Exception Handling

Overview

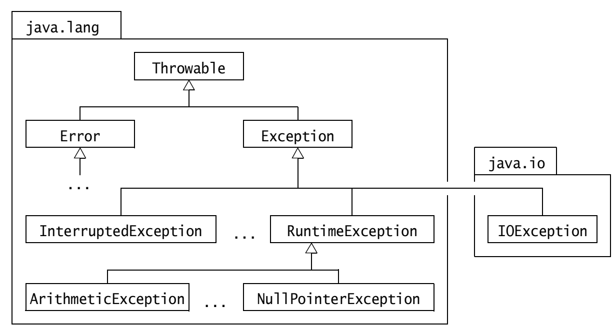
Exception Handling:

- Exception Hierarchy
- Exceptions: checked and unchecked
- `try-catch-finally` block
- `throw`-statement
- `throws`-clause
- Scenarios through the `try-catch-finally` block
- Exception and Inheritance

Exceptions

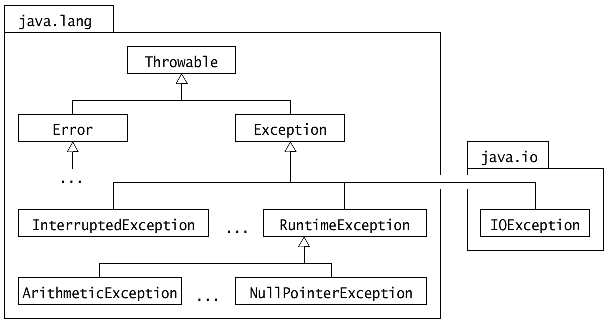
- Exceptions are *runtime* errors which indicate *exceptional conditions*.
- These exceptional conditions are usually not part of the primary behavior of a program.
- Examples of such exceptions are:
  - index out of bounds, division by zero, illegal parameter
  - file not found, read or write error on a file, using `null`-literal to reference an object.
  - network link failure, security violation
- Exception handling is based on the "*throw and catch*" paradigm.
  - an exception is thrown when a situation warrants it, and is caught by an *exception handler.*

- If an exception is not handled by any exception handler in the program, it is handled by the *default exception handler* (i.e. the runtime system).
  - For terminal-based applications, the program is aborted after the exception has been handled by the default exception handler.
  - For GUI-based applications, the program continues after the exception has been dealt with by the default exception handler — the user interaction is handled by a different thread (the *AWT* thread).
  - In all cases, the default exception handler prints the *trace* of the *method activation stack*.



Exception Hierarchy

- Exceptions are objects of classes which extend the class `Throwable`.
- New exception types are created by extending (subclasses of) the `Exception` class.
- A checked exception is an exception which is not `RuntimeException` or `Error` (or a subclass of these two classes).



Example: Default Exception Handling

```
1.  // DivisionByZero1.java
2.
3.  public class DivisionByZero1 {
4.
5.      public static void division() {
6.          int num1 = 10;
7.          int num2 = 0;
8.          System.out.println(num1 + " / " + num2 + " = " + (num1/num2));
9.      }
10.
11.     public static void main(String args[]) {
12.         division();
13.     }
14. }
```

Operation which can lead to an exception of type `ArithmeticException` being thrown.

Information about the exception.

Output on the terminal:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DivisionByZero1.division(DivisionByZero1.java:8)
    at DivisionByZero1.main(DivisionByZero1.java:12)
```

Trace of Method Activation Stack.

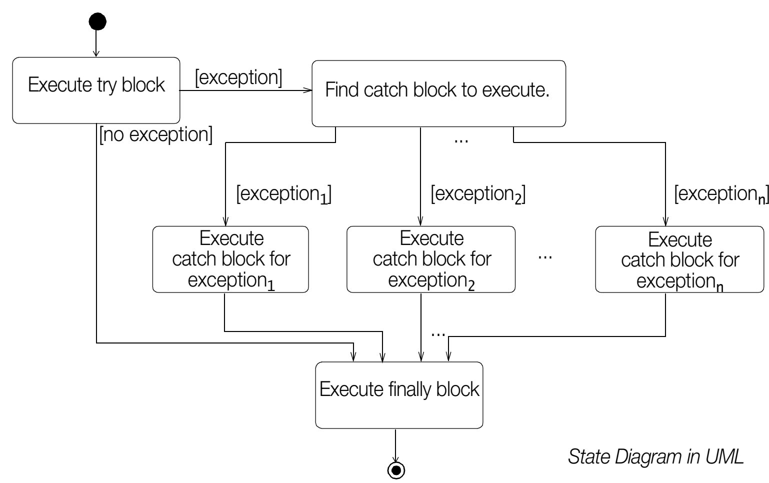
- Exception `java.lang.ArithmeticException` is thrown in line 8, and handled by the default exception handler which dumps the contents of the method activation stack.

`try-catch-finally` Construct

try {   // try block
  <*statements*>
} catch (<*exception type*$_1$> <*parameter*$_1$>**) {**   // catch block
  <*statements*>
}
...
  catch (<*exception type*$_n$> <*parameter*$_n$>**) {**   // catch block
  <*statements*>
} finally {   // finally block
  <*statements*>
}

The `catch` block and the `finally` block are optional, but at least one of them must be specified with a `try` block.

`try-catch-finally` Execution



Example: Explicit Throwing and Catching of an Exception

```
// DivisionByZero3.java
public class DivisionByZero3 {
    public static void division() {
        int num1 = 10;
        int num2 = 0;

        try {
            if (num2 == 0) throw new ArithmeticException();
           System.out.println(num1 + " / " + num2 + " = " + (num1 / num2));
        }
        catch (ArithmeticException Exception) {
            System.out.println(Exception);
        }
    }
    public static void main(String args[]) {
        division();
    }
}
```

An exception can be thrown using the `throw`-statement. In which case, rest of the `try`-block is not executed.

Output on the terminal:

