Exception Handling

GLTi Institutional Presentation

HSBC Technology and Services

HSBC

The world's local bank

Agenda

- Traditional Error Handling
- Exceptions
- Inheritance
  - Throwable
    - Error
    - Exception
      - RuntimeException
- Exception Handling
  - Terminology
    - try
    - catch
    - throw/throws
    - finally
- Exception Methods
  - `printStackTrace`
- Order of catch blocks

Error Handling

- Every programmer, even the most precise and thorough, is sometimes faced with his/her code behaving in a troublesome manner.
  - To prevent the application from crashing, all these errors and problems must be dealt with.
  - The traditional way of doing so is by adding **error handling code**.
    - The problem with this approach is that it overcomes the original code, it forces most of your code to deal with 'potential' errors.
    - The natural flow of the application is disturbed.
    - The return value of your methods is captivated to that mission.

Traditional Error Handling

```
void copyFiles(String src, String dest) {
   Status stat = readFile(src, data);
   if(stat == OPEN_READING_FILE_PROBLEM) {
     // ... handle the problem
     return;
   } if(stat == READING_FILE_PROBLEM) {
     // ... handle the problem
     return;
   }
   stat=writeToFile(dest,data);
   if(stat == OPEN_WRITING_FILE_PROBLEM) {
     // ... handle the problem
     return;
   } if(stat == FORMAT_PROBLEM) {
     // ... handle the problem
     return;
   } if(stat=WRRITING_PROBLEM) {
     // ... handle the problem
     return;
   } if(stat=CLOSING_FILE_PROBLEM) {
     // ... handle the problem}
     return;
   }
}
```

Exception Handling

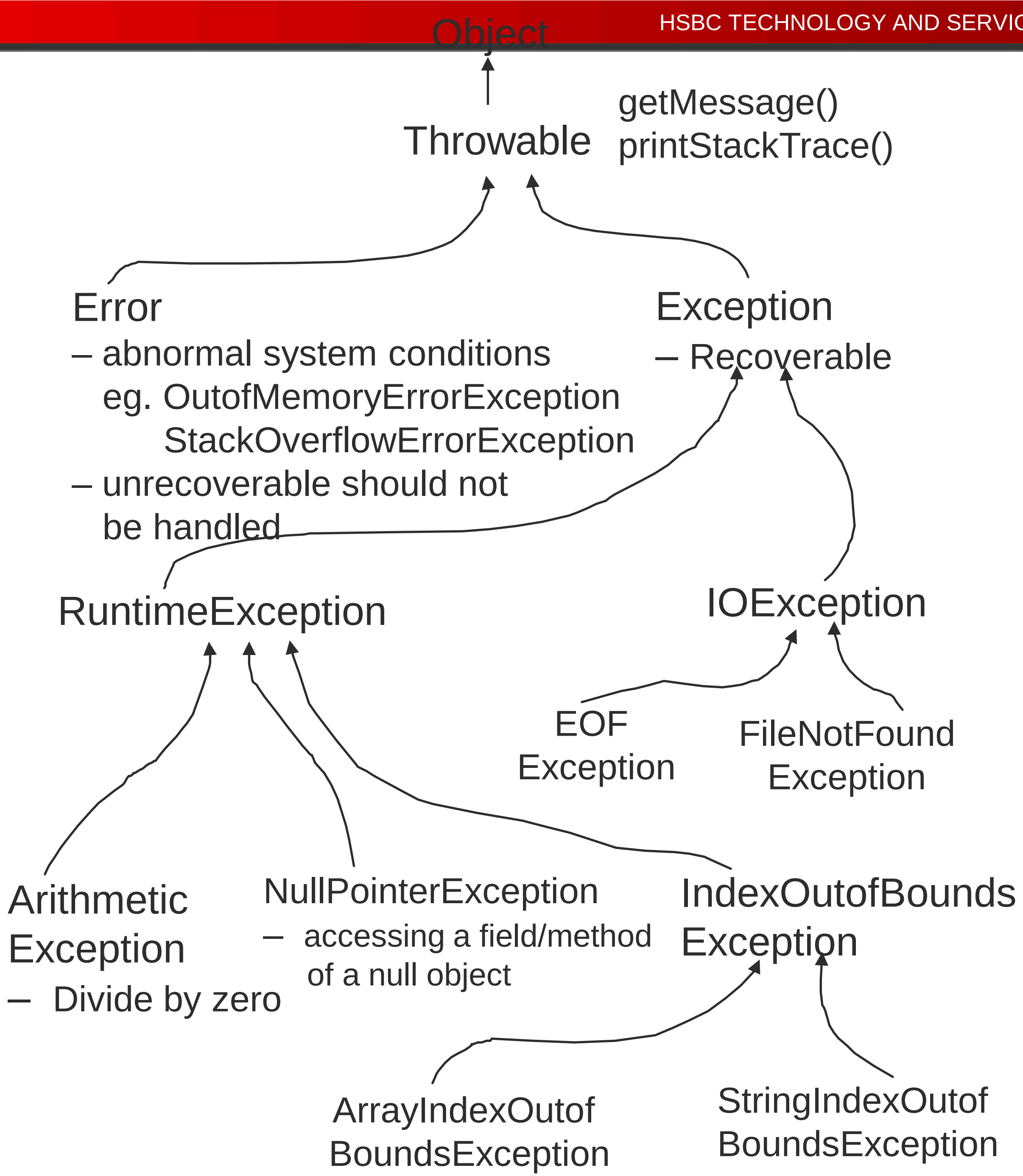
```
try
{
        code that might throw exception(s)
}
catch(Exception e)
{
        code that handles the exception(s)
}
```

