Exception Handling

Advanced Programming

Introduction

- Good programmer's assumption:

"Any code of significant length has bugs"

Techniques to code with quality

- **Design code** that can:
  - **Detect errors** when they happen, without crashing the program or, worse, the system (**fault awareness**)
  - **Recover from an error**, giving the user a choice to continue using the program and to save the greatest possible amount of work already done (**fault recovery**)
  - **Keep on running** consistently notwithstanding an error, still performing activities that are not affected by the error (**fault tolerance**)

Error handling

- All these techniques, from a programmer's perspective, are collectively known as **error handling**

- The best would be to have any portion of a program to be **fault tolerant**, but
- usually we have to be happy if only the **critical parts of the code are fault tolerant** (or, at least, able to recover from an error), while the others are only fault aware

Classic error handling

- Pre-OO era languages, as C or Pascal:
  - **Return an error code to the caller of a function,**
    - either using the return value of the function itself or an additional output parameter
- Extremely **error-prone technique**, the user of a method ("the caller") must test the error code to see if the function had performed flawlessly
- Programmers often forgot, or neglected, to test error codes

Classic error handling

- Excess of locality
  - sometimes the caller **doesn't have enough information** to recover from an error issued from a function
- Therefore the solution usually were:
  - passing the error code further up the calling tree
  - calling global error handling routines through **goto-like instructions**
  - using **global variables**

Classic error handling

- Such programs were highly coupled and intermixed
  - rather obscure, i.e. extremely difficult to read, unless very well documented
- The concept of exception

The concept of exception

The concept of exception

- In a very general sense, an exception is **any event that does not belong to the normal flow of control** of a program
- An exception could be an error…
  - e.g. running out of memory, a wrong argument value, etc.

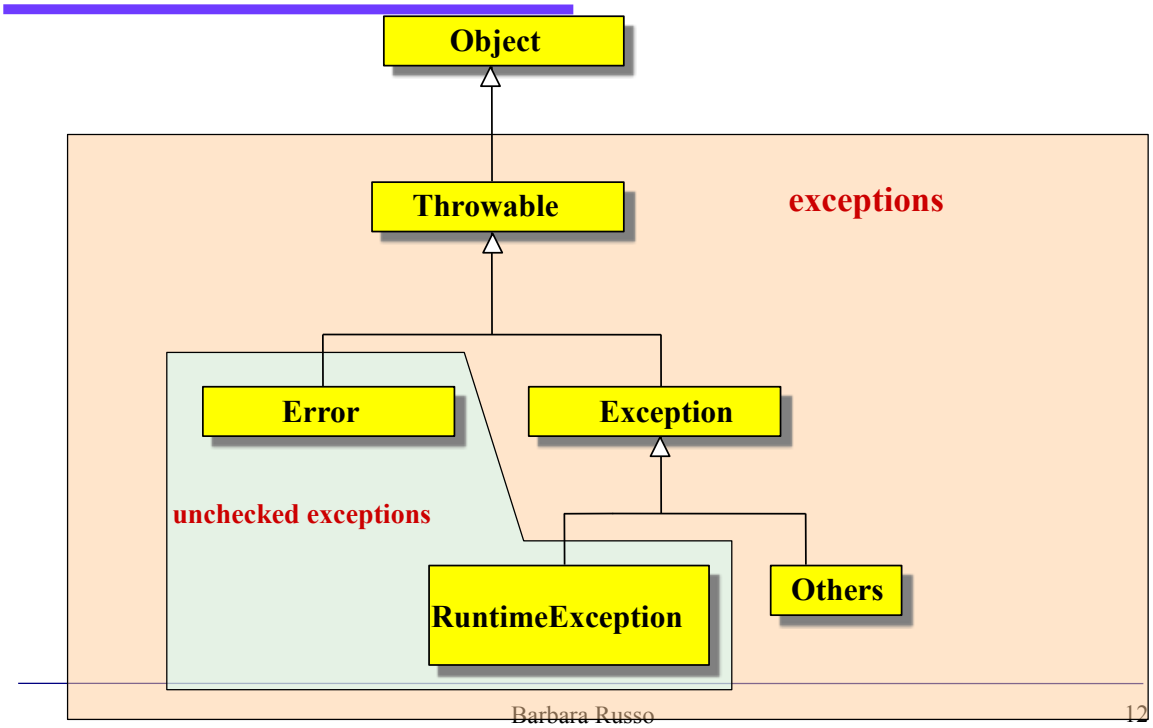
The concept of exception

- … But this is not always the case, an exception could also be:
  - an **unusual** result from a math computation routine
  - an **unexpected** (but not wrong) request to an operating system
  - the **detection** of an input from the "**external world**" (say an interrupt request, a login request, a power shortage, etc.)

