

Exceptions in Java

What is an Exception

- o Imagine the following code in some Applet:

```
int age = Integer.parseInt(ageTF.getText().trim());
```
- o Obviously, we are expecting a number will appear in the TextField and that it constitutes a legal integer age.
- o Consider, however, the following possibilities:
 1. What if the user types a "\$" instead of a 4 by mistake?
 2. What if the user enters a decimal point number rather than an integer?
 3. What if the user holds down the "3" key too long and an extremely long number is accidentally entered?
- o We do not expect circumstances such as these -- but they do happen!

2

What is an Exception

- o Some things that can go wrong during the execution of a program (such as on the earlier slide) cannot be detected at compile-time — because the user has not yet made the mistake by entering the wrong data!!
- o Another example: your program may attempt to divide one number by zero (ex. examSum/numberOfStudents)
- o Or your program may require that an integer value be entered into a TextField, and the user of the program enters a float value or some other illegal character.
- o From the compiler's point of view, there is nothing wrong with these statements, and problems will arise only when the program is actually executing.
- o At that point an internal alarm goes off, and Java attempts to "**throw an exception**" signifying that something untoward has occurred.

3

Example

```
import java.awt.*;
import java.applet.*;

public class TrivialApplet extends Applet
{
    // Deliberately divides by zero to produce an exception.
    public void init()
    {
        int numerator = 10;
        int denominator = 0;
        System.out.println ("This text will be printed.");
        System.out.println (numerator/denominator);
        System.out.println ("This text will not be printed.");
        // because exception occurs prior to execution of this
    }
}
```

Note also: There is no code to handle the exception, if it occurs!

4

What is an Exception

- The system then immediately halts its normal mode of execution and goes off looking for help.
- With luck, the system will find some code in your program that will **catch** the exception and deal with it.
- Once caught, the alarm is silenced and the system picks up execution at a location after the block that contained the offending statement.
 - Java has its own terminology for exceptions.
 - *Exceptions are indicted by being thrown, and are detected elsewhere by being caught.*

5

Terminology of Exceptions

- An *exception* is an object that describes an unusual or erroneous situation
- Exceptions are thrown by a program, and may be caught and handled by another part of the program
- A program can be separated into a normal execution flow and an exception execution flow
- An error is also represented as an object in Java, but usually represents an unrecoverable situation and should not be caught

6

Some Possible Exceptions

<i>ArithmeticException</i>	- something, such as division by zero, has gone wrong in an arithmetic expression
<i>NumberFormatException</i>	- indicates that an illegal number format is being used.
<i>StringIndexOutOfBoundsException</i>	- an attempt has been made to use an inappropriate String index.
<i>NullPointerException</i>	- a class method is being called by an object instance that is currently null.
<i>EOFException</i>	- an end-of-file mark has been seen.
<i>IllegalArgumentException</i>	- a method has been called with an invalid argument.
<i>IndexOutOfBoundsException</i>	- an index into an array is out of bounds.

7

Exceptions

- As indicated on the earlier slide, Java has a predefined set of exceptions and errors that can occur during execution
- A program can deal with an exception in one of three ways:
 - ✓ ignore it
 - ✓ handle it where it occurs
 - ✓ handle it in another place in the program
- The manner in which an exception is processed is an important design consideration

8

Lexicon: Actors and Actions

- **Operation**
A method which can possibly raise an exception.
- **Invoker**
A method which calls operations and handles resulting exceptions.
- **Exception**
A concise, complete description of an abnormal event.
- **Raise**
Brings an exception from the operation to the invoker, called `throw` in Java.
- **Handle**
Invoker's response to the exception, called `catch` in Java.
- **Backtrack**
Ability to unwind the stack frames from where the exception was raised to the first matching handler in the call stack.

9

Classifying Java Exceptions

- **Unchecked Exceptions**
It is not required that these types of exceptions be caught or declared on a method.
 - *Runtime exceptions* can be generated by methods or by the JVM itself.
 - *Errors* are generated from deep within the JVM, and often indicate a truly fatal state.
 - Runtime exceptions are a source of major controversy!
- **Checked Exceptions**
Must either be caught by a method or declared in its signature.
 - Placing exceptions in the method signature generates major complications
 - This requirement is viewed with derision in the hardcore C++ community.
 - A common technique for simplifying checked exceptions is subsumption.

10

Keywords for Java Exceptions

- **throws**
Describes the exceptions which can be raised by a method.
- **throw**
Raises an exception to the first available handler in the call stack, unwinding the stack along the way.
- **try**
Marks the start of a block associated with a set of exception handlers.
- **catch**
If the block enclosed by the try generates an exception of this type, control moves here; watch out for implicit subsumption.
- **finally**
Always called when the try block concludes, and after any necessary catch handler is complete.

