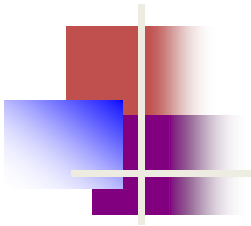
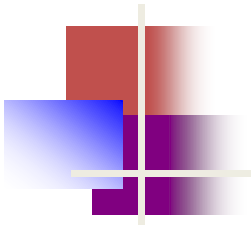


Exceptions



Introduction

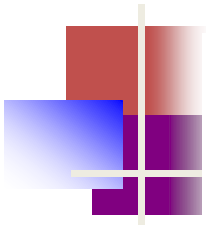
- A program can be written assuming that nothing unusual or incorrect will happen.
- This is normally overly optimistic.
  - The file containing the needed information will always exist.
  - REALLY!

What is an exception?

- An ***exception*** is an error condition that changes the normal flow of control in a program

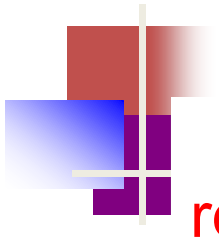
- Without handling ***exception***s (pseudocode)

```
read-file {
        openTheFile;
        determine its size;
        allocate that much memory;
        closeTheFile;
 }
```



One way to handle Exceptions: If-else

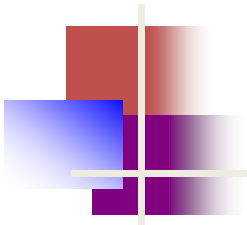
```
openFiles;
 if (theFilesOpen) {
     determine the lenth of the file;
     if (gotTheFileLength){
       allocate that much memory;
       if (gotEnoughMemory) {
            read the file into memory;
            if (readFailed) errorCode=-1;
            else errorCode=-2;
       }else  errorCode=-3;
     }else errorCode=-4 ;
  }else errorCode=-5;
```



```
read-File;
{ try {
     openTheFile;
     determine its size;
     allocate that much memory;
     closeTheFile;
  }catch(fileopenFailed) { dosomething; }
   catch(sizeDetermineFailed) {dosomething;}
   catch(memoryAllocateFailed){ dosomething;}
   catch(readFailed){ dosomething;}
   catch(fileCloseFailed) { dosomething; }
}
```

