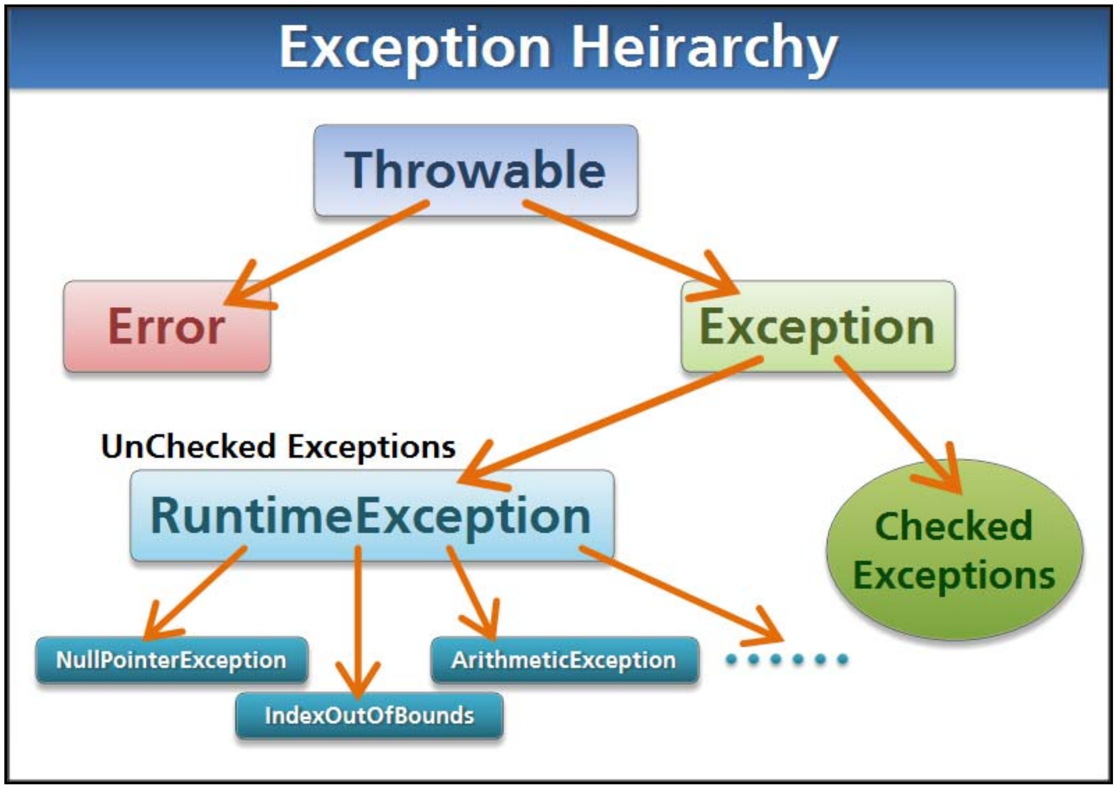
Exception Heirarchy and Strategy

It is helpful to see the Heirarchy of the Exception class structure. In this video, I intend to pursue several strategies for error handling using exceptions. Knowing the class Heirarchy will assist you in making your strategy work for you.



Throwable - Mother of all exceptions for a class structure point of view

Error - Errors related to your system. For example, your system is out of memory. In general there isn't much you can do about this errors from an application point of view. I usually don't worry about them because there isn't anything I can do about them. It's like worrying about the fact that we could be hit with an Asteroid which would wipe out civilization as we know about it.

Exception - Everything that you normally have to deal with as a programmer falls under the Exception umbrella

RuntimeException - All **`"unchecked" exceptions`** fall under this umbrella. The compiler won't force your program to worry about these errors. There are many exceptions derived from RuntimeException (many of you have encountered some of these by now if you make errors) such as :
`NullPointerException`
`IndexOutOfBounds`
`ArithmeticException`
`....`

Checked Exceptions - Any class not derived from RuntimeException, but falls under the Exception umbrella. I will give you a few examples of these in this course, but you will encounter them more in future advanced classes.

Exception Handling strategies:

I will go through some of the possible strategies for handling errrors.

Note that **`often we will use a combination`** of these philosophies. However, for the sake of clarifying the options, I have detailed some very distinct options for handling exceptions.

