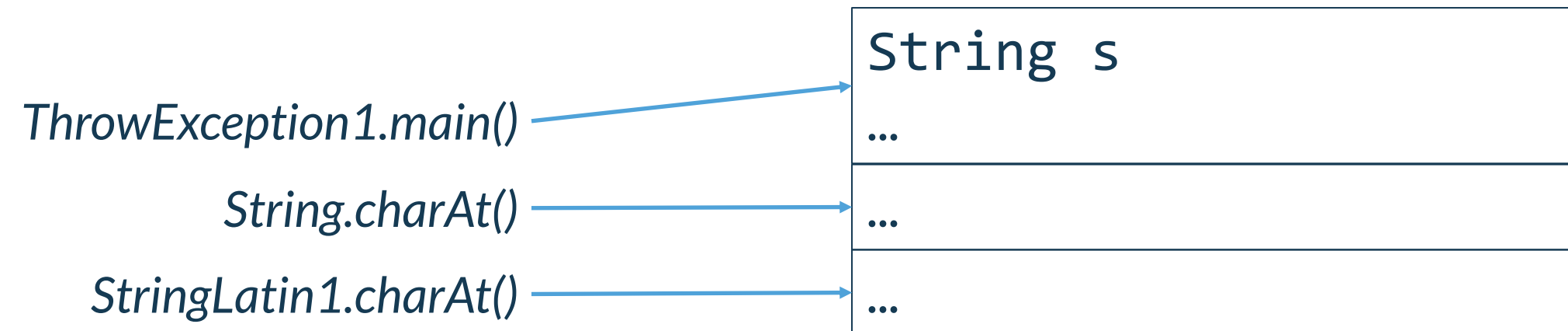




# Stack trace

*A stack trace is a report of the active frames of a given thread.*

## Stack



*The exception stack trace reports all the method in the stack by indicating the fully-qualified class name and the line number of the last executed instruction for each method.*

```
$ java.exe it.units.sdm.exceptions.ThrowException1
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 10
    at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:48)
    at java.base/java.lang.String.charAt(String.java:1512)
    at it.units.sdm.exceptions.ThrowException1.main(ThrowException1.java:8)
```

*The same information can be obtained for debugging by invoking the static method `Thread.dumpStack()`*



# Catching exceptions

```
public class ThrowException1 {  
    public static void main(String[] args) {  
        String s = "Hello";  
  
        try {  
            System.out.println(s.charAt(10));  
        } catch (StringIndexOutOfBoundsException ex) {  
            System.out.println("An error happened: " + ex.getMessage());  
        }  
    }  
}
```

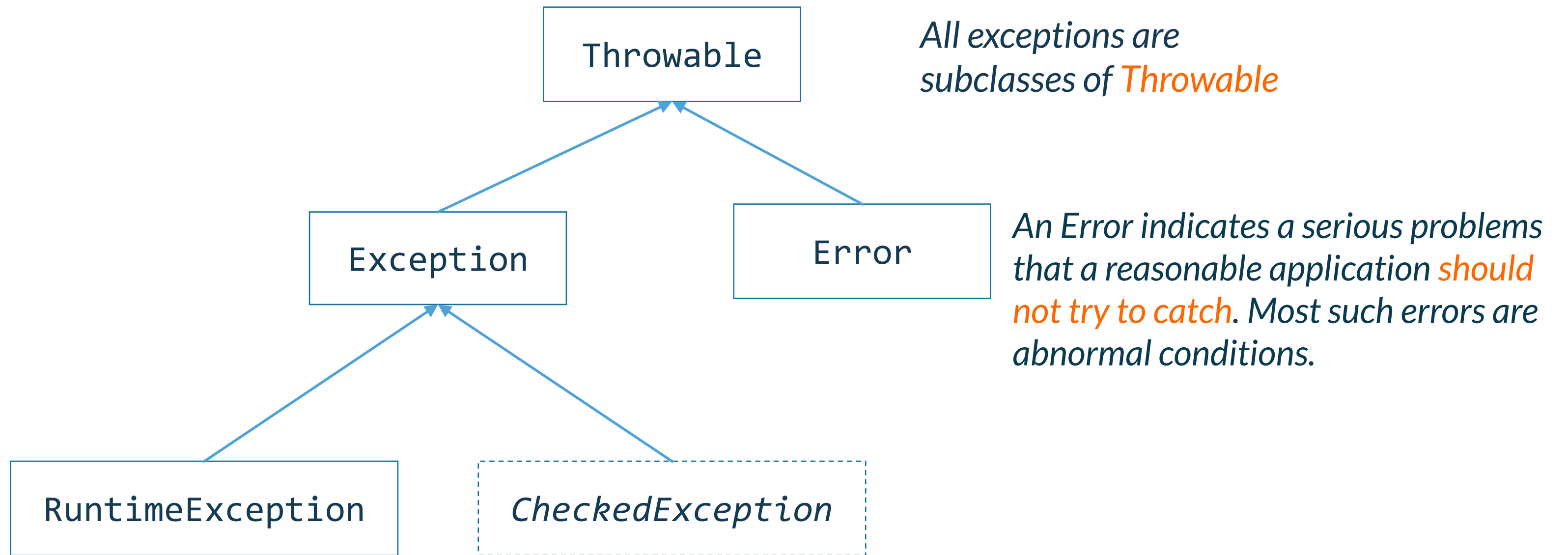
```
$ java it.units.sdm.exceptions.ThrowException1  
An error happened: String index out of range: 10
```