Exceptions

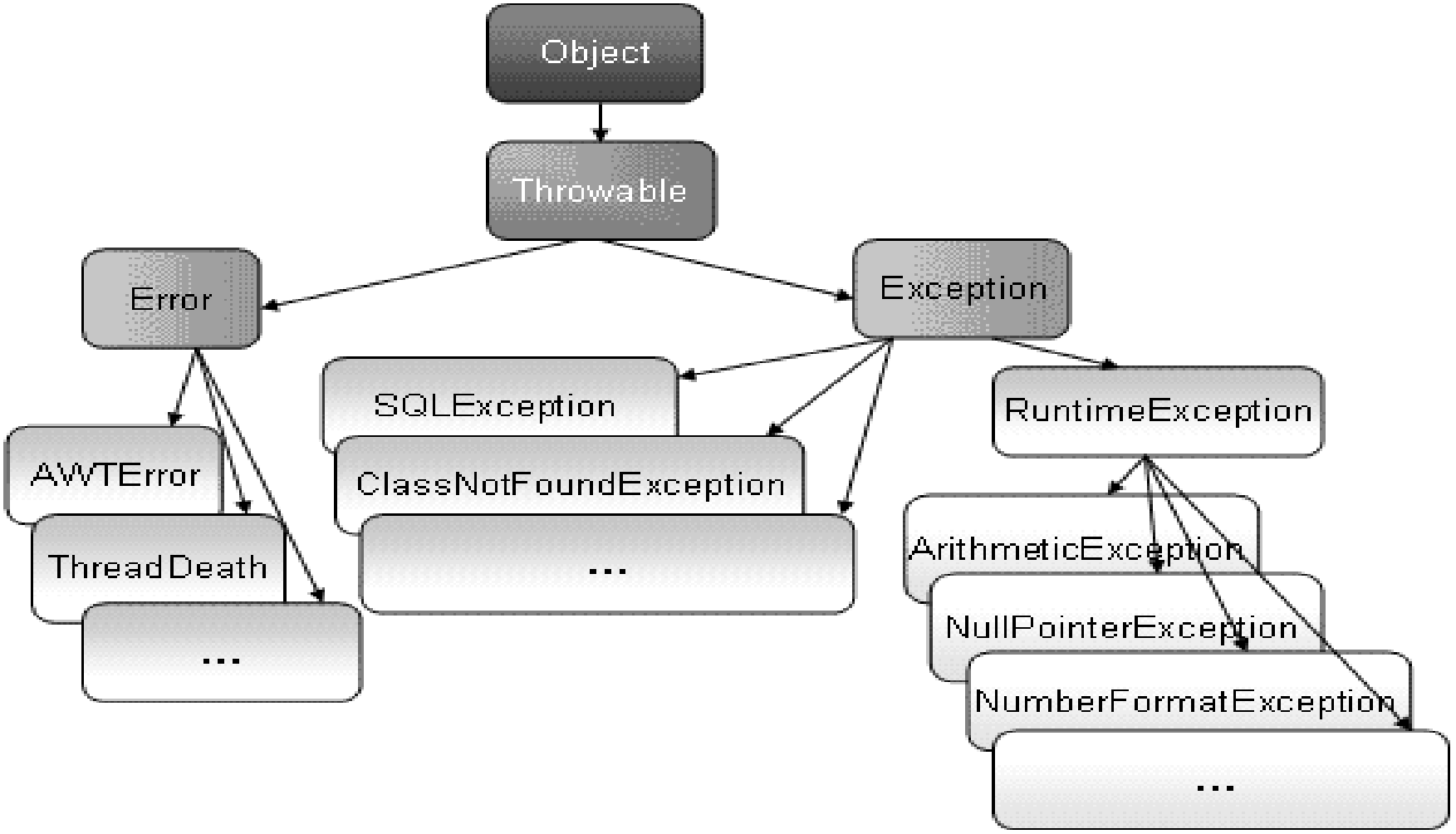
- **Exceptions** in Java separates error handling from main business logic
- SEPERATION OF CONCERNS! ONE BLOCK MUST ONLY DO ONE THING!!!!!!!!!
- Java has a uniform approach for handling all synchronous errors
  - From very unusual (e.g. out of memory)
  - To more common ones your program should check itself (e.g. index out of bounds)
  - From Java run-time system errors (e.g., divide by zero)
  - To errors that programmers detect and raise deliberately



Exceptions

- Either your code or Java signals when something unusual happens.
- The signaling process is called ***throwing*** an exception.
- Somewhere in your program, you can place code to ***handle*** the exception.

Exception Hierarchy



RuntimeException

Exception	Reason for Exception
ArithmeticException	These Exception occurs, when you divide a number by zero causes an Arithmetic Exception
ClassCastException	These Exception occurs, when you try to assign a reference variable of a class to an incompatible reference variable of another class
ArrayStoreException	These Exception occurs, when you assign an array which is not compatible with the data type of that array
ArrayIndexOutOfBoundsException	These Exception occurs, when you assign an array which is not compatible with the data type of that array

RuntimeException

Exception	Reason for Exception
NullPointeException	These Exception occurs, when you try to implement an application without referencing the object and allocating to a memory
NumberFormatException	These Exception occurs, when you try to convert a string variable in an incorrect format to integer (numeric format) that is not compatible with each other
NegativeArraySizeException	These are Exception, when you declare an array of negative size.

RuntimeException

Exception	Reason for Exception
ClassNotFoundException	This Exception occurs when Java run-time system fail to find the specified class mentioned in the program
InstantiationException	This Exception occurs when you create an object of an abstract class and interface
IllegalAccessException	This Exception occurs when you create an object of an abstract class and interface
NotSuchMethodException	This Exception occurs when the method you call does not exist in class

RuntimeException

- **ArithmeticException**

  a=b/0;

- **NullPointerException**

  int a[] = null; a[0] = 1;

  String str = null; str.length()

- **ClassCastException**

  Object obj = new Object(); String str = (String) obj;

RuntimeException

- **NegativeArraySizeException**
  - int a[] = new int [-1];
- **ArrayIndexOutOfBoundsException**
  - int a[] = new int[1]; a[1] = 1;
- **StringIndexOutOfBoundsException**
  - "abc".charAt(-1)
- **NumberFormatException**
  - int j = Integer.parseInt("abc");

Multiple Throws and Catches

- A **try** block can throw any number of different types of exceptions.
- Each **catch** block can catch only one type of exception.
- Multiple **catch** blocks after a **try** block can **catch** multiple types of exceptions.

```
...
try{ }
catch(ArrayIndexOutOfBoundsException e){ }
catch(Exception e) { }
...
```

