

Exceptions

Lecture 15 Object-Oriented Programming

Definition

- An *exception* represents an error condition that can occur during the normal course of program execution.
- When an exception occurs, or is *thrown*, the normal sequence of flow is terminated. The exception-handling routine is then executed; we say the thrown exception is *caught*.

Not Catching Exceptions

```
String inputStr;
int    age;

inputStr = JOptionPane.showInputDialog(null, "Age:");
age      = Integer.parseInt(inputStr);
```

Error message for invalid input

```
java.lang.NumberFormatException: ten
    at java.lang.Integer.parseInt(Integer.java:405)
    at java.lang.Integer.parseInt(Integer.java:454)
    at Ch8Sample1.main(Ch8Sample1.java:20)
```

Exceptions in Java

- Process for handling exceptions
 - *try* some code
 - *catch* exception thrown by tried code
 - *finally*, clean up if necessary
 - **try**, **catch**, and **finally** are reserved words
- **try** denotes code that may throw exception
 - place questionable code within a *try block*
 - a *try block* must be immediately followed by a *catch block*.
 - A *catch block* must be preceded by a *try block*

Exceptions in Java

- **catch** exception thrown in **try** block and write special code to handle it
 - catch blocks distinguished by **type** of exception
 - can have several *catch blocks*, each specifying a particular type of exception possibly thrown in *try block*
 - once exception is handled, execution continues after the catch block in caller
- **finally** (optional)
 - special block of code that is executed whether or not exception is thrown
 - follows *catch block*

Lecture 15

Object-Oriented Programming

5

Try catch Block

- **try** block enclosed in curly braces `{ }`
- **catch** block mirrors method definition
 - takes exception as formal parameter
- **catch** block based on type of exception parameter it handles
 - most specific exception type in an exception hierarchy *must* lexically come first
 - formal parameter of type `java.lang.Exception` is the most general and would catch any subclass from the exception library

Lecture 15

Object-Oriented Programming

6

Try catch Block Syntax

- Here's the basic syntax (typically in sender):

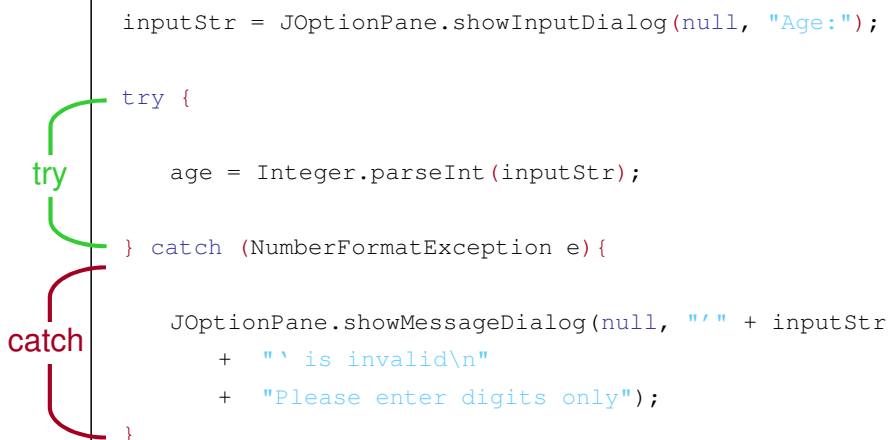
```
try {
    <code>
}
catch (most_specific_exception_type name) {
    <code in response to exception>
}
catch (more_general_exception_type name) {
    <code in response to exception>
}
...
finally { <code> }
```

