



# Exception Handling



# Program Errors

- Syntax (compiler) errors
  - Errors in code construction (grammar, types)
  - Detected during compilation
- Run-time errors
  - Operations illegal / impossible to execute
  - Detected during program execution
  - Treated as **exceptions** in Java
- Logic errors
  - Operations leading to incorrect program state
  - May (or may not) lead to run-time errors
  - Detect by debugging code



# Exception-Handling Fundamentals

- An *exception* is an abnormal condition that arises in a code sequence at run time
- A Java exception is an object that describes an **exceptional condition that has occurred** in a piece of code
- When an exceptional condition arises, an object representing that exception is created and **thrown in the method** that caused the error
- An exception can be caught to handle it or pass it on
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code

# What are Exceptions?

Many “exceptional” things can happen during the running of a program, e.g.:

- User mis-types input
  - Web page not available
  - Array index out of bounds
  - Method called on a null object
  - Out of memory
  - Bug in the actual language implementation
  - File not found
  - Divide by zero
- checked
- unchecked
- sys errors

Exceptions are unexpected conditions in programs.

We can distinguish 3 categories:

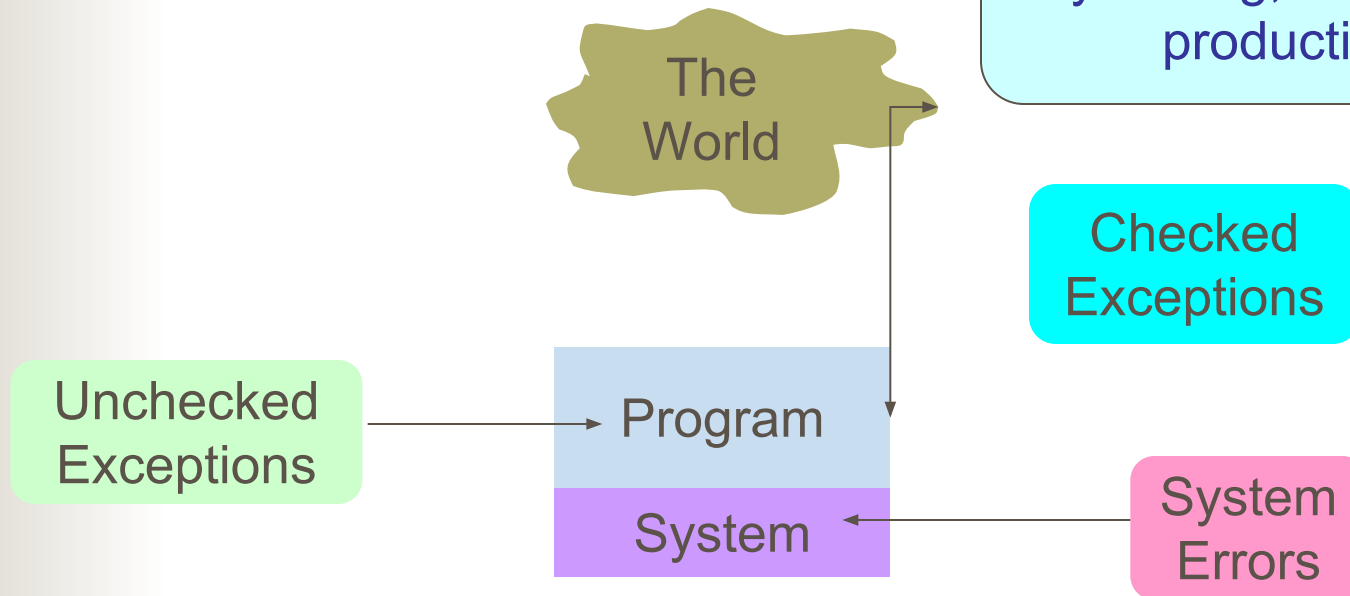
- *checked* exceptions — Problems to do with the program's interaction with “the world”.

- *unchecked* exceptions — Problems to do with the program (i.e. violations of the contract).

- *system errors* — Problems to do with the hardware. These are outside our control.