When an exception is being thrown, the rest of the code (placed between the troublesome code and the end of the try block) will not be executed. Execution is passed to the `catch` block.

Exceptions

- Java provides us with a mechanism that is quite different from the traditional one.
  - Errors, problems, unexpected behavior and other misfortunes are all grouped together under the term **exceptions**.
    - An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
    - It signals that something abnormal has happened, a diversion of the normal occurrences.

Inheritance



Object

Throwable

getMessage()
printStackTrace()

Error
– abnormal system conditions
eg. OutofMemoryErrorException
StackOverflowErrorException
– unrecoverable should not be handled

Exception
– Recoverable

RuntimeException

IOException

EOF Exception

FileNotFound Exception

Arithmetic Exception
– Divide by zero

NullPointerException
– accessing a field/method of a null object

IndexOutofBounds Exception

ArrayIndexOutof BoundsException

StringIndexOutof BoundsException

A Bit of Terminology

- The creating of an exception is described using the word **throw**.
  - An exception is being thrown.
  - It is thrown, as a parameter object, to a part of the code that is responsible for dealing with it, or in other words: **catch** it.
  - This object contains information describing the nature of the problem and the state of the program when the exception occurred.

Handling Exceptions

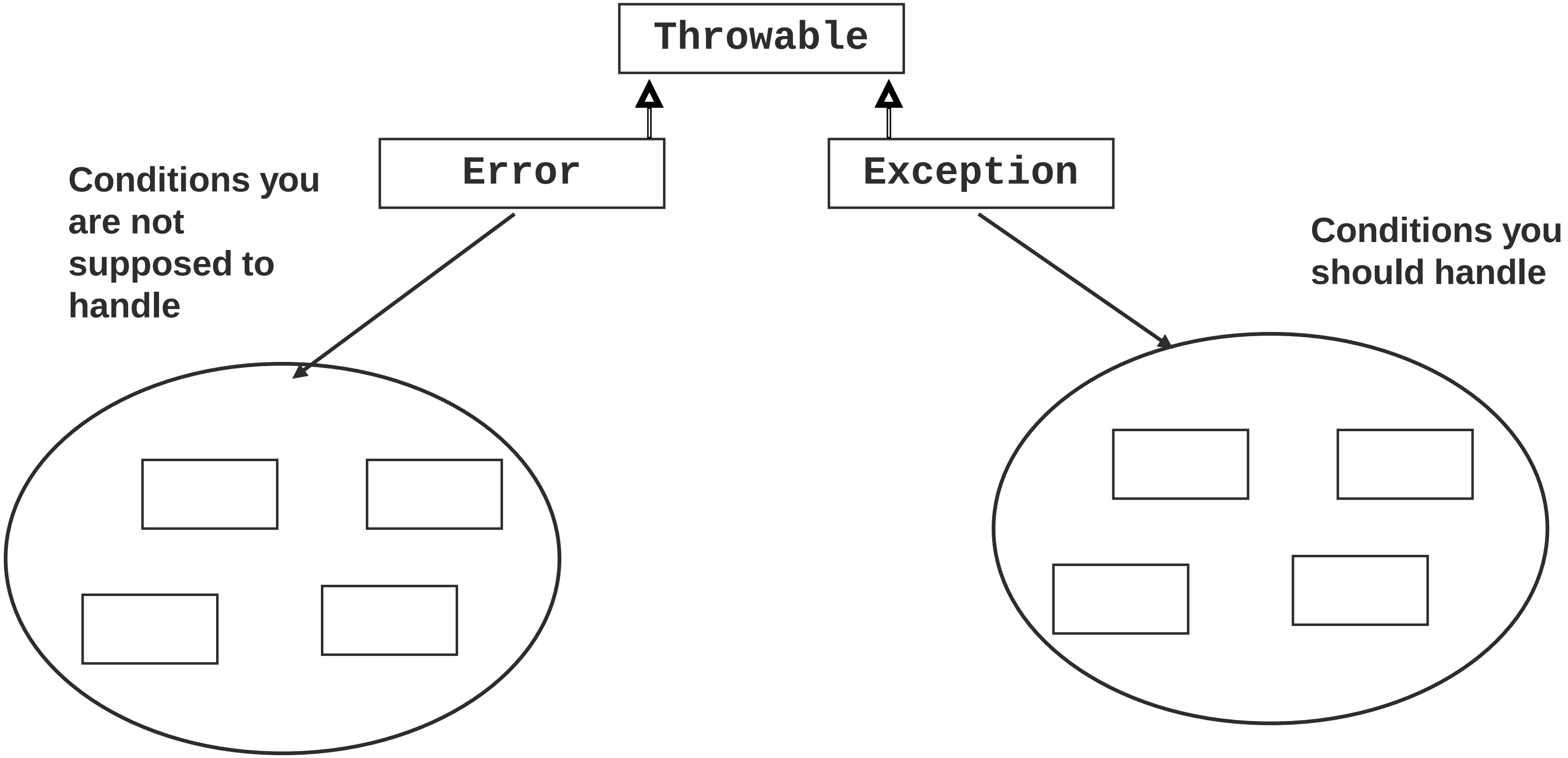
- Dealing with exceptions can be achieved using 3 types of code blocks:
  - **`try`**, **`catch`** and **`finally`**.
- The **`try`** block is used to enclose the part of the code that might raise an exception.
  - This block must always be followed by at least one **`catch`** block.
- The **`catch`** block is responsible for catching the exception that was thrown along the **`try`** block.