Exceptions in OO languages

- In OO languages, **exceptions are objects**, i.e. instances of some class
- They are used to pass information about unusual events to and from the different parts of the program that are design to work with them

Hierarchy of exception classes in Java



Throwing Exceptions in Java

- Every time a piece of code detects an unusual event it can throw an exception to signal this event to its caller(s), or — more generally — to the **outer dynamic scopes** (more on this later on)

Ways to throw exceptions

- There are two possible ways to throw an exception
  - Directly for a malfunctioning piece of code enclosed in a **try block**
  - Through a so-called **throw statement**, i.e. the keyword **throw** followed by an **object** of a class derived (directly or indirectly) from class java.lang.Throwable

`throws` in the method signature

- As throwing and catching exceptions affect the way a function relates to other functions, a `throws` statement followed by the **type of the exception** can be included in the **method declaration**

`throws` in the method signature

- For each **checked exception,**
- if you do not include in a method/constructor signature a "throws" clause for that **exception type (or a superclass** of that exception type)
- a **compile-time error** occurs

Example

```
void move(Train train) throws T1, T2;
```

- The throws clause allows the compiler to ensure that code for handling such error conditions has been included

Example

NegativeArgument is a trivial class (is empty!), only used to instantiate exceptions

```
public class NegativeArgument extends Throwable { }

public class A {

   public void compute(int x) throws NegativeArgument {
       if ( x < 0 ) throw new NegativeArgument();
   }

}
```

Exceptions specification in the method declaration is mandatory for **checked** exceptions in Java

Here a new object is instantiated; then it is thrown as exception, to indicate that the argument supplied to the method has a wrong value

Exception handlers

- Thrown exception follow an alternative path of execution, until they are **caught** somewhere by special constructs, called **exception handlers**
- Exception handlers are sort of functions, having one and only one argument, whose **type must (substantially) match** that of the exception being caught
- Exception handlers are specified in **catch statement**

Try-catch statements

- Any time we want to provide handlers for some kinds of exceptions, we must **enclose** the **code** that may throw the exceptions in a **try-block**
- Right after the try-block we can specify the handlers, using the keyword **catch** followed by an argument list (as if it were a function)
- Error handlers must have just one argument, and it must be of the same type of the exception being caught

finally clause in Java (1/2)

- In Java the try-catch statement can be provided with a **finally clause** after all the handlers
- A finally clause is made of the keyword **finally** followed by a block
- The code inside that block is **always executed**, no matter if the code in the previous try-block throws an exception or not

finally clause in Java

- If it **does not throw** an exception, then the finally-block is executed just **after the try-block** has ended
- If it **does**, then …
  - If the exception is handled by a previous catch-clause, the finally-block is executed **just after the exception has been handled**
  - If there is no handler matching the exception, the finally-block is **executed just before stack unwinding (see later) resumes and looks for an handler in an outer dynamic scope**

Question

- Is the following code legal?

```
try {
} finally {
}
```

Solution: yes finally will be always executed

Question

- Is there anything wrong with this exception handler as written? Will this code compile?

```
try {
} catch (Exception e) {
} catch (ArithmeticException a) {
}
```

Solution: the first catch catches all the types of exception the second will be never reached

Question

1. `int[] A;`

   `A[0] = 0`
2. A program is reading a stream and reaches the end of stream marker
3. Before closing the stream and after reaching the end of stream marker, a program tries to read the stream again

Match each case in the list on the left with an item below:

__checked exception

__compile error

__no exception

Answers:

1 (compile error). The array is not initialised

2 (no exception)

3 (checked exception)

```
public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entering" + " try statement");
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
    } catch (IndexOutOfBoundsException e) {
        System.err.println("Caught IndexOutOfBoundsException: " +  e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " +  e.getMessage());
    } finally {
        if (out != null) {
            System.out.println(“Closing PrintWriter");
            out.close();
        }
        else {
            System.out.println("PrintWriter not open");
        }
    }
}
```


Stack unwinding mechanism

Semantics of Throwing Exceptions

- When a **throw statement** is executed,
- the normal flow of control is interrupted, and
- a special mechanism, known as **stack unwinding**, takes place

Semantics of Throwing Exceptions

- Stack unwinding is used to determine **where to resume execution**
- To understand where the exception handling mechanism resumes execution after a throw, we must examine the concept of dynamically enclosing scope