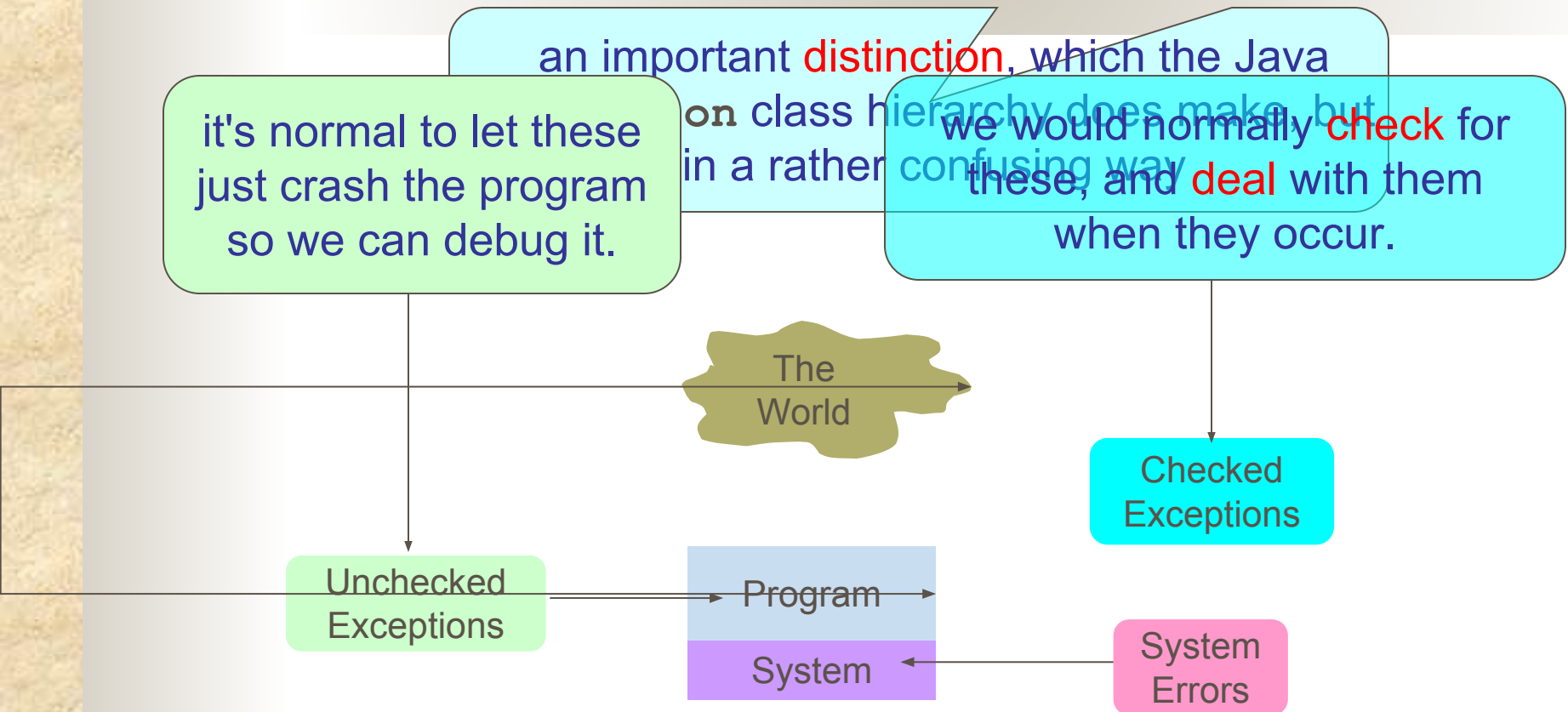
The world is unpredictable, so we would **expect** these things to happen in production code, and so need to handle them.

These should be removed by testing, and not occur in production code.





# Checked vs Unchecked Exceptions



**Exception handling** is the business of handling these things appropriately.



# Exception Handling

- Performing action in response to exception
- Examples
  - Exit program (abort)
  - Ignore exception
  - Deal with exception and continue
    - Print error message
    - Request new data
    - Retry action

# Exception Handling – Exit Program

- Approach

- Exit program with error message / error code

- Example

```
if (error) {  
    System.err.println("Error found"); // message  
    System.exit(1);                    // error code  
}
```

- Problem

- Drastic solution
  - Event must be handled by user invoking program
  - Program may be able to deal with some exceptions



# Exception Handling – Error Code

- Approach

- Exit function with return value  $\Rightarrow$  error code

- Example

A( ) { if (error) return (-1); }

B( ) { if ((retval = A( )) == -1) return (-1); }

- Problems

- Calling function must check & process error code
    - May forget to handle error code
    - May need to return error code to caller
  - Agreement needed on meaning of error code
  - Error handling code mixed with normal code

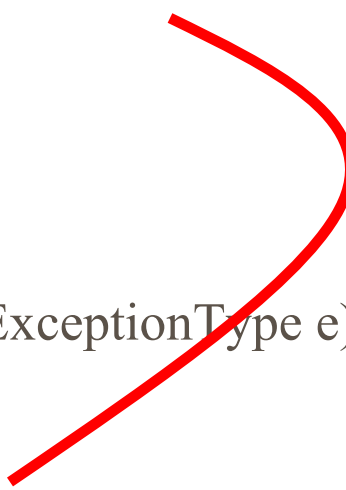
# Exception Handling – Throw Exception

- Approach
  - Throw exception

- Example

```
A() {  
    if (error) throw new ExceptionType();  
}
```

```
B() {  
    try {  
        A();  
    }  
    catch (ExceptionType e) { ...action... }  
}
```