*Exceptions can be caught by  
try-catch blocks*





# The Exceptions hierarchy



*RuntimeException and its subclasses are **unchecked exceptions**. Unchecked exceptions do not need to be declared in a method or constructor's throws clause.*

*The **checked exception** classes are all exception classes other than the unchecked exception classes. Checked exceptions need to be declared in method or constructor's throws clause.*



# Exception types

<b>Throwable</b>	<i>Don't catch Throwable.</i>
<b>Error</b>	<i>Errors happening at the JVM level.</i>
<b>Exception</b>	<i>The base class for checked and unchecked exceptions. By Catching Exception you catch all the exceptions that can be solved by the application logic.</i>
<b>RuntimeException (unchecked exceptions)</b>	<i>Might indicate a bug in the application. Usually are not caught because the problem is at the code level. E. g. NullPointerException, ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException, etc.</i>
<b>Checked exception</b>	<i>Indicate exceptions that can be addressed by the application logic. You should catch and manage them. E. g. when you catch a FileNotFoundException you can ask the user to indicate another file.</i>



# Creating your own exceptions

```
public class FixedSizeDisplay {  
  
    private final int size = 10;  
  
    public void display(String text) throws TextTooLongException {  
        if (text.length() > size) {  
            var newText = text.substring(0, 10);  
            System.out.println(newText);  
            throw new TextTooLongException(text.length());  
        }  
        System.out.println(text);  
    }  
  
    public static class TextTooLongException extends Exception {  
  
        public TextTooLongException(int size) {  
            super("Text length: " + size + " exceeds display size");  
        }  
    }  
}
```



# Exceptions in interfaces 1/2

```
it.units.sdm.Display
```

```
public interface Display {  
  
    void display(String text) throws TextTooLongException;  
  
}
```

```
it.units.sdm.Calculator
```

```
public class Calculator {  
  
    final Display Springhare;  
    //...  
  
    Calculator(Display display) {  
        this.display = display;  
    }  
  
    void onePressed() throws TextTooLongException {  
        string += "1";  
        display.display(string);  
    }  
  
}
```

When we invoke *display(String)* we must either declare to throw the *TextTooLongException* or catch it



# Exceptions in interfaces 2/2

```
public interface Display {  
    void display(String text) throws TextTooLongException;  
}  
  
class MyDisplay implements Display {  
    @Override  
    public void display(String text) throws MyTextTooLongException {  
        //..  
    }  
}  
  
class ConsoleDisplay implements Display {  
    @Override  
    public void display(String text) {  
        //..  
    }  
}  
  
class PopupDisplay implements Display {  
    @Override  
    public void display(String text) throws Exception {  
        //..  
    }  
}
```