Dynamically enclosing scopes

- A simple definition of dynamically enclosing scope (**DES**) :
  - A dynamically enclosing scope is any **scope whose activation record lies on the stack** at the moment when that statement is executed or equivalently
  - Any executable code block which has called the function where the statement is placed, directly or indirectly

Example of dynamically enclosing scopes

pseudo-code

```
class A {
   void f() {
        for ( /*…*/ ) {
             statement;
        }
   }
}
```

These are always DES for "statement": anytime "statement" is executed, their ARs are on the stack

```
void g(A anA) {
   anA.f()
}
```

This is a DES for "statement" only when "statement" is executed because g was called

```
void h() {
   {
        A anotherA;
        g(anotherA);
   }
}
```

This is a DES for "statement" only when "statement" is executed because h was called

Stack unwinding mechanism

- Throwing an exception causes the normal flow of control to **stop** and execution is **diverted** on an alternative path
- Execution resumes in an exception handler whose argument type **matches** that of the exception thrown
- If there is no such handler, the exception is passed to the system, which usually issues a **run-time error**

Stack unwinding mechanism

- To find the right handler to call, if any, the stack unwinding mechanism must find the **nearest try-block dynamically enclosing the throw statement**
  - If this try-block has a handler whose argument matches the thrown exception, then its body is executed, after passing the exception as an argument
  - if there is no such handler, the stack unwinding resumes, and it searches outwards the next **dynamically enclosing try-block for a matching handler** (and so on, recursively, until the exception is handled or the system issues an error)

Stack unwinding mechanism

- After the execution has entered the body of the handler, the exception itself is considered handled, i.e. the stack unwinding mechanism stops
- This process of reaching out for outer dynamically enclosing scopes is also known as **exception propagation**
  - An exception propagates out to the outer dynamic scopes until it is properly handled or it reaches the system

Exception Propagation

- If an exception is never caught anywhere in the hierarchy, the Java interpreter will print an error message, the stack trace and exit
- To do this, it invokes the default handler

Default handler

- Any exception that is not caught by your program will ultimately be processed by the **default handler**
  - It is provided by the JRE
- The default handler displays a string describing the exception, prints a **stack trace** and terminates the program, e.g.,:

```
Java.lang.ArithmeticException: / by zero
     at Dates.main(Dates.java:4)
```


Stack Trace

- Stack trace from the default exception handler shows the entire call stack
- The call stack is quite useful for **debugging**, because it pinpoints the precise sequence of steps that led to the error

Throwable.getStackTrace()

- Analysing the flow of program logic, step by step is called **tracing**
- `Throwable.getStackTrace()` tells you where you are in the code, and how you got there
- It lets you put the name of the current method in error messages, or the name of the caller of the current method, or its caller etc.
- see LECT7 code


```
Throwable t = new Throwable();
StackTraceElement[] es = t.getStackTrace();
for ( int i=0; i<es.length; i++ ){
   StackTraceElement e = es[i];
   System.out.println( " in class:" + e.getClassName()
          + " in source file:" + e.getFileName()
          + " in method:" + e.getMethodName()
          + " at line:" + e.getLineNumber()
          + " " + ( e.isNativeMethod() ? "native" : "" ) );
   }
```

Notes

- When you use multiple **catch** statements, the exception subclasses must come before any of their superclasses
  - This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
  - Thus, a subclass would never be reached if it came after its superclass.
  - Unreachable code is an error (compile–time).
  - To fix the problem reverse the order of catches

Notes

- If we attempt to throw and exception from a method without declaring throws or sounding it with try/catch, it will not compile:

```
public void compute(){
     throw new Exception();
}
```

X

```
public static void  compute() throws Exception{
     throw new Exception();
}
```

```
public void  compute(){
     try {
          throw new Exception();
     } catch (Exception e) {

          e.printStackTrace();
     }
}
```

The statement assert

- The statement **assert** throws an exception of type AssertionError
  - We do not need the statement throw or to catch exc.

```
public class AssertionExample {
   static void removeNode(Object x) {

         /*Try and Catch blocks are not required because
         the exception is already handled with the statement assert */

         ArrayList myVector = new ArrayList();

         assert(myVector.size()!=0);
         myVector.remove(x);

   }
   public static void main(String[] args){
         Object myObject = new Object();
         AssertionExample.removeNode(myObject);
   }
}
```


Notes

- Like all uncaught exceptions, assertion failures are generally labeled in the stack trace with the file and line number from which they were thrown
- Assertion raise EssertionError exceptions (child of Error)

Notes

- In some cases **assert** may be expensive to evaluate. For instance,
  - A method to find the minimum element in an unsorted list and an assertion verifies that the selected element is indeed the minimum
- Use assert when the computation is at least as expensive as the work done by the method itself

Disabling assertions

- Assertions can be enabled or disabled when the program is started, and are disabled by default
- Once disabled, they are essentially equivalent to empty statements in semantics and performance
- Thus, do not disable them or use assertions in cases that can

When using assertions

- Internal Invariants (with switch statements)
- Control-Flow Invariants (if conditions)
- Preconditions and Postconditions

When using assertions

- As a rule of thumb, use assertions when **the expressions contained in them are free of side effects:**
  - evaluating the expression should not affect any state that is visible after the evaluation is complete.