I don't want to rate any of these strategies as being better than the other. Depending on the situation, I might pick any of the following options as my best choice for the set of conditions I am currently operating in.

"Evel Knievel" programming style

Errors possible:

1. No parameters
2. Only one parameter
3. bad parameter - illegal character

4. bad parameter - index < 0 OR index >= 10
5. One of the data[] entries could be zero

```
package evel_knievel;

public class ExceptionHeirarchy {

    int[] data={1,2,3,4,5,6,7,8,9,10};

    int[] getIndices(String[] args)
    {
        int[] indices= new int[2];
        int element;

        for (element=0; element <=1; element ++)
            indices[element]= Integer.parseInt(args[element]);

        return indices;
    }

    int compute(String[] args)
    {
        int[] indices = getIndices(args);
        int total = 0;
        for (int i=indices[0]; i <= indices[1]; i++)
        {
            total = total + 100/data[i];
        }
        return total;
    }


    public static void main(String[] args) {
        ExceptionHeirarchy eh = new ExceptionHeirarchy();

        int result = 0;

        result = eh.compute(args);

        System.out.println("Program finishes, value="+result);
    }

}
```

Global Handling

This is for the person who wants to keep things going but doesn't want to spend a lot of time analyzing the possible errors and recovery strategies available.

Repeat some of the above errors to see how they are handled

```
package global_handling;

public class ExceptionHeirarchy {

    int[] data={1,2,3,4,5,0,7,8,9,10};

    int[] getIndices(String[] args)
    {
        int[] indices= new int[2];
        int element;

        for (element=0; element <=1; element ++)
            indices[element]= Integer.parseInt(args[element]);

        return indices;
    }

    int compute(String[] args)
    {
        int[] indices = getIndices(args);
        int total = 0;
        for (int i=indices[0]; i <= indices[1]; i++)
        {
            total = total + 100/data[i];
        }
        return total;
    }


    public static void main(String[] args) {
        ExceptionHeirarchy eh = new ExceptionHeirarchy();

        int result = 0;
        try
        {
            result = eh.compute(args);
        }

        catch(RuntimeException e)
        {
            System.out.println(e);
            e.printStackTrace(System.out);
            //e.printStackTrace();
        }


        System.out.println("Program finishes, value="+result);
    }

}
```

Throwable features

Take a look at the features of the Throwable class. All exceptions inherit these features.

- getMessage()
- toString()
- printStackTrace(System.out)
- printStackTrace()

Other global handling options

Note that you can put in other class combinations to catch these errors as long as you understand the heirarchy. Throwable will catch everything possible.

```
 /*
      // Note that more specific exceptions must precede more general exceptions

        catch (ArrayIndexOutOfBoundsException e)
        {

        }
        catch (ArithmeticException e)
        {

        }
        catch (NumberFormatException e)
        {

        }
        */
        // catch (Exception e)
        //catch (Throwable e)
```