Catch the More Specific Exceptions First

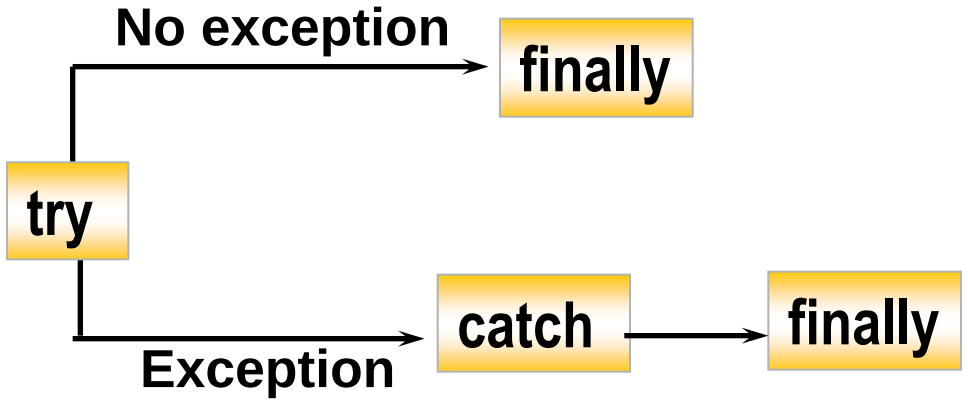
- **catch** blocks are examined in order.
- The first matching **catch** block is executed.
- More specific exceptions should precede less specific exceptions,
- i.e. exceptions lower in the exception hierarchy should come before exceptions higher in the exception hierarchy.

- s. It
- ‹t
- un-

The finally Block

- A **finally** block can be added after a **try** block and its **catch** blocks.
- The **finally** block is executed
  - if the **try** block throws no exceptions
  - if the **try** block throws an exception which is caught by a catch block
  - if an exception is thrown but not caught

i.e. it is always executed.



The finally Block

- Finally is useful when we are processing a file

```
try {
    do something to a file
}catch(IOException e) {
    exception handling
}finally {
    does not matter exception occurs or not, close
    the file.
}
```

throw

We can **throw** an exception explicitly using:

throw <exception object>;

```
try {
        if(flag<0){
            throw new NullPointerException();
        }
}
```

Why?

- When certain exception happens, JVM throws the exception for us. JVM actually creates an object of that exception and throws it.

```
try {
1 int i= 1 ;
2 int j = 0 ;
3 int res = 0 ;
4 res = i/j ; //Exception}
catch(ArithmeticException e)
{ System.out.println("Catch "+e));}
```

- Line 4: try block is interrupted and JVM creates an object of the ArithmeticException and throws it. We don't need to explicitly throw the exception.

Why?

However, some exceptions will not be thrown by JVM, such as:

```
int age = 0;
age = -100;
System.out.println(age);
```

No one's age is -100, in this case, we have to throw the

```
Exception e = new Exception();
throw e;}

Or

throw new Exception();
```

Philosophically

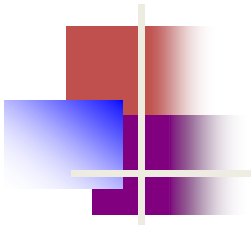
Exceptions come in 3 types

-- the manageable, but is this really an answer;

-- the manageable, and I actually seem to be winning; and

-- good grief, what is the point? I guess something is better than nothing.

Type 1



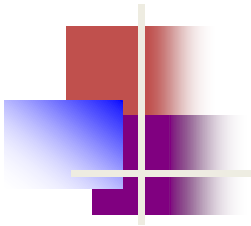
- This is the simplest case of error handling: When a step in some action fails, all the subsequent steps in that action are simply NOT executed.
- This is where exceptions shine because the application code need not worry about checking for errors after each step; once the exception is thrown (either directly or by a called routine), the routine exits automatically.
- Its caller will have a chance to catch it or do nothing and let the exception bubble up to its caller, etc on up the call stack.

