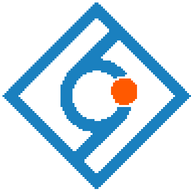


Exception hierarchy

Throwable

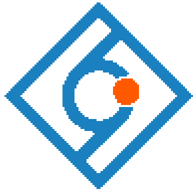
Error

Exception

IOException

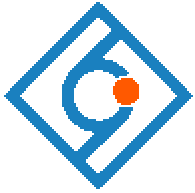
Runtime Exception

Written by Paul Pu All Rights Reservedwww.torontocollege.com



The classification of Exception

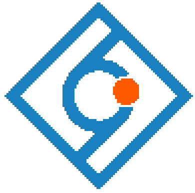
- **The Error** hierarchy describes internal errors and resource exhaustion inside the Java runtime system.. There is little you can do if such an internal error occurs, beyond notifying the user and trying to terminate the program gracefully.
- When doing Java programming, you focus on the Exception hierarchy.
  - A RuntimeException happens because you made a programming error. If it is a RuntimeException, it was your fault. RuntimeException includes such problems as:
    - A bad cast
    - An out-of-bounds array access
    - A null pointer access
  - Other Exceptions: not inherit from RuntimeException. For example: IOException



The classification of Exception

From	To
Error	Unchecked exception
RuntimeException	Unchecked exception
Other exception	Checked exception

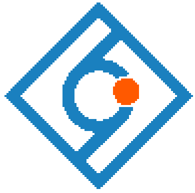
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Throwing Exception

- Like any object in Java, you always create exceptions on the heap using **new**, which allocates storage and calls a constructor. There are two constructors in all standard exceptions: the first is the default constructor, and the second takes a string argument so you can place pertinent information in the exception:
- if(t == null) throw new NullPointerException("t = null");
- This string can later be extracted using various methods, as will be shown later.

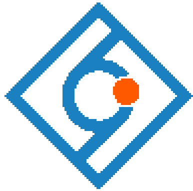
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Throwing Exception

- The keyword **throw** causes a number of relatively magical things to happen. Typically, you'll first use **new** to create an object that represents the error condition. Next you use the throw keyword to actually throw the exception. The two are normally combined a single statement.

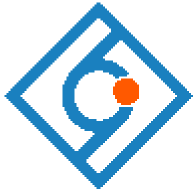
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Throws: The exception specification

- In Java, you’re required to inform the client programmer, who calls your method, of the exceptions that might be thrown from your method. This is civilized because the caller can know exactly what code to write to catch all potential exceptions. Of course, if source code is available, the client programmer could hunt through and look for **throw** statements, but often a library doesn’t come with sources.

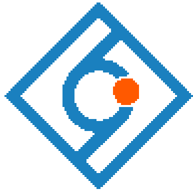
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Throws: The exception specification

- To prevent this from being a problem, Java provides syntax (and *forces* you to use that syntax) to allow you to politely tell the client programmer what exceptions this method throws, so the client programmer can handle them. This is the *exception specification* and it's part of the method declaration, appearing after the argument list.

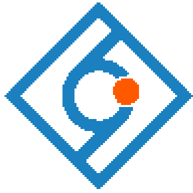
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Throws: the exception specification

```
1 public void doSomeIO (String targetUrl)
2 throws MalformedURLException, EOFException {
3// new URL might throw MalformedURLException
4 URL url=new URL(targetUrl);
5 //open the url and read from it
6 //set flag ‘completed’ when IO is successful
7 //so if we get here with completed= = false,
8 //we got unexpected end of file
9 If(!completed)
10 throw new EOFExcetion (“Invalid file contends);
11 }
12 }
```

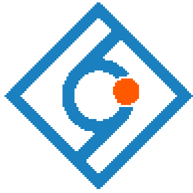
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Throws: the exception specification

- Line 11 demonstrates the use of the **throw** statement
- Line 2 could be shortened to throws IOException. This is because both MalformedURLException and EOFException are subclasses of IOException

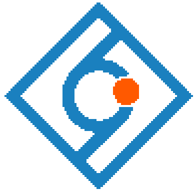
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Flow of Control in Exception Conditions
Using try{} catch {}

```
1 int x=(int) (Math.random()*5);
2 int y=(int)(Math.random()*10);
3 int[] z=new int[5];
4 try {
 5  System.out.println("y/x gives "+(y/x));
 6  System.out.println("y is "
7 +y+"Z[y] is" +z[y]);
8 }
9 catch (ArithmeticException e) {
10 System.out.println(Arithmetic problem"+e);
11 }
12 catch (ArrayIndexOutOfBoundException e) {
 13 System.out.println(Subscript problem"+e);
14 }
```