  - It is possible to create a **`catch`** block to every type of exception that might be thrown of the **`try`** block, or use the hierarchy mechanism to group together some related exceptions.

A Little More About `catch`

- There is quite a lot you can do to handle the exception that was just tossed to your lap:
  - Show a warning message and return.
  - Exit the program.
  - Re-throw the exception.
  - Throw another exception.
  - Try to solve the problem.

Exception Is an Object

- A Java exception is an object.
  - It is a sub class of the standard class **`Throwable`**, or to be more accurate, a sub class of either **`Error`** or **`Exception`** that derive from **`Throwable`**.


Throwable
Error
Exception
Conditions you are not supposed to handle
Conditions you should handle

Errors

- Luckily, you are not supposed to deal with exceptions that are defined by the class **`Error`** or one of its sub classes.
  - The reason for this exemption is that they occur in situations that you are not able to do anything about.
    - **`ThreadDeath`** is thrown when you deliberately killed a thread. Attempting to catch it will interfere with the thread's proper destruction.
    - **`LinkageError`** Reports errors arising during the linkage process.
    - **`VirtualMachineError`** Though very serious, is completely untreatable through your application.

Exceptions

- We can roughly divide all the exceptions that extend the **`Exception`** class into two categories:
  - **Run time exceptions**: usually mistakes inside your code.
    - The complier allows you to ignore them, mostly because there is usually very little you can do to recover from them.
    - Nevertheless you can, if you like, react to them.
    - Examples for **`RunTimeException`** sub classes:
      - **`ArithmeticException, ArrayStoreException, CannotRedoException, CannotUndoException, ClassCastException, EmptyStackException, IndexOutOfBoundsException.`**
  - **Other kind of exceptions:**
    - As apposed to **`runtimeException`**, they must be dealt with. Failing to do so will result in a compiling error.