When not to use assertions

- Do not use assertions to do any work that your application requires for regular operation!
- Because assertions may be disabled, programs must not assume that the boolean expression contained in an assertion will be evaluated

Example

- Remove all of the null elements from a list names, and knew that the list contained one or more nulls. It would be **wrong** to do this:

```
assert names.remove(null);
```

- It works fine when asserts are enabled, but would fail when they were disabled, as it would no longer remove the null elements from the list

Example

- The **correct** code is to perform the removal before the assertion and then assert that the removal succeeded:

```
boolean nullsRemoved = names.remove(null);
assert nullsRemoved;
```


Enabling and disabling assertion

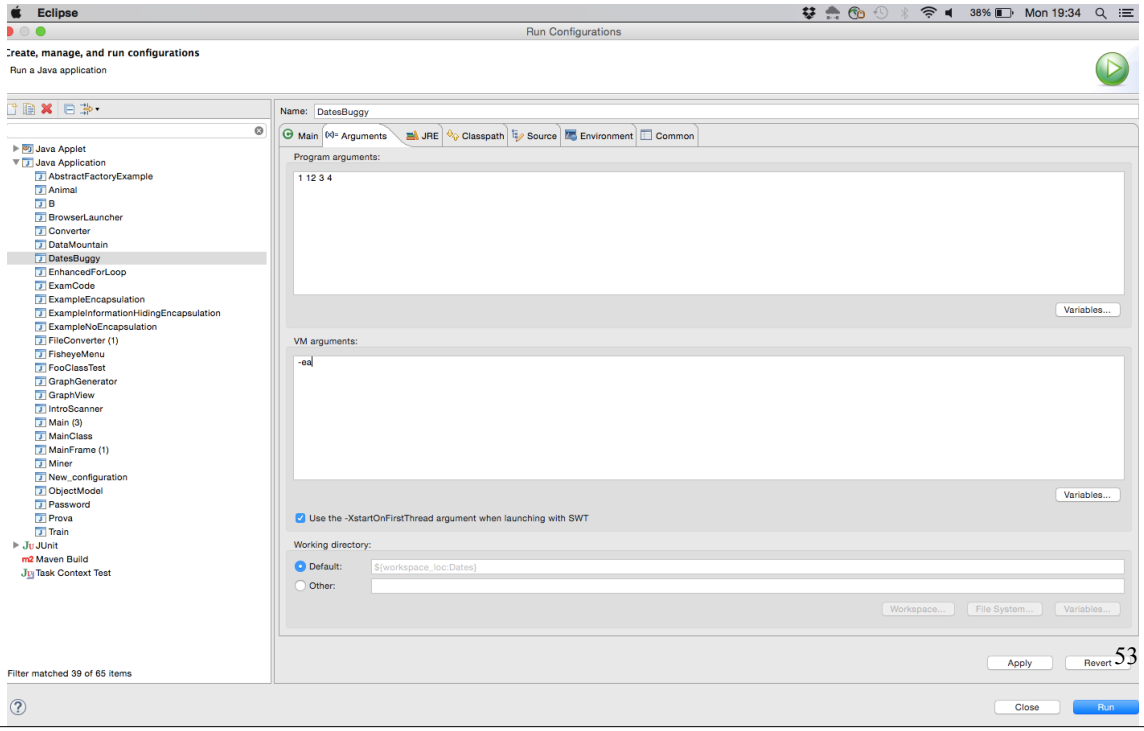
- Enable in package it.unibz.inf

java -ea:it.unibz.inf... Dates

- Disabling in class it.unibz.inf.Calendar

java -da:it.unibz.inf.Calendar Dates

Enabling assertion in Eclipse

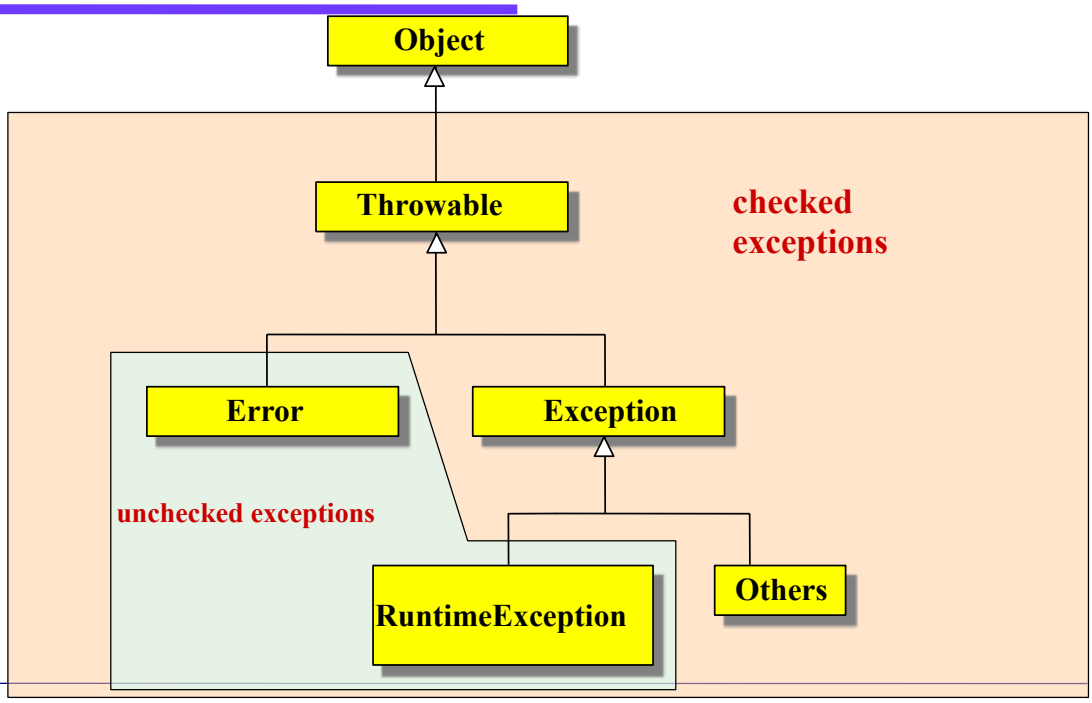


Java checked and unchecked exceptions

Checked and unchecked exceptions

- As we said, every exception class must be derived from **java.lang.Throwable**
- Any exception derived directly from Throwable or its derived class **Exception** is known as **checked exception**
- Any exception derived from the following other two subclasses of Throwable is called **unchecked**:
  - java.lang.**RuntimeException** (derived from java.lang.Exception)
  - java.lang.**Error** (derived directly from Throwable)

Hierarchy of exception classes in Java



Defining Exceptions

- A programmer can define her/his own exceptions, both