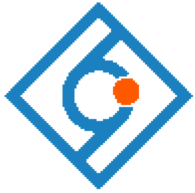
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Flow of Control in Exception Conditions
Using try{} catch {}

- In this example, an Exception is possible at line 5 and at line 6. Line 5 has the potential to cause a division by 0, which in integer arithmetic results in an ArithmeticException being thrown. Line 6 will sometimes throw an ArrayIndexOutOfBoundsException

Written by Paul Pu All Rights Reservedwww.torontocollege.com

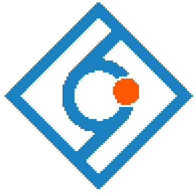


Flow of Control in Exception Conditions
Catching any exception

- It is possible to create a handler that catches any type of exception. You do this by catching the base-class exception type **Exception** (there are other types of base exceptions, but **Exception** is the base that's pertinent to virtually all programming activities):

- catch(Exception e) { System.out.println("caught an exception"); }

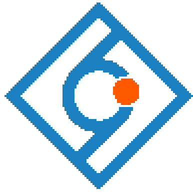
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Flow of Control in Exception Conditions
Catching any exception

- This will catch any exception, so if you use it you'll want to put it at the *end* of your list of handlers to avoid preempting any exception handlers that might otherwise follow it.

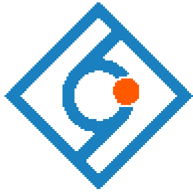
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Flow of Control in Exception Conditions
Using try{} catch {}

- Basic **Exception** methods
  - **String getMessage( )**
    String getLocalizedMessage**( )**
    Gets the detail message, or a message adjusted for this particular locale.
    String toString( )
    Returns a short description of the Throwable, including the detail message if there is one.
    void printStackTrace( )
    void printStackTrace(PrintStream)
    void printStackTrace**(**PrintWriter**)**
    Prints the Throwable and the Throwable's call stack trace. The call stack shows the sequence of method calls that brought you to the point at which the exception was thrown. The first version prints to standard error, the second and third prints to a stream of your choice (in Chapter 11, you'll understand why there are two types of streams).
    Throwable fillInStackTrace**( )**

Written by Paul Pu All Rights Reserved www.torontocollege.com

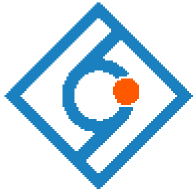


Flow of Control in Exception Conditions
Using try{} catch {}

- Here’s an example that shows the use of the basic **Exception** methods:
  // Demonstrating the Exception Methods.
- public class ExceptionMethods {
- public static void main(String[] args) {
- try { throw new Exception("Here's my Exception");
- } catch(Exception e) { System.out.println("Caught Exception");
  tln("e.getMessage(): " + e.getMessage()); System.out.println(
  intln("e.getLocalizedMessage(): " + e.getLocalizedMessage());
  LocalizedMessage()); System.out.println("e.toString(): " + e);
  stem.out.println("e.printStackTrace():"); e.printStackTrace();
  ce():"); e.printStackTrace(); e.printStackTrace(System.err); }
  ):"); e.printStackTrace(); e.printStackTrace(System.err); }

rintStackTrace(); e.printStackTrace(System.err); }
intStackTrace(); e.printStackTrace(System.err); }

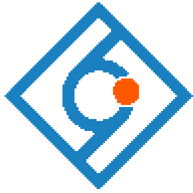
Written by Paul PS All Rights Reservedwww.torontocollege.com



Flow of Control in Exception Conditions Using try{} catch {}

- The output for this program is:

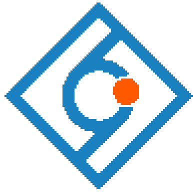
- Caught Exception e.getMessage(): Here's my Exception e.getLocalizedMessage(): Here's my Exception e.toString(): java.lang.Exception: Here's my Exception e.printStackTrace(): java.lang.Exception: Here's my Exception at ExceptionMethods.main(ExceptionMethods.java:7) java.lang.Exception: Here's my Exception at ExceptionMethods.main(ExceptionMethods.java:7)



Creating your own exceptions

- To create your own exception class,you are force to inherit from an existing type of exception.