**Java exception backtracks to caller(s) until matching catch block found**

# Exception Handling – Throw Exception

## ■ Advantages

- Compiler ensures exceptions are caught eventually
- No need to explicitly **propagate** exception to caller
  - **Backtrack** to caller(s) automatically
- Class hierarchy defines meaning of exceptions
  - No need for separate definition of error codes
- Exception handling code separate & clearly marked



# Exception-Handling Fundamentals

- Java exception handling is managed by via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**
- Program statements to monitor are contained within a **try** block
- If an exception occurs within the **try** block, **it is thrown**
- Code within **catch block** catch the exception and handle it
- **System generated exceptions** are automatically thrown by the Java run-time system
- To **manually throw an exception**, use the keyword **throw**
- Any exception that is thrown out of a method must be specified as such by a **throws** clause



# Using try and catch

- Handling an exception has two benefits,
  - It allows you to fix the error
  - It prevents the program from automatically terminating
- The **catch** clause should follow immediately the **try** block
- Once an exception is thrown, program control transfer out of the **try** block into the catch block
- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism





# Exception-Handling Fundamentals

- Any code that absolutely must be executed before a method returns is put in a **finally** block
- General form of an exception-handling block

```
try{  
    // block of code to monitor for errors  
}  
catch (ExceptionType1 exOb){  
    // exception handler for ExceptionType1  
}  
catch (ExceptionType2 exOb){  
    // exception handler for ExceptionType2  
}  
//...  
finally{  
    // block of code to be executed before try block ends  
}
```

# Example

```
class Exc2 {  
    public static void main(String args[]) {  
        int d, a;  
  
        try { // monitor a block of code.  
            d = 0;  
            a = 42 / d;  
            System.out.println("This will not be printed.");  
        } catch (ArithmeticException e) { // catch divide-by-zero error  
            System.out.println("Division by zero.");  
        }  
        System.out.println("After catch statement.");  
    }  
}
```

## Output:

Division by zero.

After catch statement.

# Using try and catch

- A **try** and **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement

```
import java.util.Random;

class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();

        for(int i=0; i<10; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmeticException e) {
                System.out.println("Division by zero.");
                a = 0; // set a to zero and continue
            }
            System.out.println("a: " + a);
        }
    }
}
```



# Uncaught Exceptions

- If an exception is not caught by user program, then execution of the program stops and it is caught by the default handler provided by the Java run-time system
- Default handler prints a stack trace from the point at which the exception occurred, and terminates the program

**Ex:**

```
class Exc0 {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;  
    }  
}
```

**Output:**

```
java.lang.ArithmeticException: / by zero  
    at Exc0.main(Exc0.java:4)
```

Exception in thread "main"





# Multiple catch Clauses

- If more than one can occur, then we use multiple catch clauses
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed
- After one **catch** statement executes, the others are bypassed



# Example

```
class MultiCatch {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch (ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}
```



## Example (Cont.)

- If no command line argument is provided, then you will see the following output:  
a = 0  
Divide by 0: java.lang.ArithmeticException: / by zero  
After try/catch blocks
- If any command line argument is provided, then you will see the following output:  
a = 1  
Array index oob: java.lang.ArrayIndexOutOfBoundsException  
After try/catch blocks.