  - Runtime Exception
  - Error (not advisable) and
  - Checked Exception

Declaring Exceptions in methods

- Any method that causes an exception to occur must
  - either **catches** the exception (with try, throw or assert followed by catches)
  - or specifies the type of the exception (or a superclass of it) with a **throws** clause in the method declaration (checked exception)

Unchecked exceptions

- **Unchecked exceptions** are not required to be caught by a method or do not require to declare “throws” in the method signature e.g.,:
  - ArrayIndexOutofBoundsException
  - NullPointerException

When to use them

- If a client can reasonably be expected to recover from an exception, make it a checked exception.
- If a client cannot do anything to recover from the exception, make it an unchecked exception.

Java unchecked exceptions

- Unchecked exceptions impose no extra constraint in writing the code and are thrown by the JRE
- You can define your own unchecked exception classes by subclassing Error or RuntimeException

Error

- Error is a base class for all the exceptions thrown by the system (operating system, virtual machine, etc.) so it is not advisable to use it as a superclass for user-defined exceptions

RunTimeException

- RuntimeException is a base class for exceptions that impose an unbearable burden on the programmer, if they were implemented as checked exceptions

Runtime exceptions

- Runtime exceptions can occur anywhere in a program, and in a typical one they can be very numerous
- Having to add runtime exceptions in every method declaration would reduce a program's clarity
- Thus, the compiler does not require that you catch or specify runtime exceptions (although you can).