Type 1

```
void DoIt() {
        // An exception in Foo means
        // Bar doesn't get called
        Thing thing = Foo();
        Bar(thing);
}

Thing Foo() {
        if (JupiterInLineWithPluto) {
                throw new PlanetAlignmentException();
        }
        return new Thing();
}
```

Type 2

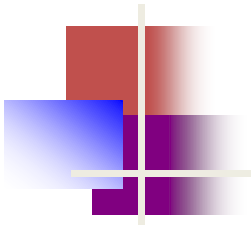


- A second, slightly more advanced case of this error handling is when, like in the first error case you want to halt execution of the current code, but before you do you need to free any resources previously allocated.
- This is different than the "just stop executing the action" case, because we actually need to do some additional work in the presence of the error.
- In garbage collected languages like Java, this it's more typically closing opened files or sockets (although they will eventually get closed by the garbage collector regardless).
- In this style of error handling, you are simply returning resources you've acquired, be it memory, file handles, locks, etc .
- Most programming languages offer simple ways to deal with this: Java has "finally" blocks

Type 2

```
void DoIt() {
        Thing thing = Foo();
        thing.CreateTempFiles();
        try {
                Bar(thing);
                Baz(thing);
        } finally {
                // This gets called regardless
                // of exceptions in Bar and Baz.
                thing.DeleteTempFiles();
        }
}
```

Type 2



- To generalize the description of this type of error handling, you are returning the software back to the default state.
- Whatever intermediate state your code was in is now lost forever.
- Stack frames are popped, memory freed, resources recovered, etc. And that's okay because you want those things to go away and start fresh.

Type 2(b)

- This type of error handling is for error conditions that are known and understood and there is an action the code should take in the situation.
- This differs from other error handling as these errors aren't "exceptional", they are expected and we have alternate paths to take, we don't just go home and pretend like it never happened.
- One example might be attempting to deliver a SMTP mail message and the connection times out. The error handling in that case may be to look for a backup host, or put aside the mail message for later delivery.
- With this type of error handling, status codes are easier to deal with syntactically and logically: "if" and "switch" statements are more compact and natural than "try/catch" for most logic flow.

Type 2(b)

Error codes:

```
if (DeliverMessage(msg, primaryHost) == FAILED) {
        if (DeliverMessage(msg, secondaryHost) == FAILED) {
                PutInFailedDeliveryQueue(msg);
        }
}
```

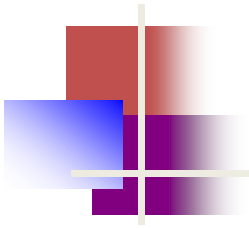
Type 2(b)

Exceptions:

```
try {
        DeliverMessage(msg, primaryHost);
} catch (FailedDeliveryException e) {
        try {
                DeliverMessage(msg, secondaryHost);
        } catch (FailedDeliveryException e2) {
                PutInFailedDeliveryQueue(msg);
        }
}
```

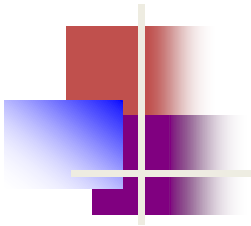
Type 2(b)

- But regardless if you use error codes or exceptions, Plan B error handling isn't particularly difficult.
- The error conditions and scenarios are understood and your code has actions to deal with those scenarios.
- If you use status codes here, this type of error handling is as natural as regular application code. And that's the way it should be, it should be just like adding any other branching logic.
- Exceptions aren't as useful here, because in this case they aren't "exceptional" and the code to handle common conditions becomes much more convoluted.



Type 3

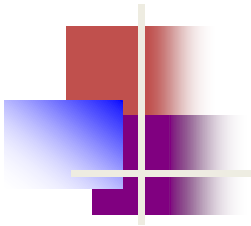
The third, and truly nastiest case of error handling, is when you must "undo" any state changes your program has made leading up to the error condition. This is where things can get real complicated real quick, you aren't just freeing resources like before, you are backing up in time to a previous program state.



Type 3

- And how do you do that?
- How do you put back state you've changed?
- Do keep a copy of every variable and property change so you can put it back?
- Where do you keep it?
- What if the change is down in some deeply nested composite object?
- What if another thread or some other code already sees the state change and acted on it?
- What happens if another error happens while putting stuff back?

Type 3



- **This is the hard stuff.**

- This is the stuff where the error handling easily becomes as complex as the application logic, and sometimes to do it right it has to be even more complex.

- So what can we do?

Type 3

- What techniques or secrets can we use to make this error handling easier? If only we had something that reversed the actual flow of time, that could do the trick.
- Or maybe we shouldn't be trying to figure out an easier way to do this type of error handling, but rather *avoiding the need for it altogether*.
- No good answers exist at this point – number one strategy,

minimise the number of mutation (or write) operations and avoid unnecessary (especially local) variables.