# Example

```
public class Etest {
public static void main(String args[]){
// What we expect to happen
try {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    System.out.println( x + "/" + y + " = " + x/y ); }

// Things which can go wrong
catch (IndexOutOfBoundsException e) {
    System.out.println( "Usage: Etest <int> <int>" ); }
catch (NumberFormatException e) {
    System.out.println( e.getMessage() + " is not a
number" ); }

// Do this regardless
finally {
    System.out.println( "That's all, folks" ); }
} // main
} // Etest
```

```
public class Etest {
public static void main(String args[]){
// What we expect to happen
try {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    System.out.println( x + "/" + y + " = " + x/y ); }

// Things which can go wrong
catch (IndexOutOfBoundsException e) {
    System.out.println( "Usage: Etest <int> <int>" ); }
catch (NumberFormatException e) {
    System.out.println( e.getMessage() + " is not a
number" ); }

// Do this regardless
finally {
    System.out.println( "That's all, folks" ); }
} // main
} // Etest
```

```
> java Etest 99 42
99/42 = 2
That's all, folks
```



```
public class Etest {
public static void main(String args[]){
// What we expect to happen
try {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    System.out.println( x + "/" + y + " = " + x/y ); }

// Things which can go wrong
catch (IndexOutOfBoundsException e) {
    System.out.println( "Usage: Etest <int> <int>" ); }
catch (NumberFormatException e) {
    System.out.println( e.getMessage() + " is not a
number" ); }

// Do this regardless
finally {
    System.out.println( "That's all, folks" ); }
} // main
} // Etest
```

```
> java Etest 99
Usage: Etest <int> <int>
That's all, folks
```



```
public class Etest {
public static void main(String args[]){
// What we expect to happen
try {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    System.out.println( x + "/" + y + " = " + x/y ); }

// Things which can go wrong
catch (IndexOutOfBoundsException e) {
    System.out.println( "Usage: Etest <int> <int>" ); }
catch (NumberFormatException e) {
    System.out.println( e.getMessage() + " is not a
number" ); }

// Do this regardless
finally {
    System.out.println( "That's all, folks" ); }
} // main
} // Etest
```

```
> java Etest 99 fred
fred is not a number
That's all, folks
```

```
public class Etest {
public static void main(String args[]){
// What we expect to happen
try {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    System.out.println( x + "/" + y + " = " + x/y ); }

// Things which can go wrong
catch (IndexOutOfBoundsException e) {
    System.out.println( "Usage: Etest <int> <int>" ); }
catch (NumberFormatException e) {
    System.out.println( e.getMessage() + " is not a
number" ); }

// Do this regardless
finally {
    System.out.println( "That's all, folks" ); }
} // main
} // Etest
```

```
> java Etest fred
fred is not a number
That's all, folks
```

```
public class Etest {
public static void main(String args[]){
// What we expect to happen
try {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    System.out.println( x + "/" + y + " = " + x/y ); }

// Things which can go wrong
catch (IndexOutOfBoundsException e) {
    System.out.println( "Usage: Etest <int> <int>" ); }
catch (NumberFormatException e) {
    System.out.println( e.getMessage() + " is not a
number" ); }

// Do this regardless
finally {
    System.out.println( "That's all, folks" ); }
} // main
} // Etest
```

```
> java Etest 99 0
That's all, folks
java.lang.ArithmeticException: / by zero
at Etest.main(Etest.java:8)
```

# Using `finally` for Cleanup

Finalizers aren't much good for releasing resources

To get guaranteed cleanup of network connections etc.

use `finally`

because we don't know when (or even if) they will be called

```
Socket s;  
InputStream in;  
try {  
    s = new Socket(...);    ...  
    in = s.getInputStream(); ...  
}
```

```
finally {  
    try { if (in != null) in.close();  
        s.close();  
    }  
    catch (IOException e) {}  
}
```

So we actually need a `try ... catch` block within the `finally` clause

e.g. if the network goes down at the wrong moment

# With Exception Handling - Example

```
class WithExceptionCatchThrow{  
    public static void main(String[] args){  
        int a,b; float r; a = 7; b = 0;  
        try{  
            r = a/b;  
            System.out.println("Result is " + r);  
        }
```

Program Does Not  
reach here

when exception occurs

```
        catch(ArithmeticException e){  
            System.out.println(" B is zero);  
            throw e;  
        }
```

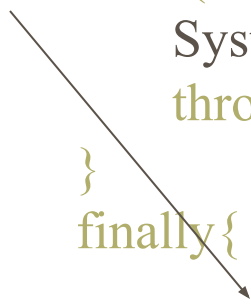
```
        System.out.println("Program is complete");  
    }  
}
```



# With Exception Handling - Example

```
class WithExceptionCatchThrowFinally{
    public static void main(String[] args){
        int a,b; float r; a = 7; b = 0;
        try{
            r = a/b;
            System.out.println("Result is " + r);
        }
        catch(ArithmeticException e){
            System.out.println(" B is zero);
            throw e;
        }
        finally{
            System.out.println("Program is complete");
        }
    }
}
```