Available methods

- You can retrieve information regarding a thrown Exception (or Error) using the `throwable` methods:
  - **`public String getMessage()`**
    - Description of the problem.

    e.g. `/ by zero`
  - **`public String toString()`**
    - The name of the actual class of this object **+ `getMessage()`**output.

    e.g. `java.lang.ArithmeticException: / by zero`

`printStackTrace` Methods

- **`public void printStackTrace()`**
    - Prints an output like the one on the last slide to the standard error stream.
- **`public void printStackTrace(PrintStream s)`**
    - Print the output to the specified print stream.
- **`public void printStackTrace(PrintWriter s)`**
    - Prints the output to the specified print writer.
- The last method is useful when an application is re-throwing an error or exception:
    - **`public Throwable fillInStackTrace()`**

printStackTrace

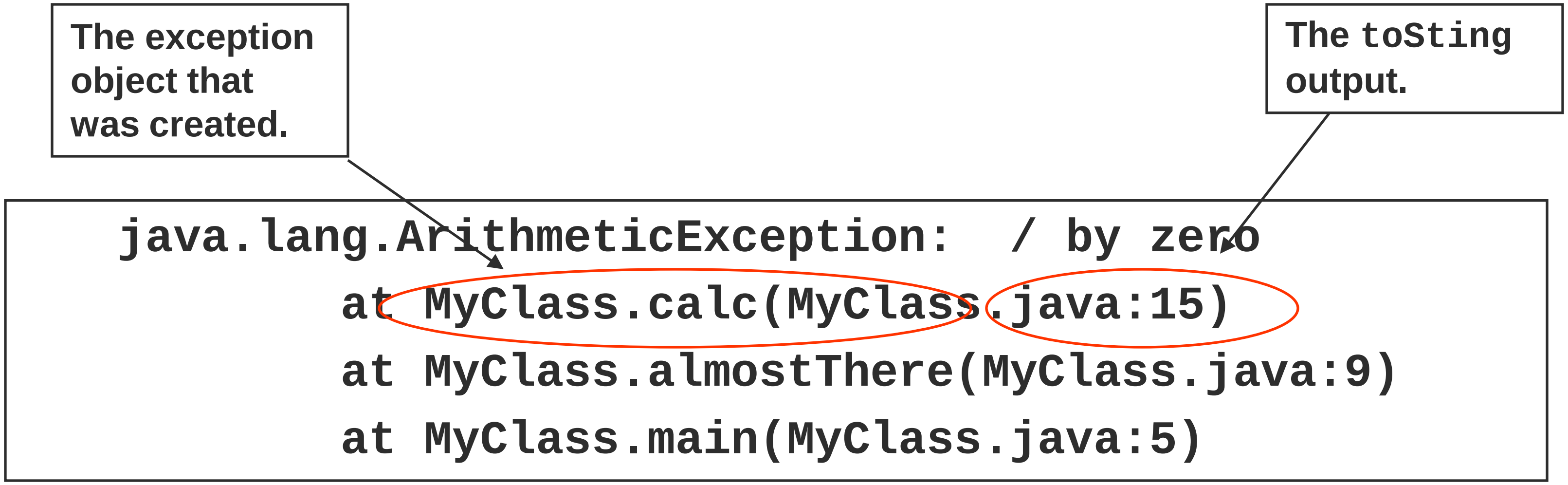
- The purpose of the **`printStackTrace`** methods is to supply the user with information about the trace that the exception has gone through inside the application.

```
class MyClass
{
     public static void main(String[] argv) {
         almostThere(0);
     }
     static void almostThere(int a) {
         calc(a);
     }

     static void calc(int num) {
         int b=1;
         try {
             System.out.println(b/num);
         } catch (Exception e){
             e.printStackTrace();
         }
     }
 }
```

The output is displayed on the next slide

printStackTrace output



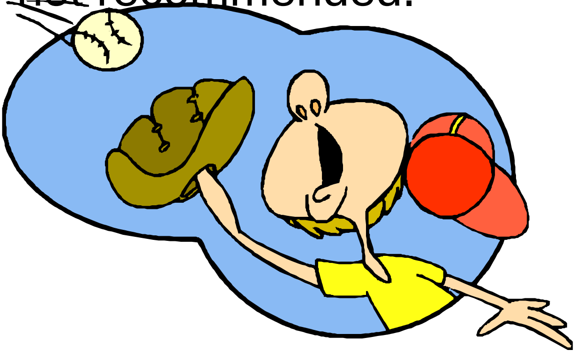
The exception object that was created.