Written by Paul Pu All Rights Reservedwww.torontocollege.com

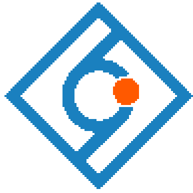


Creating your own exceptions

- // From 'Thinking in Java, 2nd ed.' by Bruce Eckel
- // www.BruceEckel.com. See copyright notice in CopyRight.txt.
- // Inheriting your own exceptions.

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg);
  }
}
```

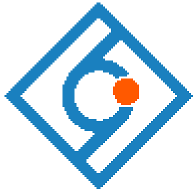
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Creating your own exceptions

```
public class Inheriting {
  public static void f() throws MyException {
    System.out.println(
      "Throwing MyException from f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println(
      "Throwing MyException from g()");
    throw new MyException("Originated in g()");
  }
```

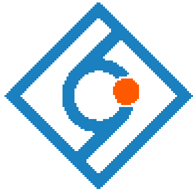
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Creating your own exceptions

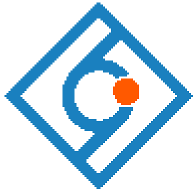
```
public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace();
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace();
    }
  }
} ///:~
```

Written by Paul Pu All Rights Reservedwww.torontocollege.com



Creating your own exceptions

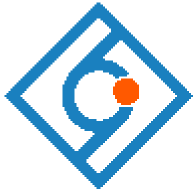
- The process of creating your own exception can be taken further. You can add extra constructors and member:
- `// From 'Thinking in Java, 2nd ed.' by Bruce Eckel`
- `// www.BruceEckel.com. See copyright notice in CopyRight.txt.`
- `// Inheriting your own exceptions.`
- `class MyException2 extends Exception {`
- `  public MyException2() {}`
- `  public MyException2(String msg) {`
- `    super(msg);`
- `  }`
-



Creating your own exceptions

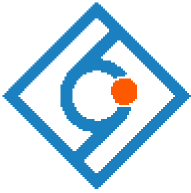
```
public MyException2(String msg, int x) {
      super(msg);
      i = x;
  }
  public int val() { return i; }
  private int i;
}
```

Written by Paul Pu All Rights Reservedwww.torontocollege.com



Creating your own exceptions

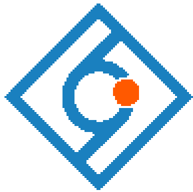
```
public class Inheriting2 {
  public static void f() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from f()");
    throw new MyException2();
  }
  public static void g() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from g()");
    throw new MyException2("Originated in
g()");
  }
```



Creating your own exceptions