Java checked exceptions

- As we said, any attempt to violate the contract provided by the exceptions' specification will cause a compile-time error
- A programmer can define her/his own checked exception classes by subclassing Throwable (not advisable) or (better) Exception

Examples

- **Checked exception:** java.io.IOException
- Example. Suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor of java.io.FileReader
- Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally
- But sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException
- A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name

Examples

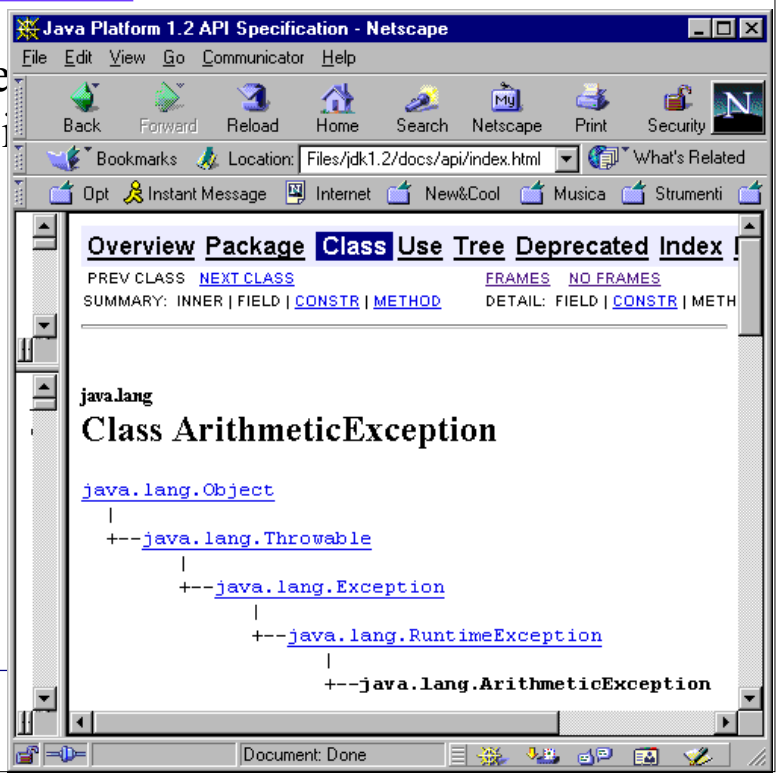
- **Unchecked: java.lang.Error**
- These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.
- Example. Suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction
- The unsuccessful read will throw java.io.IOError
- An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense to print a stack trace and exit

Examples

- **Unchecked: java.lang.RuntimeException**
- These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from
- These usually indicate programming bugs, such as logic errors or improper use of an API.
- For example, consider the application described previously that passes a file name to the constructor for FileReader
- If a logic error causes a null to be passed to the constructor, the constructor will throw NullPointerException
- The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur

Built-In Exceptions

- You can check whe
  exception from thei



Since SDK 7

- Three exception features:
  - try-with-resources
  - multi-catch, and
  - precise re-throw

try-with-resources

- To close a file that we do not need anymore we call close( )
  - Forgetting to close a file can result, for example, in memory leaks (i.e. a failure in a program to release discarded memory, causing impaired performance or failure.)
- The **try-with-resources** statement automatically closes a file when it is no longer needed.
  - In this approach, no explicit call to close( ) is executed

try-with-resources statement

- It is a try statement that declares one or more resources.
  - A resource is an object that must be closed after the program is finished with it
- It ensures that each resource is closed at the end of the statement.
  - Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource

Example from Java Tutorial

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```


With traditional try

- We use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly or use

Prior SDK 7

```
static String readFirstLineFromFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();//it throws an IOException
    } finally {
        try{
            if (br != null) br.close(); //it throws another IOException
       }catch(IOEXception e){System.out.println("Error for closing file")}
    }
}
```

close() can still throw an **IOException suppressing the exception** that eventually is thrown by readFirstLineFromFile() method; with traditional try and catch (as here) the method exception is lost.
With the try-with-resource (as previous slide) the exception of the method is suppressed and one can get it back by calling getSuppressed()

multi-catch

- Multi-catch feature allows two or more exceptions to be caught by the same catch clause
  - separate each exception type in the catch clause with the OR operator
  - Each multi-catch parameter is implicitly final (not need to use final)

Example

```
// Demonstrate the multi-catch feature.
class MultiCatch {
  public static void main(String args[]) {
    int a=10, b=0;
    int vals[] = { 1, 2, 3 };

    try {
      int result = a / b; // generate an ArithmeticException

//      vals[10] = 19; // generate an ArrayIndexOutOfBoundsException

      // This catch clause catches both exceptions.
    } catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception caught: " + e);
    }

    System.out.println("After multi-catch.");
  }
}
```


Cleaner code

```
catch (IOException) {
    logger.log(ex);
    throw ex;
}
catch (SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Cleaner code

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```


Precise re-throw after Java SE 7

- There are times when you want to re-throw an exception that you have handled

Example wrapping