The toSting output.

```
    java.lang.ArithmeticException:  / by zero
            at MyClass.calc(MyClass.java:15)
            at MyClass.almostThere(MyClass.java:9)
            at MyClass.main(MyClass.java:5)
```

Multiple `catch` Blocks

- When your program is prone to raise multiple kinds of exceptions, you are advised to equip it with corresponding **`catch`** blocks.
- It is possible to supply a different **`catch`** block for every exception that might be thrown, or decrease their number by creating a **`catch`** block that deals with a super class of those expected exceptions.
  - An extreme condition will be to catch the father of all exception: the **`Exception`** object itself, this is of course not recommended.

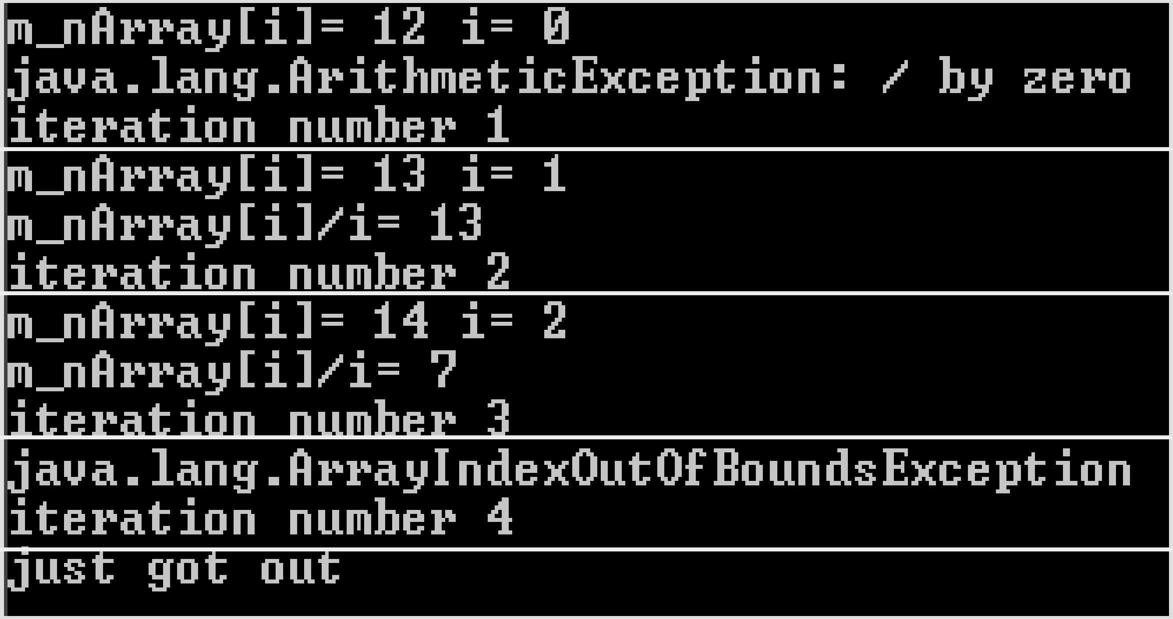


Making an Exception

```
class MyException
{
 public static void main(String args[])
 {
  int m_nArray[] = {12,13,14};
  for (int i=0;i<=m_nArray.length;i++){
   try {
    System.out.println("i= "+ i +" m_nArray[i]= " + m_nArray[i]+"
              m_nArray[i]/i= " + m_nArray[i]/i);
   }
   catch(ArithmeticException a){
    System.out.println(a);
   }
    catch(ArrayIndexOutOfBoundsException e){
    System.out.println(e);
   }
   System.out.println("iteration number " + (i+1));
  }
  System.out.println("just got out");
 }
}
```

The Output

First iteration: division by zero



```
m_nArray[i]= 12 i= 0
java.lang.ArithmeticException: / by zero
iteration number 1
m_nArray[i]= 13 i= 1
m_nArray[i]/i= 13
iteration number 2
m_nArray[i]= 14 i= 2
m_nArray[i]/i= 7
iteration number 3
java.lang.ArrayIndexOutOfBoundsException
iteration number 4
just got out
```

Fourth iteration: exceeding the array

After the `catch` block

How Does It Work?