local handling better errors

- Suppose that you want to collect as much detailed information as you can get, but you are uncomfortable(indecisive) making decisions on a local level.
- So instead you pass the decision process to a higher level authority by throwing your own Exception.
- Try out the possible errors to see how they are handled.

```
package local_handling_better_errors;

enum ParmErrorType {BAD_DATA, PARM_INDEX_OUT_OF_BOUNDS};
enum DataErrorType {DIVIDE_BY_ZERO, DATA_INDEX_OUT_OF_BOUNDS};

class DataError extends RuntimeException // I can choose to extend other Exception classes if I wish.
{
    int whichIndex=0;
    DataErrorType errorType;


    DataError(DataErrorType errorType, int index)
    {
        whichIndex = index;
        this.errorType = errorType;
    }

    public String toString()
    {
        if (errorType == DataErrorType.DATA_INDEX_OUT_OF_BOUNDS)
            return "Data access index out of bounds: " + whichIndex;
        else
            return "Divide by zero at: " + whichIndex;
    }
}

class BadParameter extends RuntimeException
{

    int whichIndex=0;
    String badData="";
    ParmErrorType errorType;


    BadParameter(ParmErrorType errorType, int index)
    {
        this(errorType, index, "");
    }

    BadParameter(ParmErrorType errorType, int index, String badData)
    {
        this.errorType = errorType;
        whichIndex = index;
        this.badData = badData;
    }
    public String toString()
    {
        if (errorType == ParmErrorType.PARM_INDEX_OUT_OF_BOUNDS)
            return "Missing Parameter: " + whichIndex;
        else
            return "Bad Parameter value: "+ badData;
    }

}

public class ExceptionHeirarchy {

    int[] data={1,2,3,4,5,0,7,8,9,10};

    int[] getIndices(String[] args)
    {
        int[] indices= new int[2];
        int element=0;

        try
        {
            for (element=0; element <=1; element ++)
                indices[element]= Integer.parseInt(args[element]);
        }
        catch (NumberFormatException e)
        {
            throw new BadParameter(ParmErrorType.BAD_DATA, element, args[element]);
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            throw new BadParameter(ParmErrorType.PARM_INDEX_OUT_OF_BOUNDS, element);
        }
        return indices;
    }

    int compute(String[] args)
    {
        int[] indices = getIndices(args);
        int total = 0;
        for (int i=indices[0]; i <= indices[1]; i++)
        {
            try
            {
                total = total + 100/data[i];
            }
            catch (ArithmeticException e)
            {
                throw new DataError(DataErrorType.DIVIDE_BY_ZERO, i);
            }
            catch (ArrayIndexOutOfBoundsException e)
            {
                throw new DataError(DataErrorType.DATA_INDEX_OUT_OF_BOUNDS, i);
            }
        }
        return total;
    }


    public static void main(String[] args) {
        ExceptionHeirarchy eh = new ExceptionHeirarchy();

        int result = 0;
        try
        {
            result = eh.compute(args);
        }

        catch (DataError e)
        {
            System.out.println("Our DataError: " + e.toString());
            System.out.println("whichIndex=" + e.whichIndex);
        }
        catch (BadParameter e)
        {
            System.out.println("Our BadParameter: " + e.toString());
            System.out.println("whichIndex=" + e.whichIndex);
            System.out.println("badData="+ e.badData);
        }
        catch(Exception e)
        {
            System.out.println("Shouldn't get here");
            System.out.println(e);
            e.printStackTrace(System.out);
            //e.printStackTrace();
        }


        System.out.println("Program finishes, value="+result);
    }

}
```

Local handling and fixing of problems

So you like to catch problems at their earliest points and you are willing to be decisive and make appropriate decisions to keep the show rolling. You will print out a few little messages informing your user about the decisions you made.

```
package local_handling_fixit;


public class ExceptionHeirarchy {

    int[] data={1,2,3,4,5,0,7,8,9,10};

    int[] getIndices(String[] args)
    {
        int[] indices= {-1, -1};
        int element=0;

        try
        {
            for (element=0; element <=1; element ++)
                indices[element]= Integer.parseInt(args[element]);
        }
        catch (NumberFormatException e)
        {
            String which = "first";
            if (element == 1)
                which = "second";
            System.out.println(which + " parameter is bad ... we will default to reasonable choices. ");
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            if (element ==0)
                System.out.println("Missing all parameters ... we will default these and move on");
            else
                System.out.println("Missing last parameter ... we will default this and move on");
        }
// Fixup bad indices
        if (indices[0] == -1)
            indices[0] = 0;
        if (indices[1] == -1)
            indices[1] = data.length-1;
        return indices;
    }

    int compute(String[] args)
    {
        int[] indices = getIndices(args);
        int total = 0;
        for (int i=indices[0]; i <= indices[1]; i++)
        {
            try
            {
                total = total + 100/data[i];
            }
            catch (ArithmeticException e)
            {
                System.out.println("Divide by zero at index: " + i + " skipping data and moving on.");
                // Nothing needed to skip this iteration
            }
            catch (ArrayIndexOutOfBoundsException e)
            {
                System.out.println("Accessing non-existent data  at index: " + i +
                        " skipping data and moving on.");
                // Nothing needed to skip this iteration
            }
        }
        return total;
    }


    public static void main(String[] args) {
        ExceptionHeirarchy eh = new ExceptionHeirarchy();

        int result = 0;
        try
        {
            result = eh.compute(args);
        }


        catch(Exception e) // Just in case we forgot something.
        {
            System.out.println("Shouldn't get here");
            System.out.println(e);
            e.printStackTrace(System.out);
            //e.printStackTrace();
        }


        System.out.println("Program finishes, value="+result);
    }
```

```
}
```

Last Updated: July 28, 2014 10:26 AM