```
public class ExampleExceptionRethrowBeforeSE7{
    public static void demoRethrow() {
        try {
            throw new IOException("Error");
        }
        catch(IOException exception) {
            //Do some handling and then re-throw:
            throw new RuntimeException(exception);
        }
    }
    public static void main( String[] args ) {
        try {
            demoRethrow();
        }
        catch(RuntimeException exception) {
            System.err.println(exception.getCause().getMessage());
        }
   }
}
```


Issues

- The problem with the “Example wrapping” is that it is not really re-throwing the original exception
- It is wrapping it within another exception
- The developer needs to remember that an exception has been wrapped into another one: the code does not know it!
- For example we use “getCause()”

Example two children

```
static class FirstException extends Exception {
     public FirstException(String message) {
          super(message);
     }
}
static class SecondException extends Exception {
     public SecondException(String message) {
          super(message);
     }
}
public static void rethrowException(String exceptionName) throws Exception {
    try {
        if (exceptionName.equals("First")) {
           throw new FirstException();
        } else {
           throw new SecondException();
        }
    } catch (Exception e) {
      throw e;
    }
  }
```


Issues

- When the method is called the developer does not know which exception has been raised
- We cannot use getCause() as methods are static and we always get the message of the parent class

```
public static void main(String[] args ){
        try {
            rethrowWrapping();
        }
        catch(RuntimeException exception){
            System.err.println(exception.getMessage());
        }
}
```

Precise re-throw after Java SE 7

- When you declare one or more exception types in a catch clause, and re-throw the exception handled by this catch block, the **compiler** verifies that the type of the re-thrown exception meets the following conditions
  - the associated try block can throw it,
  - It is not handled by a preceding catch clauses, and
  - It is a subtype or supertype of the catch parameters

After Java SE 7

- The **compiler** allows you to specify the exception types in the throws clause in the method declaration because you can re-throw an exception that is a supertype of any of the types declared in the throws

Example: wrapping

```
public class ExampleExceptionRethrowSE7{
   public static demoRethrow() throws IOException {
    try {
        throw new IOException("Error");
    }
    catch(Exception exception) {
         // Do some handling and then re-throw:
        throw exception;
    }
   }

   public static void main( String[] args ){
    try {
        demoRethrow();
    }
    catch(IOException exception) {
    System.err.println(exception.getMessage());
    }
   }
}
```

- **checked exceptions that the associated try block throws,**
- **are not handled by a preceding catch clause, and**
- **are a subtype or supertype of the parameter**

Example: two children

```
public void rethrowException(String exceptionName)
  throws FirstException, SecondException {
    try {
      if (exceptionName.equals("First")) {
           throw new FirstException();
        } else {
           throw new SecondException();
        }
    }
    catch (Exception e) {
      throw e;
    }
  }
```

Ex. on exception management in Java (1/2)

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

class NegError extends RuntimeException {
}

public class SlideCodeTest {

    static int neg(int x) {
          if (x < 0)
                throw new NegError();
          return -x;
    }

// continued on next page…
```

Here the exception class is derived from **RuntimeException** (derived from **Throwable**), instead of being derived directly from **Throwable**. This to avoid checked exception.

There is no need to include throws in the declaration of the method as it is an **unchecked exception**

Here we use the reference returned by operator **new** as the operand of the **throw** clause.