Program reaches here

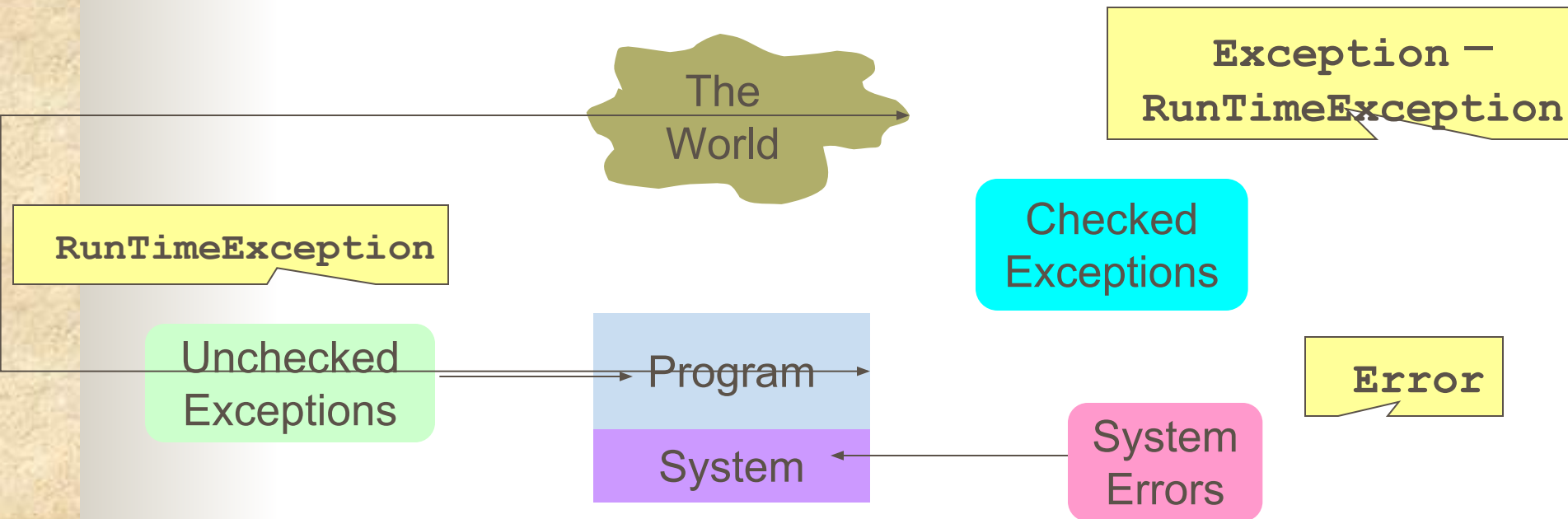
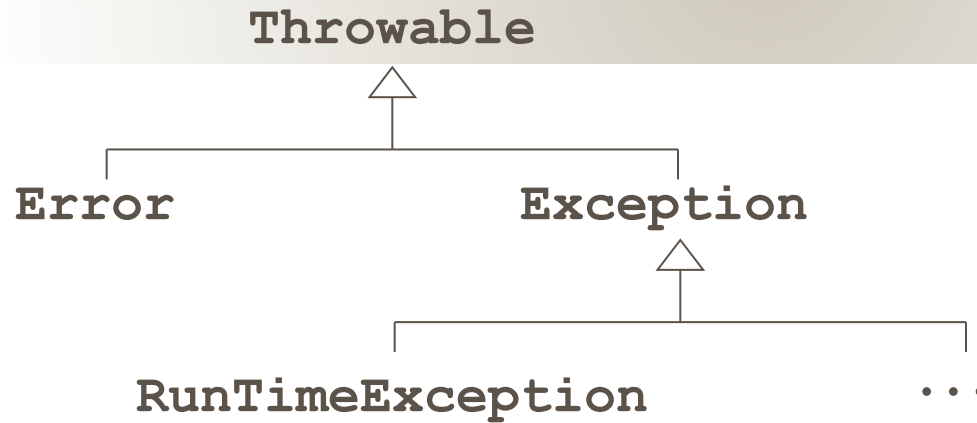




# Exception Types

- All exception types are subclasses of the built-in class **Throwable**
- Throwable has two subclasses, they are
  - Exception (to handle exceptional conditions that user programs should catch)
    - An important subclass of Exception is **RuntimeException**, that includes division by zero and invalid array indexing
  - Error (to handle exceptional conditions that are not expected to be caught under normal circumstances).  
i.e. stack overflow