Lecture 15

Object-Oriented Programming

7

Catching an Exception



```
inputStr = JOptionPane.showInputDialog(null, "Age:");

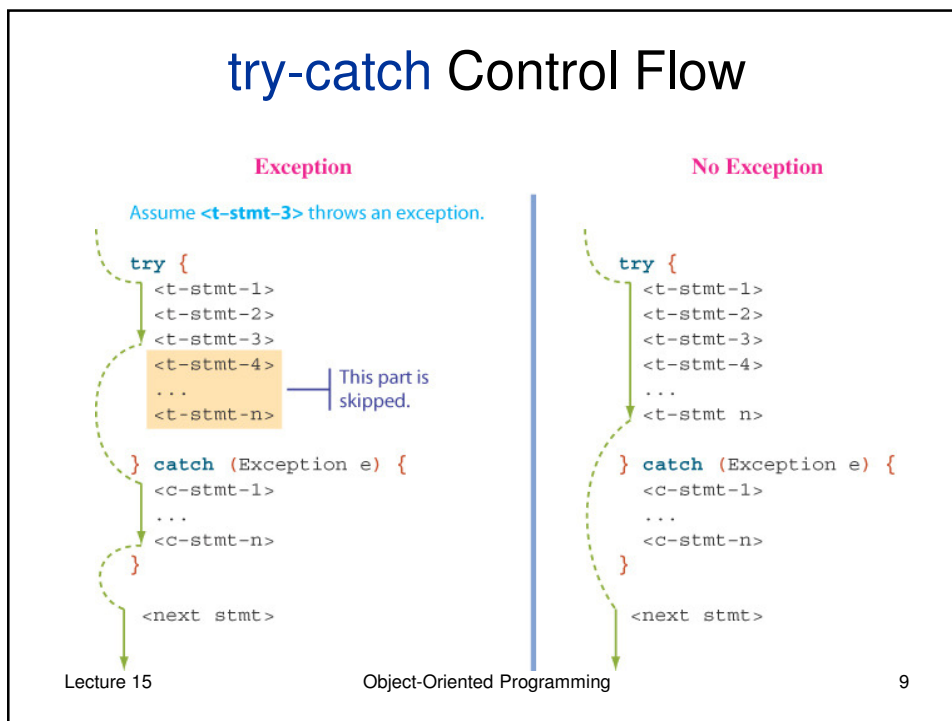
try {
    age = Integer.parseInt(inputStr);
} catch (NumberFormatException e) {
    JOptionPane.showMessageDialog(null, "" + inputStr
        + "` is invalid\n"
        + "Please enter digits only");
}
```

Lecture 15

Object-Oriented Programming

8

try-catch Control Flow



Getting Information

- There are two methods we can call to get information about the thrown exception:
 - **getMessage**
 - **printStackTrace**

```
try {
    . . .
} catch (NumberFormatException e) {

    System.out.println(e.getMessage());
    System.out.println(e.printStackTrace());
}
```

0

Multiple catch Blocks

- A single try-catch statement can include multiple catch blocks, one for each type of exception.

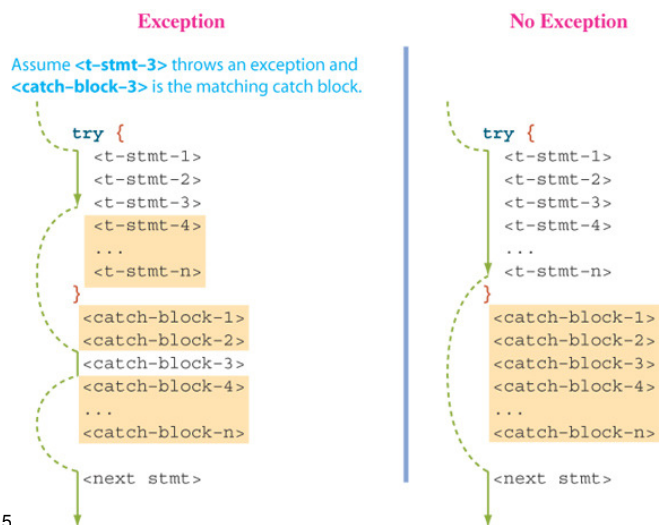
```
try {
    . . .
    age = Integer.parseInt(inputStr);
    . . .
    val = cal.get(id); //cal is a GregorianCalendar
    . . .
} catch (NumberFormatException e) {
    . . .
} catch (ArrayIndexOutOfBoundsException e) {
    . . .
}
```

Lecture 15

Object-Oriented Programming

1

Multiple catch Control Flow



The `finally` Block

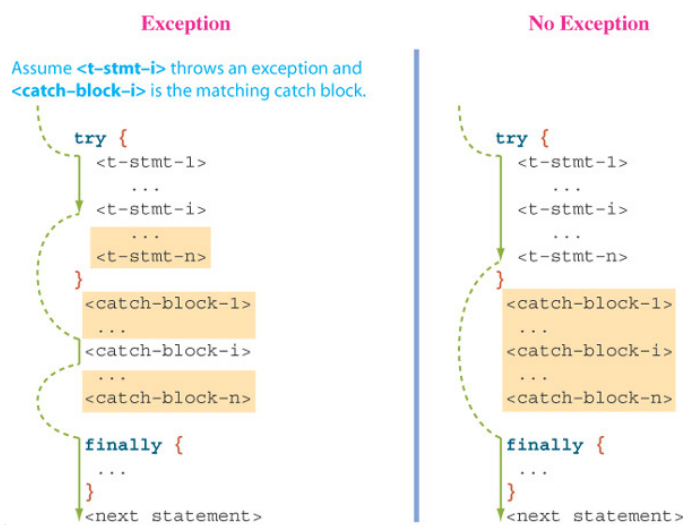
- There are situations where we need to take certain actions regardless of whether an exception is thrown or not.
- We place statements that must be executed regardless of exceptions in the `finally` block.