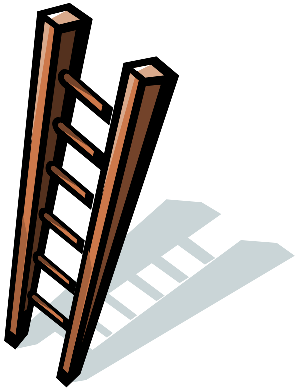
- The program performs a loop:
  - On the first iteration we attempt to make a division by zero, so an exception is thrown and caught by the **`catch`** block who deals with **`ArithmeticException`**. (line 13)
  - The second and third iterations go smoothly.
  - On the fourth iteration we are exceeding the bounds of the array and create an exception that is being caught by the **`catch`** block who deals with **`ArrayIndexOutOfBoundsException`** (line 17).
  - Line 21 is located just after the **`catch`** block and is executed on every iteration no matter whether an exception was thrown

Order of `catch` Blocks

- You should carefully consider the order of the **`catch`** blocks.
  - When using exceptions of the same hierarchy, you should place them in order: a sub class will come before its superclass.
  - Except for it being very reasonable, failing to do this will result in a compile error.

An Escapee

- So far, we wrapped the exception-prone methods in a **`try`** block and followed it with a **`catch`** block.
- As we will soon see, it is possible not to catch the exception in the method which had it thrown, but rather upwards in the calling methods hierarchy.
  - An exception that is not caught will propagate its way upwards until it is either caught or reaches the entrance point of the program.
    - If this is the case, the program will end with an explanatory message.



Climbing Up

- There is a significant difference between the way a **`RuntimeException`** and a regular exception climb up the methods ladder.
  - A **`RuntimeException`** can be caught (or not) in every one of the methods who invoked the one that threw it.
  - A regular exception needs a little more help:
    - A method must indicate that it does not handle an exception by itself, rather it throws it upwards – it does so by adding the keyword **`throws`** and the type of the exception:

      ```
      public void myMethod() throws IOException
      {
      }
      ```

Propagating Example

```
import java.io.*;
class MyThrow
{
   public static void main(String args[]){
     MyThrow my = new MyThrow();
   }
   public MyThrow(){
      catchIt();
   }

   public void catchIt(){
     try
     {
       makeCalc(0);
     }
     catch(ArithmeticException a)
     {
       System.out.println(a);
     }
     catch(IOException io)
     {
       System.out.println(io);
     }
   }
```

Using throws

```
public void makeCalc(int div) throws IOException
  {
     divide(div);
  }
  // This may throws an IOException:
  public void divide(int y) throws IOException
  {
     int x = 9;
     System.in.read();
     System.out.println(x/y);
  }
}
```

The `System.in.read()` method stops the program until the enter key is pressed.

It is used here to demonstrate the propagating of an `IOException`

Nesting Blocks

- A **`try`** and **`catch`** pair may appear inside another try-catch block, which in turn may also be located inside such a pair.
- Nesting try-catch block enables us to catch exceptions that were thrown inside the inner blocks by the outer ones.

```
try
{
  try
  {
    try
    {
    }
    catch(exceptionC C)
    {
    }
  }
  catch(exceptionB B)
  {
  }
}
catch(exceptionA A)
{
}
```

Nesting Example

```
import java.io.*;
class Nesting2
{
  public static void main(String args[])
  {
    try
    {
      try
      {
        int x=8,y=9;
        System.in.read();
        System.out.println(x/y);
      }
      catch(ArithmeticException math)
      { }
    }
    catch(IOException e)
    {
      System.out.println(e);
    }
  }
}
```

The `finally` Block

- The **`finally`** block is optional. When included, it is placed after the **`catch`** block(s).
- The code in this block is always executed whether an exception was thrown or not.
- It is recommended to use it to perform some cleaning and finishing chores.
  - Closing files, killing threads, closing sockets and more.