11

General Syntax

```
public void setProperty(String p_strValue) throws
    NullPointerException {
    if (p_strValue == null) { throw new
        NullPointerException("..."); }
}
public void myMethod() {
    MyClass oClass = new MyClass();
    try {
        oClass.setProperty("foo");
        oClass.doSomeWork();
    } catch (NullPointerException npe) {
        System.err.println("Unable to set property:
            "+npe.toString());
    } finally {
        oClass.cleanup();
    }
}
```

12

Canonical Example 1

```
public void foo() {
    try { /* marks the start of a try-catch block */
        int a[] = new int[2];
        a[4] = 1; /* causes a runtime exception due
to the index */
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("exception: " +
            e.getMessage());
        e.printStackTrace();
    }
}
```

13

Canonical Example 2

```
/* This code also compiles, but throws an
exception at runtime! It is both less obvious
and more common (an off-by-one-error). */
public int[] bar() {
    int a[] = new int[2];
    for (int x = 0; x <= 2; x++) { a[x] = 0; }
    return a;
}
```

14

throw(s) Keyword

```
/* The IllegalArgumentException is considered unchecked,
and remains so even making it part of the signature */
public void setName(String p_strName) throws
    IllegalArgumentException
{
    /* valid names cannot be zero length */
    if (p_strName.length() == 0) {
        throw new IllegalArgumentException("...");
    }
    m_strName = p_strName;
}

public void foo() {
    setName(""); /* No warning about unhandled exceptions.
*/
}
```

15

throw(s) Keyword, part 2

```
/* Make a bad parameter exception class */
class NuttyParameterException extends Exception { ... }

/* To really make an invoker pay attention, use a checked
exception type rather than a Runtime Exception type, but
you must declare that you will throw the type! */
public void setName(String p_strName) /* error here! */
{
    /* valid names cannot be zero length */
    if (p_strName == null || p_strName.length() == 0) {
        throw new NuttyParameterException("...");
    }
    m_strName = p_strName;
}
```

16

throw(s) Keyword, part 3

```
/* Make a bad parameter exception class */
class NuttyParameterException extends Exception { ... }

/* To really make an invoker pay attention, use a checked
exception type rather than a Runtime Exception type. */
public void setName(String p_strName) throws
    NuttyParameterException
{
    /* valid names cannot be zero length */
    if (p_strName == null || p_strName.length() == 0) {
        throw new NuttyParameterException("...");
    }
    m_strName = p_strName;
}
public void foo() {
    setName(""); /* This does result in an error. */
}
```

17

try Keyword

```
/* The try statement marks the position of the first bytecode
instruction protected by an exception handler. */
try {

    UserRecord oUser = new UserRecord();
    oUser.setName("Fred Stevens");
    oUser.store();

    /* This catch statement then marks the final bytecode instruction
    protected, and begins the list of exceptions handled. This info
    is collected and is stored in the exception table for the
    method. */
} catch (CreateException ce) {
    System.err.println("Unable to create user record in the
database.");
}
```

18

catch Keyword

```
/* A simple use of a catch block is to catch the exception raised
by the code from a prior slide. */
try {
    myObject.setName("foo");
} catch (NuttyParameterException npe) {
    System.err.println("Unable to assign name: " + npe.toString());
}

try { /* example 2 */
    myObject.setName("foo");
} catch (NuttyParameterException npe) { /* log and relay this
problem. */
    System.err.println("Unable to assign name: " + npe.toString());
    throw npe;
}
```

19

catch Keyword, part 2

```
/* Several catch blocks of differing types can be concatenated. */
try {
    URL myURL = new URL("http://www.mainejug.org");
    InputStream oStream = myURL.openStream();
    byte[] myBuffer = new byte[512];
    int nCount = 0;

    while ((nCount = oStream.read(myBuffer)) != -1) {
        System.out.println(new String(myBuffer, 0, nCount));
    }
    oStream.close();
} catch (MalformedURLException mue) {
    System.err.println("MUE: " + mue.toString());
} catch (IOException ioe) {
    System.err.println("IOE: " + ioe.toString());
}
```

20

finally Keyword

```
URL myURL = null;
InputStream oStream = null;

/* The prior sample completely neglected to discard the network
resources */
try {
    /* Imagine you can see the code from the last slide here... */
} finally { /* What two things can cause a finally block to be
missed? */
    /* Since we cannot know when the exception occurred, be
careful! */
    try {
        oStream.close();
    } catch (Exception e) {
    }
}
}
```