```
public static void h() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from h()");
    throw new MyException2(
      "Originated in h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException2 e) {
      e.printStackTrace();
    }

```

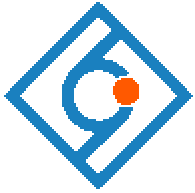
Written by Paul Pu All Rights Reservedwww.torontocollege.com



Creating your own exceptions

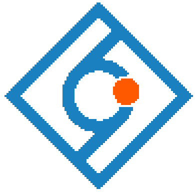
```
try {
      g();
    } catch(MyException2 e) {
      e.printStackTrace();
    }
    try {
      h();
    } catch(MyException2 e) {
      e.printStackTrace();
      System.out.println("e.val() = " + e.val());
    }
  }
} ///:~
```

Written by Paul Pu All Rights Reservedwww.torontocollege.com



Flow of Control in Exception Conditions
Using try{} catch {} finally { }

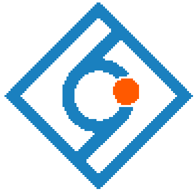
- try{
- code that might throw exceptions
- }
- catch(IOException e)
- {
- Show error dialog
- }
- Finally
- { release resources}



Flow of Control in Exception Conditions
Using try{} catch {} finally { }

- // www.BruceEckel.com. See copyright notice in CopyRight.txt.
- // The finally clause is always executed.

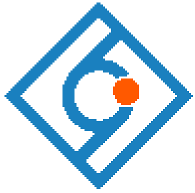
```
public class FinallyWorks {
  static int count = 0;
  public static void main(String[] args) {
    while(true) {

```



Flow of Control in Exception Conditions Using try{} catch {} finally { }

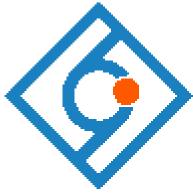
```
try {
            // Post-increment is zero first time:
            if(count++ == 0)
              throw new Exception();
            System.out.println("No exception");
          } catch(Exception e) {
            System.out.println("Exception thrown");
          } finally {
            System.out.println("In finally clause");
            if(count == 2) break; // out of "while"
          }
        }
      }
} ///:~
```

Written by Paul Pu All Rights Reservedwww.torontocollege.com



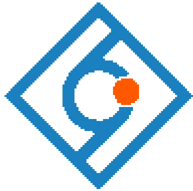
A Final Look at Java Error and Exception Handling

- ExceptTest.java



Which Exceptions Should You Throw?

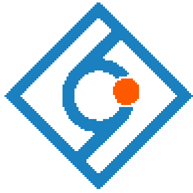
- You don't really have to list all the possible exceptions that your method could throw; some exceptions are handled by the runtime itself and are so common (well, not common, but ubiquitous) that you don't have to deal with them. In particular, exceptions of either class Error or RuntimeException (or any of their subclasses) do not have to be listed in your throws clause. They get special treatment because they can occur anywhere within a Java program and are usually conditions that you, as the programmer, did not directly cause. One good example is OutOfMemoryError, which can happen anywhere, at any time, and for any number of reasons. These two kinds of exceptions are called implicit exceptions, and you don't have to worry about them



Which Exceptions Should You Throw?

- *Implicit exceptions* are exceptions that are subclasses of the classes RuntimeException and Error. Implicit exceptions are usually thrown by the Java runtime itself. You do not have to declare that your method throws them.

Written by Paul Pu All Rights Reservedwww.torontocollege.com



When and When Not to Use Exceptions

- To finish up this section, here's a quick summary and some advice on when to use exceptions…and when not to use them.
- **When to Use Exceptions**

Because throwing, catching, and declaring exceptions are interrelated concepts and can be very confusing, here's a quick summary of when to do what.

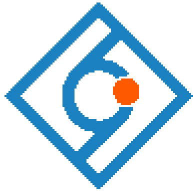
If your method uses someone else's method, and that method has a throws clause, you can do one of three things:

Deal with the exception using try and catch statements.

Pass the exception up the calling chain by adding your own throws clause to your method definition.

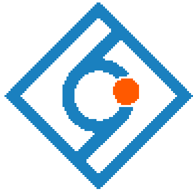
Do both of the above by catching the exception using catch and then explicitly rethrowing it using throw.

Written by Paul Pu All Rights Reservedwww.torontocollege.com



When and When Not to Use Exceptions

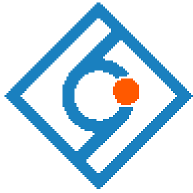
- In cases where a method throws more than one exception, you can, of course, handle each of those exceptions differently. For example, you might catch some of those exceptions while allowing others to pass up the calling chain.
- If your method throws its own exceptions, you should declare that it throws those methods using the throws clause. If your method overrides a superclass's method that has a throws clause, you can throw the same types of exceptions or subclasses of those exceptions; you cannot throw any different types of exceptions.



When and When Not to Use Exceptions

- And, finally, if your method has been declared with a throws clause, don't forget to actually throw the exception in the body of your method using throw.

Written by Paul Pu All Rights Reservedwww.torontocollege.com



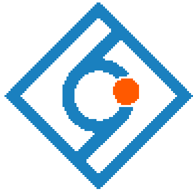
When and When Not to Use Exceptions

- **When Not to Use Exceptions**

Exceptions are cool. But they aren't *that* cool. There are several cases in which you should not use exceptions, even though they may seem appropriate at the time.

you should not use exceptions if the exception is something that you expect and a simple test to avoid that exceptional condition would make much more sense. For example, although you can rely on an ArrayIndexOutofBounds exception to tell you when you've gone past the end of the array, a simple test of the length of the array in your code to make sure you don't get that exception in the first place is a much better idea.

Written by Paul Pu All Rights Reservedwww.torontocollege.com



When and When Not to Use Exceptions

- Exceptions take up a lot of processing time for your Java program. Whereas you may find exceptions stylistically interesting for your own code, a simple test or series of tests will run much faster and make your program that much more efficient. Exceptions should only be used for truly exceptional cases that are out of your control