} finally example

```
import java.io.*;
class FinallyBlock
{
  public static void main(String args[])  {
     FileInputStream fis;
     try {
              fis = new FileInputStream("finallyblock.java");
              try {
                      fis.read();
              }
              catch(IOException e){
                      System.out.println("Read error occurred");
              }
              finally {
                      fis.close(); // Always close the file !
              }
     }
     catch (IOException e) {
              System.out.println("Open/Close error occurred");
     }
  }
}
```

Creating Your Own Exceptions

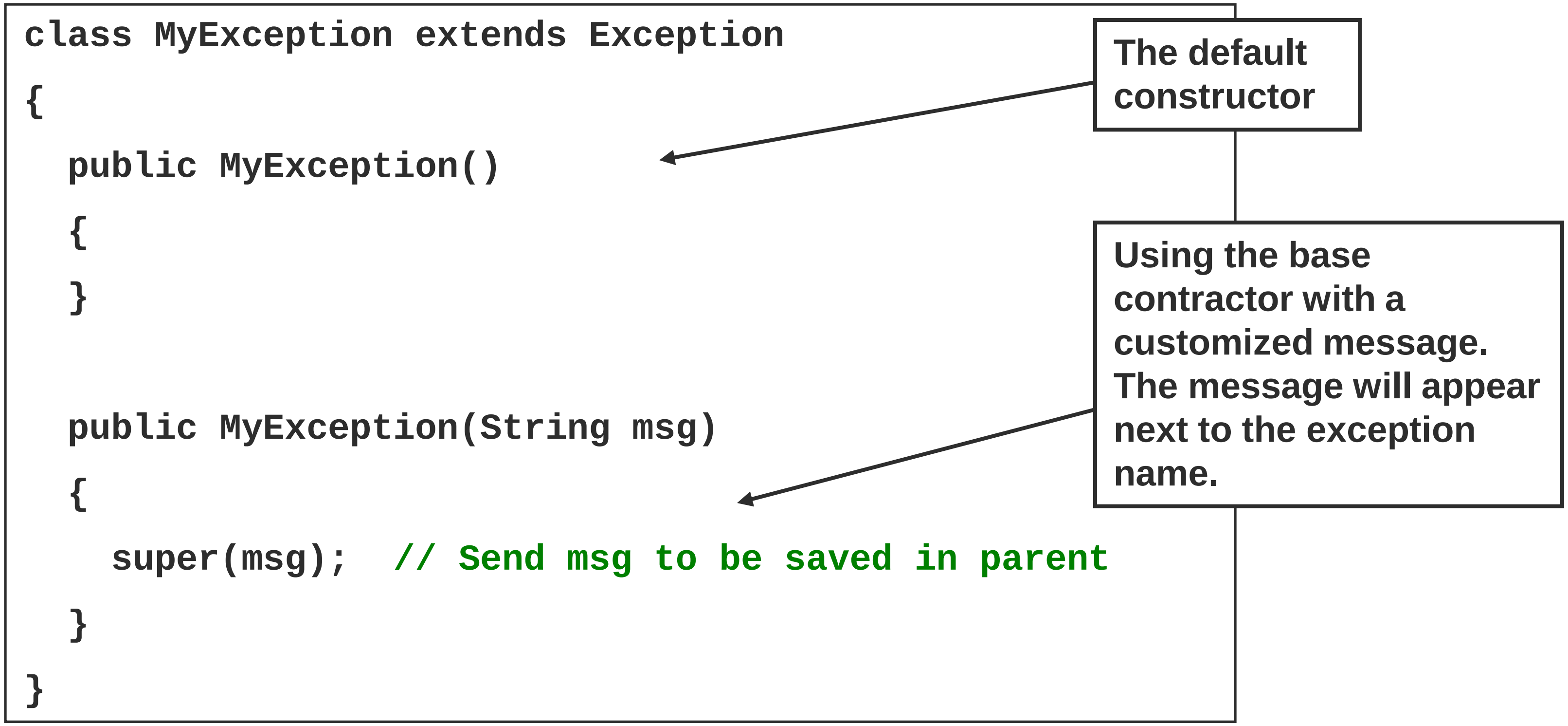
- There are plenty of exception that were carefully designed and created and are ready to use.
  - So, why bother creating some more?
- Well, there are couple of reasons.
  - When creating your own exception, you can refine its output and treat its invoker with some helpful information.
  - There are special occurrences in your application that deserve their special exception class.

So, How Do I Do It?