21

finally Keyword, part 2

```
public bool anotherMethod(Object myParameter) {  
  
    try { /* What value does this snippet return? */  
        myClass.myMethod(myParameter);  
        return true;  
    } catch (Exception e) {  
        System.err.println("Exception in anotherMethod()  
            "+e.toString());  
        return false;  
    } finally {  
        /* If the close operation can raise an exception */  
        if (myClass.close() == false) {  
            break;  
        }  
    }  
    return false;  
}
```

22

finally Keyword, part 3

```
public void callMethodSafely() {  
  
    while (true) { /* How about this situation? */  
        try {  
            /* Call this method until it returns false. */  
            if (callThisOtherMethod() == false) {  
                return;  
            }  
        } finally {  
            continue;  
        }  
    } /* end of while */  
}
```

23

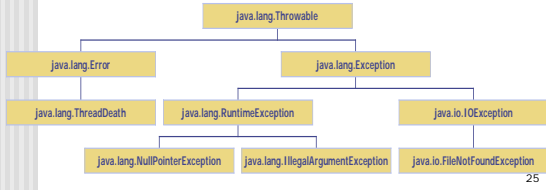
Steps of try...catch...finally

- Every try block must have at least one catch or finally block attached.
- If an exception is raised during a try block:
 - The rest of the code in the try block is skipped over.
 - If there is a catch block of the correct, or derived, type **in this stack frame** it is entered.
 - If there is a finally block, it is entered.
 - If there is no such block, the JVM moves up one stack frame.
- If no exception is raised during a try block, and there is no System.exit() statement:
 - If there is a matching finally block it is entered.

24

Java Exception Hierarchy

Throwable	The base class for all exceptions, it is required for a class to be the rvalue to a throw statement.
Error	Any exception so severe it should be allowed to pass uncaught to the Java runtime.
Exception	Anything which should be handled by the invoker is of this type, and all but five exceptions are.



Creating your own exception class

```

/* You should extend RuntimeException to create an unchecked
exception, or Exception to create a checked exception. */
class MyException extends Exception {

    /* The common constructor. It takes a text argument. */
    public MyException(String p_strMessage) {
        super(p_strMessage);
    }

    /* A default constructor is also a good idea! */
    public MyException () {
        super();
    }

    /* If you create a more complex constructor, then it is
critical that you override toString(), since this is the call
most often made to output the content of an exception. */
}
    
```

26

Three Critical Decisions

- How do you decide to raise an exception rather than return?
 1. Is the situation truly out of the ordinary?
 2. Should it be impossible for the caller to ignore this problem?
 3. Does this situation render the class unstable or inconsistent?
- Should you reuse an existing exception or create a new type?
 1. Can you map this to an existing exception class?
 2. Is the checked/unchecked status of mapped exception acceptable?
 3. Are you masking many possible exceptions for a more general one?
- How do you deal with subsumption in a rich exception hierarchy?
 1. Avoid throwing a common base class (e.g. IOException).
 2. Never throw an instance of the Exception or Throwable classes.

27

An Example of Return v. Raise

```
try {
    InputStream oStream = new
    URL("http://www.mainejug.org").openStream();
    byte[] myBuffer = new byte[512];
    StringBuffer sb = new StringBuffer();
    int nCount = 0;

    while ((nCount = oStream.read(myBuffer)) != -1) {
        sb.append(new String(myBuffer));
    }
    oStream.close();
    return sb.toString(); /* if sb.length() == 0 NOT exception */

    /* These are certainly exceptional conditions. */
} catch (MalformedURLException mue) {
    throw mue;
} catch (IOException ioe) {
    throw ioe;
}
```

28

Mapping to an Exception Class

- When you attempt to map your situation onto an existing Exception class consider these suggestions:
 - Avoid using an unchecked exception, if it is important enough to explicitly throw, it is important enough to be caught.
 - Never throw a base exception class if you can avoid it: **RuntimeException**, **IOException**, **RemoteException**, etc.
 - There is no situation which should cause you to throw the Exception or Throwable base classes. **Never**.

29

Using Unchecked Exceptions

- Use unchecked exceptions to indicate a broken contract:

```
public void setName(String p_strName) {
    /* This is a violated precondition. */
    if (p_strName == null || p_strName.length() == 0) {
        throw new IllegalArgumentException("Name parameter
        invalid!");
    }
}
```

- Be careful about creating a type *derived* from **RuntimeException**.
- A class derived from **AccessControlException** is implicitly unchecked because its parent class derives from **RuntimeException**.

30