# Exception Hierarchy in Java





# Representing Exceptions

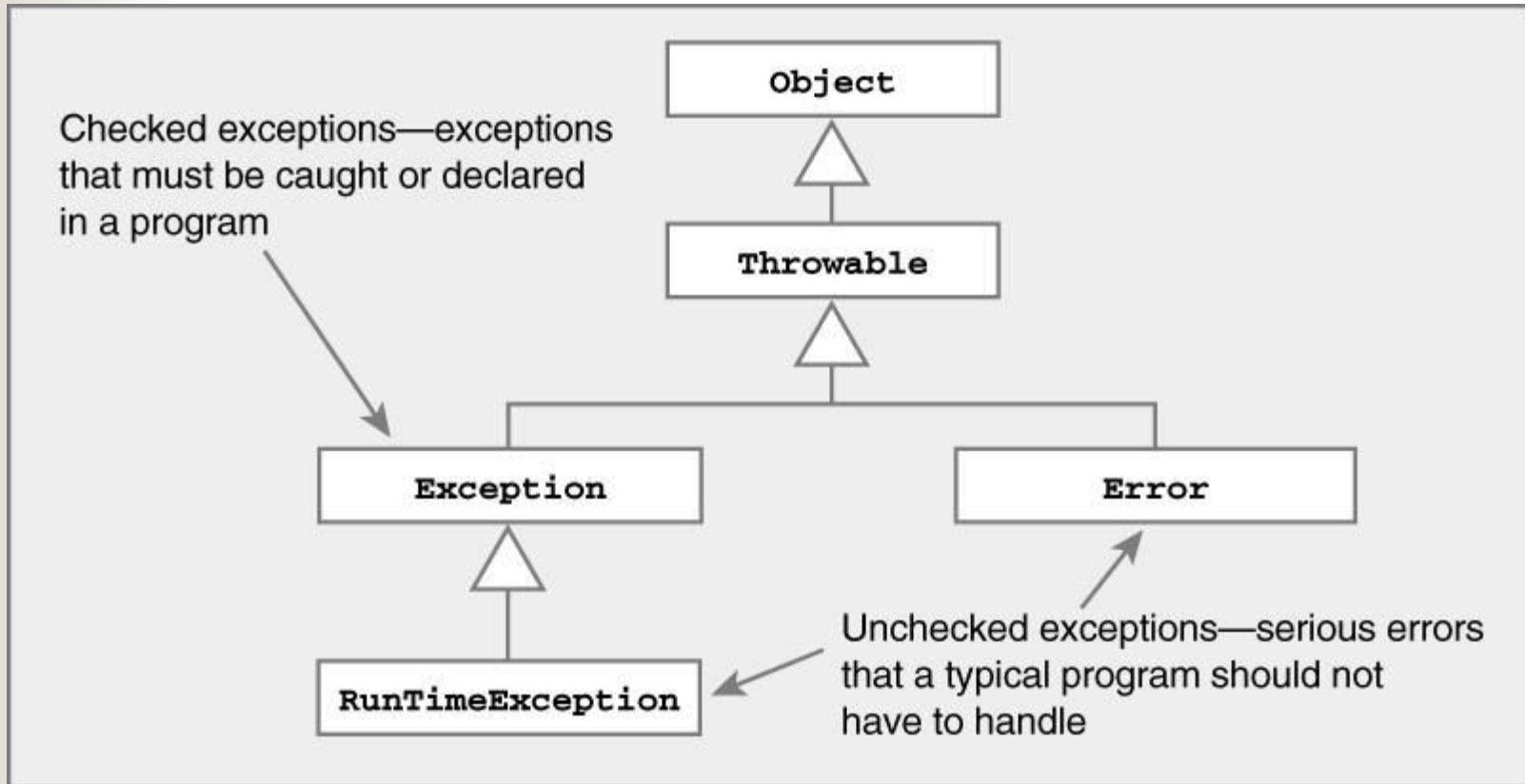
- Exceptions represented as
  - Objects derived from class Throwable

- Code

```
public class Throwable( ) extends Object {  
    Throwable( )                // No error message  
    Throwable( String mesg )    // Error message  
    String getMessage()         // Return error mesg  
    void printStackTrace( ) { ... } // Record methods  
    ...                        // called & location  
}
```