Lecture 15

Object-Oriented Programming

13

`try-catch-finally` Control Flow



Lecture 1:

Skipped portion

14

Propagating Exceptions

- Instead of catching a thrown exception by using the try-catch statement, we can propagate the thrown exception back to the caller of our method.
- The method header includes the reserved word **throws**.

```
public int getAge( ) throws NumberFormatException {  
    . . .  
    int age = Integer.parseInt(inputStr);  
    . . .  
    return age;  
}
```

Throwing Exceptions

- We can write a method that throws an exception directly, i.e., this method is the origin of the exception.
- Use the **throw** reserved to create a new instance of the Exception or its subclasses.
- The method header includes the reserved word **throws**.

```
public void doWork(int num) throws Exception {  
    . . .  
    if (num != val) throw new Exception("Invalid val");  
    . . .  
}
```


Exception Thrower

- When a method may throw an exception, either directly or indirectly, we call the method an *exception thrower*.
- Every exception thrower must be one of two types:
 - catcher.
 - propagator.

Lecture 15

Object-Oriented Programming

17

Types of Exception Throwers

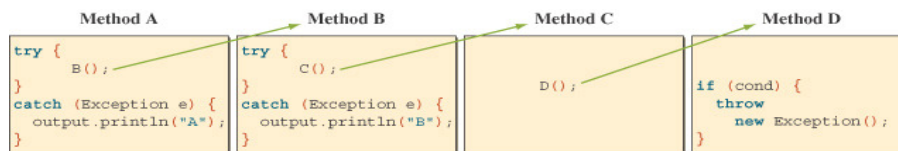
- An *exception catcher* is an exception thrower that includes a matching **catch** block for the thrown exception.
- An *exception propagator* does not contain a matching **catch** block.
- A method may be a catcher of one exception and a propagator of another.

Lecture 15

Object-Oriented Programming

18

Sample Call Sequence



Call Sequence



Stack Trace



Lecture 15

Object-Oriented Programming

19

Exception Types

- All types of thrown errors are instances of the **Throwable** class or its subclasses.
- Serious errors are represented by instances of the **Error** class or its subclasses.
- Exceptional cases that common applications should handle are represented by instances of the **Exception** class or its subclasses.

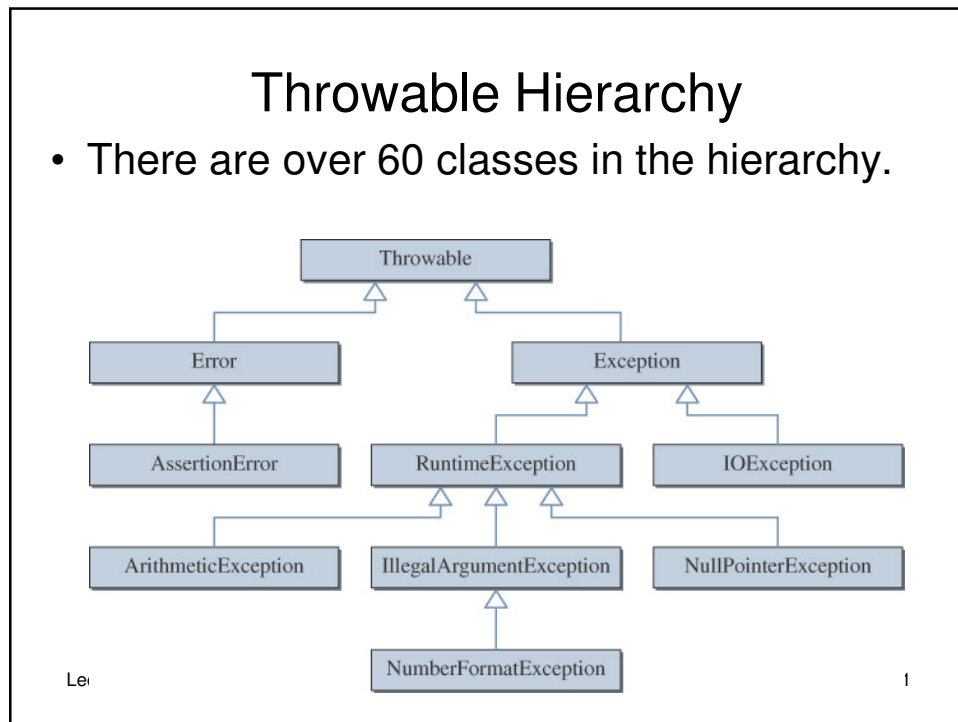
Lecture 15

Object-Oriented Programming

20

Throwable Hierarchy

- There are over 60 classes in the hierarchy.