- The first step is creating a new class that extends **`Exception`**.
  - You can, of course, extend every sub class of **`throwable`**, but it is recommended to stick to **`Exception`**.
- The method that will throw that exception will:
  - Add the keyword **`throws`** and the exception type to the method's signature.
  - Throw a new object of that type.

```
public void myMethod() throws MyException
 {
   if . . . . . .
     throw new MyException();
 }
```

User Defined Exception



```
class MyException extends Exception
{
  public MyException()
  {
  }

  public MyException(String msg)
  {
    super(msg);  // Send msg to be saved in parent
  }
}
```

The default constructor

Using the base contractor with a customized message. The message will appear next to the exception name.

User Defined Exception - Example

```
class UnAuthorizedPersonException extends Exception
{
   private String Password,ID;
   public UnAuthorizedPersonException()   { }
   public UnAuthorizedPersonException(String str)
   {
     super(str);
   }
   public UnAuthorizedPersonException(String str,String ID,String password)
   {
     super(str);
     this.ID = ID;
     this.Password = password;
   }
   public String getIntruder()
   {
     return (" ID=" + ID + " Password=" + Password);
   }
}
```

User Defined Exception - Example

```
class Users
{
  public static void checkPermit(String ID, String password) throws
         UnAuthorizedPersonException
  {
     if (!(ID.equals(“irfan") && password.equals(“pass")))
        throw new UnAuthorizedPersonException("Un authorized access
        attempt by",ID,password);
  }
}
```

A new exception is created and thrown

User Defined Exception - Example

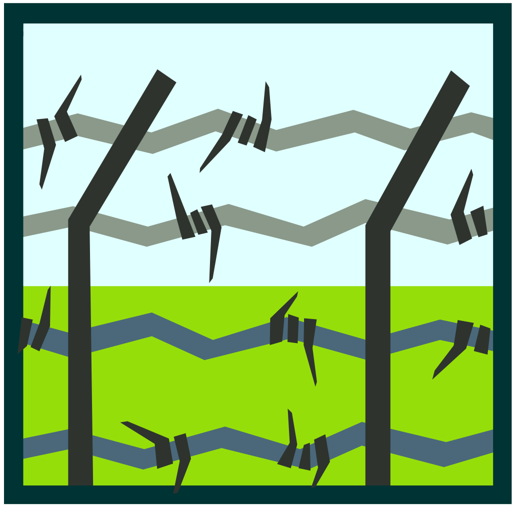
- The following small program (3 classes) demonstrates how you can create your special exception and **`throw`** it when necessary.

```
class Login
{
  public static void main(String args[])  {
    Login login = new Login(args);
  }

  public Login(String details[])  {
     if (details.length !=2)
        System.exit(-1);
     try
     {
       Users.checkPermit(details[0],details[1]);
     }
     catch(UnAuthorizedPersonException u)
     {
        System.out.println(u + u.getIntruder());
     }
  }
}
```

Exception Creation Exercise

- The new exception that you are about to create, **`IllegalArrayIndexException`** will be thrown each time that the **`RuntimeException`**, **`IndexOutOfBoundsException`** **is thrown.**
- **`IllegalArrayIndexException`** will supply the calling method with the erroneous index and an explanatory message.



Summary

- Throwable
  - Error
  - Exception
    - RuntimeException
- try
- catch
- throw/throws
- finally
- Exception Methods
  - `printStackTrace`
- Order of catch blocks

Exception Summary

- The exceptions mechanism is designed to treat various types of erroneous situations.
- Exceptions are objects that are sub classes of class **`Throwable`**.
  - You can define your own exception by extending class **`Exception`**.
- **`RuntimeException`** and **`Error`** exceptions may not be dealt with inside your code.
  - Other types of exceptions must be dealt with.
- Dealing with an exception is done by either using the **`try`** and **`catch`** blocks or by throwing it upwards.
  - The **`catch`** block can treat the problem or re-**`throw`** it to the calling method.
- To throw an exception use the **`throw`** keyword.

HSBC

The world’s local bank

About HSBC Technology and Services

HSBC Technology and Services (HTS) is a pivotal part of the Group and seamlessly integrates technology platforms and operations with an aim to re-define customer experience and drive down unit cost of production. Its solutions connect people, devices and networks across the globe and combine domain expertise, process skills and technology to deliver unparalleled business value, thereby enabling HSBC to stay ahead of competition by addressing market changes quickly and developing profitable customer relationships.

Presenter’s Contact Details:
Name: Deepak Ratnani
Role: Team Leader
Direct: + 91-20 -66423608
Email: DeepakRatnani@hsbc.co.in

Restricted for company use only