# Representing Exceptions

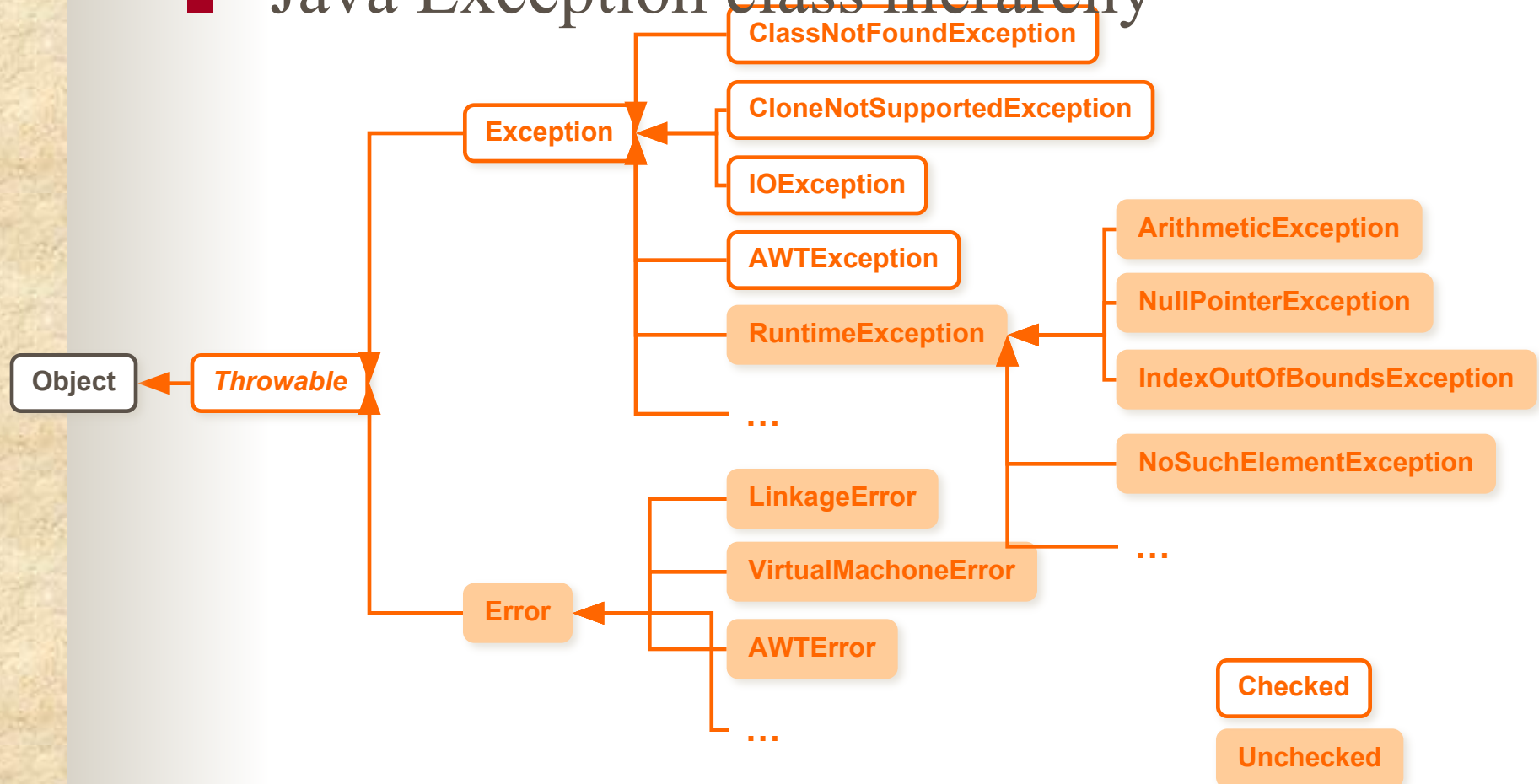
- Java Exception class hierarchy
  - Two types of exceptions  $\Rightarrow$  checked & unchecked





# Representing Exceptions

## ■ Java Exception class hierarchy



# EXCEPTION VS. ERROR

- ❑ An **Error** indicates a serious problem that a reasonable application should not try to catch.
  
- ❑ Examples:
  - ❑ `ClassFormatError`,
  - ❑ `InstantiationError`,
  - ❑ `InternalError`,
  - ❑ `NoSuchMethodError`,
  - ❑ `OutOfMemoryError`,
  - ❑ `StackOverflowError`,
  - ❑ `VirtualMachineError`.

# RuntimeException

- ❑ **RuntimeException** is the superclass of those exceptions that can be thrown during the normal operation of the JVM.
- ❑ Examples:
  - ❑ NullPointerException,
  - ❑ ArrayIndexOutOfBoundsException,
  - ❑ NegativeArraySizeException,
  - ❑ ClassCastException,
  - ❑ NumberFormatException,
  - ❑ SecurityException.



# Caution

- Remember that, exception subclass must come before any of of their superclasses
- Because, a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. So, the subclass would never be reached if it come after its superclass
- For example, **ArithmeticException** is a subclass of **Exception**
- Moreover, unreachable code in Java generates error



# Example

```
/* This program contains an error.
|
| A subclass must come before its superclass in
| a series of catch statements. If not,
| unreachable code will be created and a
| compile-time error will result.
*/
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
            System.out.println("Generic Exception catch.");
        }
        /* This catch is never reached because
           ArithmeticException is a subclass of Exception. */
        catch(ArithmeticException e) { // ERROR - unreachable
            System.out.println("This is never reached.");
        }
    }
}
```





# Nested try Statements

- A **try** statement can be inside the block of another try
- Each time a **try** statement is entered, the context of that exception is pushed on the stack
- If an inner **try** statement does not have a catch, then the next **try** statement's catch handlers are inspected for a match
- If a method call within a **try** block has **try** block within it, then then it is still nested **try**