Checked vs. Runtime

- There are two types of exceptions:
 - Checked.
 - Unchecked.
- A *checked exception* is an exception that is checked at compile time.
- All other exceptions are *unchecked*, or *runtime, exceptions*. As the name suggests, they are detected only at runtime.

Different Handling Rules

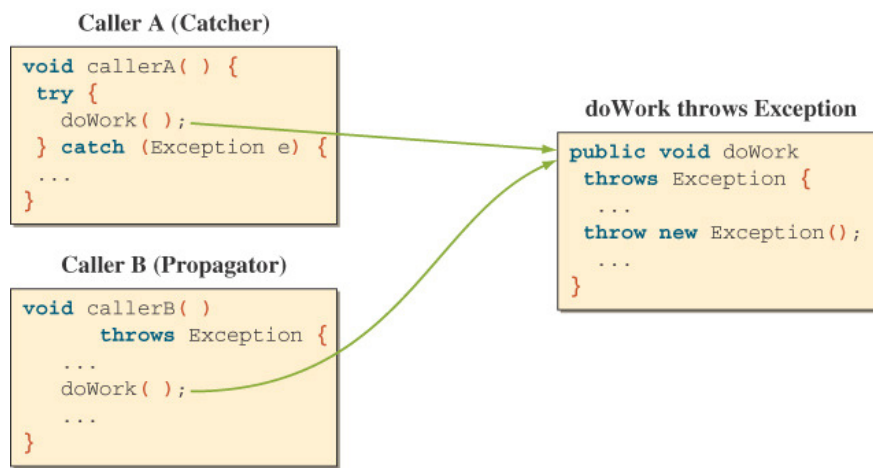
- When calling a method that can throw checked exceptions
 - use the **try-catch** statement and place the call in the **try** block, or
 - modify the method header to include the appropriate **throws** clause.
- When calling a method that can throw runtime exceptions, it is optional to use the try-catch statement or modify the method header to include a throws clause.

Lecture 15

Object-Oriented Programming

23

Handling Checked Exceptions

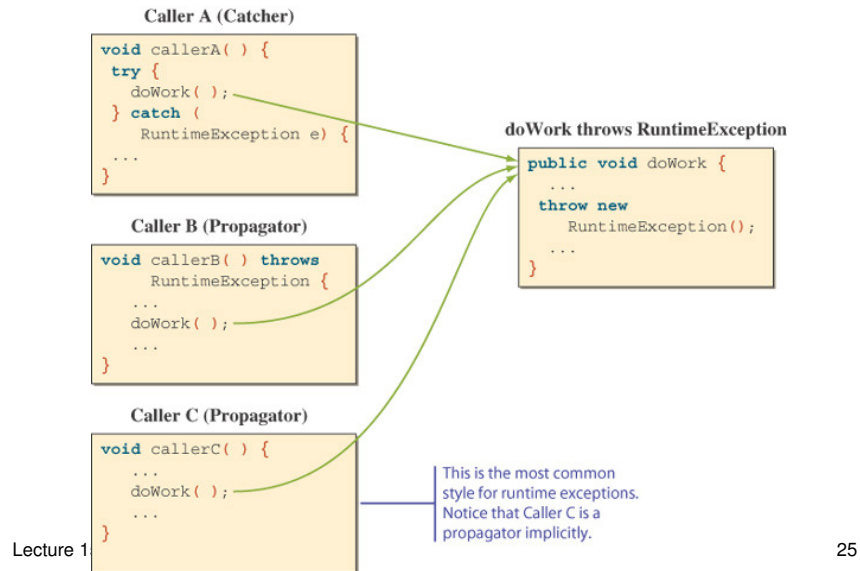


Lecture 15

Object-Oriented Programming

23

Handling Runtime Exceptions



Programmer-Defined Exceptions

- Using the standard exception classes, we can use the `getMessage` method to retrieve the error message.
- By defining our own exception class, we can pack more useful information
 - for example, we may define a `OutOfStock` exception class and include information such as how many items to order
- `AgeInputException` is defined as a subclass of `Exception` and includes public methods to access three pieces of information it carries: lower and upper bounds of valid age input and the (invalid) value entered by the user.

- These slides are developed wholly by C Thomas Wu of Naval Postgraduate College. They are used in this course with minor modifications under Creative Commons License.