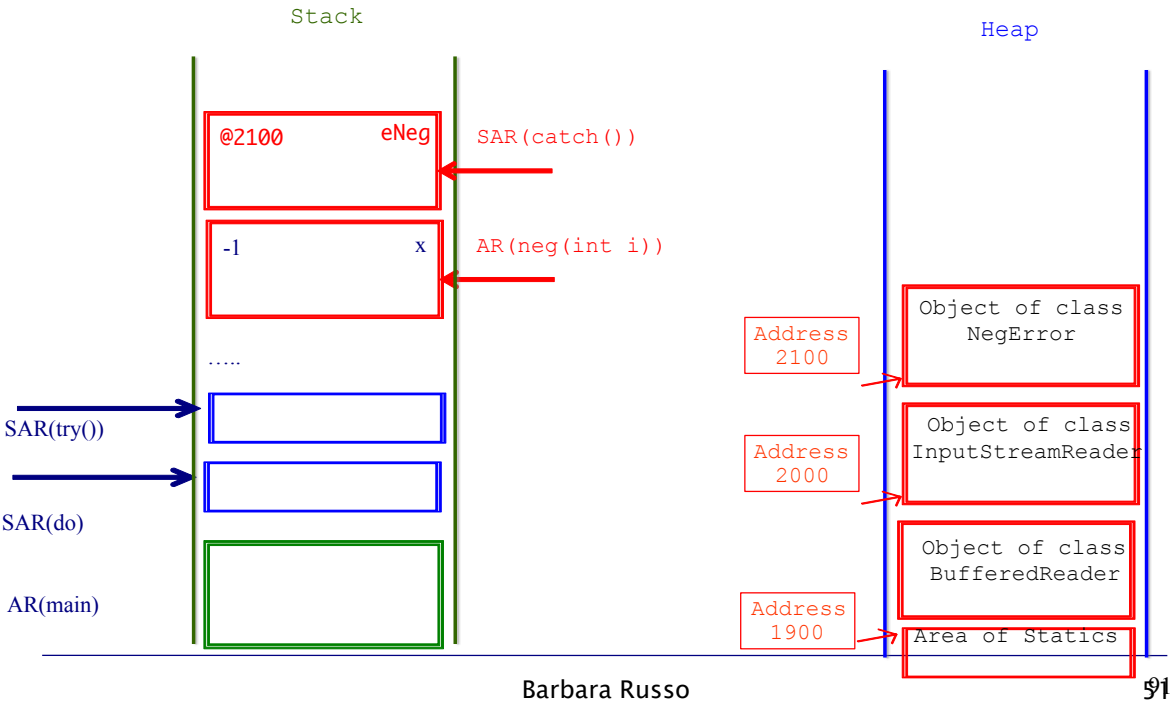
Ex. on exception management in Java (2/2)

```
   //continues
   public static void main(String[] argv) {
         int i = 0;
         boolean repeat = false;
         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
         do {
               repeat = false;
               try {
                     i = Integer.parseInt(in.readLine());
                     i = neg(i);
               }
               catch (NegError eNeg) {
                     System.out.println("\nYou entered a negative number! - Retry.\n");
                     repeat = true;
               }
               catch (IOException e) { }
         }
         while (repeat);
         System.out.println("result = " + i);
   }
}        // end of class
```

We had to put this statement inside the try block because readLine can issue a checked exception (see API of BufferedReader readline())

Since readLine can issue a checked exception (IOException), we had to provide a suitable handler, but it's empty for simplicity

With no IOException and negative i



Another example

Here the exception class is derived from **RuntimeException** but we want the compiler to check the exception anyway

```
class Natural {
    public int val;
     Natural() {
       val = 0;
     }
     Natural( int i ) throws ArithmeticException  {
         if ( i < 0 ) {
             val = 0;
             throw new ArithmeticException( "A Natural cannot be negative!" );
         }
         val = i;
     }
     public String toString() { return Integer.toString(val); }
}
```

Another example

```
public class SimpleExceptions {
   public static void main(String args[] ) {
      Natural a, b, c, d, f;
      a = new Natural(); b = new Natural(2);
      System.out.println("The value of a is " + a.val);
      System.out.println("The value of b is " + b.val);
      c = null;
      try {
          c = new Natural(1);
       }
      catch(ArithmeticException e) {
          System.out.println("Raised exception: " + e);
       }
      System.out.println("The value of c is:" + c.val);
      d = null;
      try {
          d = new Natural(-1);
       }
      catch(ArithmeticException e) {
          System.out.println("Raised exception: " + e);
      }
      System.out.println("The value of d is:" + d.val);
      f = new Natural(-3);
      System.out.println("The value of f is:" + f.val);
    }
}
```

Output

```
The value of a is 0
The value of b is 2
The value of c is:1
Raised exception: java.lang.ArithmeticException
The value of d is:null
java.lang.ArithmeticException: A Natural cannot be
negative!
        at Natural.<init>(SlideCodeTest.java:11)
        at SlideCodeTest.main(SlideCodeTest.java:45)
Exception in thread "main"
```

It is a run-time exception. Here is not caught so it propagates out of the DES

Notes

- One can declare “throws” in the class method without declaring it in the interface implemented by the class
- For example, the class constructor “throws” an exception but an interface does not have a constructor so cannot declare any “throws” for a constructor

Notes

- If the parent class throws an exception in the constructor, then the subclass **must** have a constructor (an empty one is good enough) to evidence the propagation of the exception.

```
class A {
  A() throws Exception { /* something */ }
}
class B extends A {
  B() { /* do nothing */ }
}
```


Exercise

- Complete the exercise on Natural Number by adding in the following methods:
  - divide
  - subtract
- Each method should throw appropriate exceptions to signal the occurrence of errors