*The implementation can throw a subclass of the declared exception(s)*

*The implementation **can not** throw any exception*



# Handling multiple exceptions 1/2

```
public static void main(String[] args) {  
    try {  
        myMethod();  
    } catch (UserException1 ex) {  
        // do something  
    } catch (UserException2 ex) {  
        //do something  
    }  
}
```

```
static void myMethod() throws UserException1, UserException2 {  
    if (System.currentTimeMillis() % 5 == 0) {  
        throw new UserException1();  
    } else {  
        throw new UserException2();  
    }  
}
```

```
class UserException1 extends Exception {};
```

```
class UserException2 extends Exception {};
```

*All exceptions must be caught or declared in the method declaration*

*A method can declare more exceptions*



# Handling multiple exceptions 2/2

```
public static void main(String[] args) {
    try {
        myMethod();
    } catch (UserException1 | UserException2 ex) {
        // do something
    }
}

static void myMethod() throws UserException1, UserException2 {
    if (System.currentTimeMillis() % 5 == 0) {
        throw new UserException1();
    } else {
        throw new UserException2();
    }
}

class UserException1 extends Exception {};
class UserException2 extends Exception {};
```

Exceptions can be caught together. If we perform the same recover operation.

Why don't we use `catch (Exception ex)` to catch both the exceptions?





# try-catch-finally 1/3

```
public class Storage {  
    public static void main(String[] args) throws TimeoutException {  
        var storage = new Storage();  
        for (String arg : args) {  
            storage.store(arg);  
        }  
        storage.close();  
    }  
  
    void store(String text) throws TimeoutException {  
        //do something  
    }  
  
    void close() {  
        //close  
    }  
}
```

The specification of the *Storage* class says that a storage object must be *closed*, to be sure that all data has been store.

Are we satisfying the specification?



# try-catch-finally 2/3

```
public class Storage {  
  
    public static void main(String[] args) throws TimeoutException {  
        var storage = new Storage();  
        try {  
            for (String arg : args) {  
                storage.store(arg);  
            }  
        } finally {  
            storage.close();  
        }  
    }  
  
    void store(String text) throws TimeoutException {  
        //do something  
    }  
  
    void close() {  
        //do something  
    }  
}
```

The *finally block always executes when the try block exits*. This ensures that the finally block is executed even if an unexpected exception occurs.



# try-catch-finally 3/3

```
public static void main(String[] args) throws TimeoutException {
    var storage = new Storage();
    try {
        for (String arg : args) {
            int i = 0;
            while (true) {
                try {
                    storage.store(arg);
                    break;
                } catch (TimeoutException ex) {
                    if (++i == 5) {
                        throw ex;
                    }
                }
            }
        }
    } finally {
        storage.close();
    }
}
```

*try-catch-finally* blocks can be nested.

Exceptions can be *rethrown*



# Chaining exceptions

```
public interface Display {  
    void display(String text) throws DisplayException;  
}  
  
class PopupDisplay implements Display {  
  
    @Override  
    public void display(String text) throws DisplayException {  
        try {  
            double fontWidth = 100.0 / text.length();  
        } catch (ArithmeticException ex) {  
            throw new DisplayException(ex);  
        }  
    }  
}
```

```
Exception in thread "main" it.units.sdm.DisplayException: java.lang.ArithmeticException: / by zero  
    at it.units.sdm.PopupDisplay.display(FixedSizeDisplay.java:63)  
    at it.units.sdm.MainDisplay.main(FixedSizeDisplay.java:7)  
Caused by: java.lang.ArithmeticException: / by zero  
    at it.units.sdm.PopupDisplay.display(FixedSizeDisplay.java:60)  
    ... 1 more
```



# Assignment

```
public interface Collection {  
    boolean isEmpty();  
    int getSize();  
    boolean contains(String string);  
    String[] getValues();  
}  
public interface Stack extends Collection {  
    void push(String string);  
    String pop();  
    String top();  
}
```

```
public interface List extends Collection {  
    void add(String string);  
    String get(int index);  
    void insertAt(int index, String string);  
    void remove(int index);  
    int indexOf(String string);  
}
```

*Implement the Stack and List interfaces.*

*Use exceptions to report wrong usages  
e. g. illegal indexes, empty stack, etc.*



# More on exceptions

*Failure and Exceptions A Conversation with James Gosling*

<https://www.artima.com/articles/failure-and-exceptions>





Thank you!

[esteco.com](http://esteco.com)