# Example

```
// An example nested try statements.
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* If no command line args are present,
               the following statement will generate
               a divide-by-zero exception. */
            int b = 42 / a;

            System.out.println("a = " + a);

            try { // nested try block
                /* If one command line arg is used,
                   then an divide-by-zero exception
                   will be generated by the following code. */
                if(a==1) a = a/(a-a); // division by zero

                /* If two command line args are used
                   then generate an out-of-bounds exception. */
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out-of-bounds: " + e);
            }

        } catch(ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```



# Output

- When no parameter is given:

Divide by 0: java.lang.ArithmeticException: / by zero

- When one parameter is given

a = 1

Divide by 0: java.lang.ArithmeticException: / by zero

- When two parameters are given

a = 2

Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException



# throw

- It is possible for your program to to throw an exception explicitly  
*throw ThrowableInstance*
- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass **Throwable**
- There are two ways to obtain a **Throwable** objects:
  - Using a parameter into a catch clause
  - Creating one with the **new** operator



# Example

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch (NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // re-throw the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch (NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

## Output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo



# Java's Built-in Errors

- class java.lang.Throwable (implements java.io.Serializable)
  - class java.lang.Error
    - class java.lang.LinkageError
      - class java.lang.ClassCircularityError
      - class java.lang.ClassFormatError
        - class java.lang.UnsupportedClassVersionError
      - class java.lang.ExceptionInInitializerError
      - class java.lang.IncompatibleClassChangeError
        - class java.lang.AbstractMethodError
        - class java.lang.IllegalAccessError
        - class java.lang.InstantiationError
        - class java.lang.NoSuchFieldError
        - class java.lang.NoSuchMethodError
      - class java.lang.NoClassDefFoundError
      - class java.lang.UnsatisfiedLinkError
      - class java.lang.VerifyError
    - class java.lang.ThreadDeath
    - class java.lang.VirtualMachineError
      - class java.lang.InternalError
      - class java.lang.OutOfMemoryError
      - class java.lang.StackOverflowError
      - class java.lang.UnknownError

•Small letter indicate package name

•Capital letter indicate class name

# Java's Built-in Exceptions

- class java.lang.Throwable (implements java.io.Serializable)
  - class java.lang.Exception
    - class java.lang.ClassNotFoundException
    - class java.lang.CloneNotSupportedException
    - class java.lang.IllegalAccessException
    - class java.lang.InstantiationException
    - class java.lang.InterruptedException
    - class java.lang.NoSuchFieldException
    - class java.lang.NoSuchMethodException
    - class java.lang.RuntimeException
      - class java.lang.ArithmeticException
      - class java.lang.ArrayStoreException
      - class java.lang.ClassCastException
      - class java.lang.IllegalArgumentException
        - class java.lang.IllegalThreadStateException
        - class java.lang.NumberFormatException
      - class java.lang.IllegalMonitorStateException
      - class java.lang.IllegalStateException
      - class java.lang.IndexOutOfBoundsException
        - class java.lang.ArrayIndexOutOfBoundsException
        - class java.lang.StringIndexOutOfBoundsException
      - class java.lang.NegativeArraySizeException
      - class java.lang.NullPointerException
      - class java.lang.SecurityException
      - class java.lang.UnsupportedOperationException





# User-Defined Exceptions

- Problem Statement :
  - Consider the example of the Circle class
  - Circle class had the following constructor

```
public Circle(double centreX, double centreY,  
              double radius){  
    x = centreX; y = centreY; r = radius;  
}
```

- How would we ensure that the radius is not zero or negative?



# Defining your own exceptions

```
import java.lang.Exception;
class InvalidRadiusException extends Exception {

    private double r;

    public InvalidRadiusException(double radius){
        r = radius;
    }
    public void printError(){
        System.out.println("Radius [" + r + "] is not valid");
    }
}
```



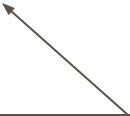
# Throwing the exception

```
class Circle {  
    double x, y, r;  
  
    public Circle (double centreX, double centreY, double  
radius ) throws InvalidRadiusException {  
        if (r <= 0 ) {  
            throw new InvalidRadiusException(radius);  
        }  
        else {  
            x = centreX ; y = centreY; r = radius;  
        }  
    }  
}
```



# User-Defined Exceptions in standard format

```
class MyException extends Exception
{
    MyException(String message)
    {
        super(message); // pass to superclass if parameter is not handled by user defined exception
    }
}
class TestMyException {
...
    try {
        ..
        throw new MyException("This is error message");
    }
    catch(MyException e)
    {
        System.out.println("Message is: "+e.getMessage());
    }
}
}
```



Get Message is a method defined in a standard Exception class.