```
java.lang.ArithmeticException
```


Example: Explicit Definition, Throwing and Handling of Exception

```
// DivisionByZero4.java
public class DivisionByZero4 {
    public static void division() {
        int num1 = 10;
        int num2 = 0;
        try {
            if (num2 == 0) throw new DivisionByZero();
            System.out.println(num1 + " / " + num2 + " = " + (num1 / num2));
        }
        catch (DivisionByZero Exception) {
            System.out.println(Exception);
        }
    }
    public static void main(String args[]) {
        division();
    }
}
class DivisionByZero extends Exception {  // Checked Exception.
    DivisionByZero() { super("Integer division by 0"); }
}
```

Output on the terminal:

```
DivisionByZero: Integer division by 0
```


"Checked" Exceptions

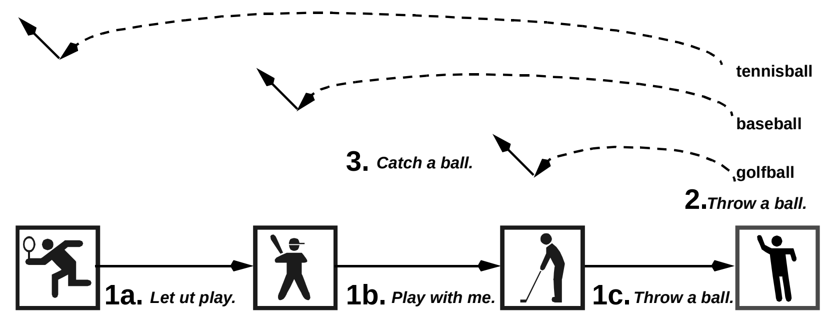
- Java demands that *checked* exceptions be declared in a `throws`-clause of the method if these exceptions are not handled by a `try-catch`-block in the method.

```
1.  // DivisionByZero5.java
2.  public class DivisionByZero5 {
3.      public static void division() throws DivisionByZero {
4.          int num1 = 10;
5.          int num2 = 0;
6.          if (num2 == 0)
7.              throw new DivisionByZero();
8.          System.out.println(num1 + " / " + num2 + " = " + (num1 / num2));
9.      }
10.     public static void main(String args[]) throws DivisionByZero {
11.         division(); // can throw a checked exception which must be declared
12.                     // in the throws-clause of the main() method!
13.     }
14. }
15. class DivisionByZero extends Exception {  // Checked Exception definition
16.     DivisionByZero() { super("Integer division by 0"); }
17. }
```

Output on the terminal:

```
Exception in thread "main" DivisionByZero: Integer division by 0
        at DivisionByZero5.division(DivisionByZero5.java:7)
        at DivisionByZero5.main(DivisionByZero5.java:11)
```


Example II: Exception Propagation



A player can only catch the ball which it can play with. After an exception is thrown, program control goes to the first method activation which can handle the exception.

→ ***program control***

Example II: Exception Propagation (cont.)

```
// (Checked) Exception definitions
class Tennisball extends Exception {};
class Baseball extends Exception {};
class Golfball extends Exception {};

// Tennis player
public class TennisPlayer {

    void startplay() {
        try {
            new BaseballPlayer().play();
            System.out.println(
                "Nobody wants to " +
                "play tennis!");
        }
        catch (Tennisball tb) {
            System.out.println(
                "Tennis is fun!");
        }
    }

    public static void main(String args[]) {
        new TennisPlayer().startplay();
    }
}
```

```
// Baseball player
class BaseballPlayer {

    void play() throws Tennisball {
        try {
            new GolfPlayer().play();
            System.out.println(
                "Nobody wants to " +
                "play baseball!");
        }
        catch (Baseball bb) {
            System.out.println(
                "Baseball is fun!");
        }
    }
}
```


Example II: Exception Propagation (cont.)

```
// Golf player

class GolfPlayer {

    void play()
            throws Tennisball, Baseball {
        try {
           new BallThrower().throwBall()
            System.out.println(
                "Nobody wants to" +
                " play golf!");
        }
        catch (Golfball gb) {
            System.out.println(
                "Golf is fun!");
        }
    }
}
```

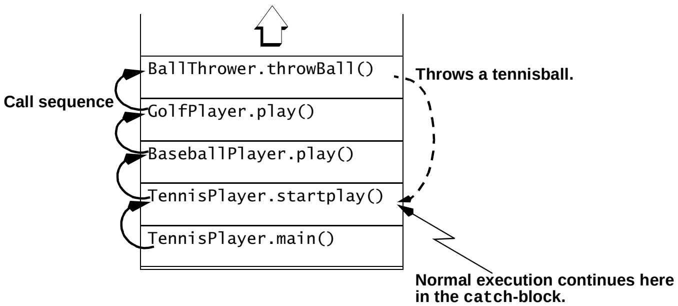
```
// Ball thrower

class BallThrower {

    void throwBall()
        throws Tennisball, Baseball,
               Golfball {

        int ballChoice = 2;
        switch (ballChoice) {
            case 1: throw new Tennisball();
            case 2: throw new Baseball();
            case 3: throw new Golfball();
        }
    }
}

// Note that execution of the method
// throwBall() is aborted when an
// exception is thrown.
```


Example II: Exception Propagation (cont.)

Output when ballChoice = 1 (i.e. tennisball):

```
Tennis is fun!
```

Method Activation Stack



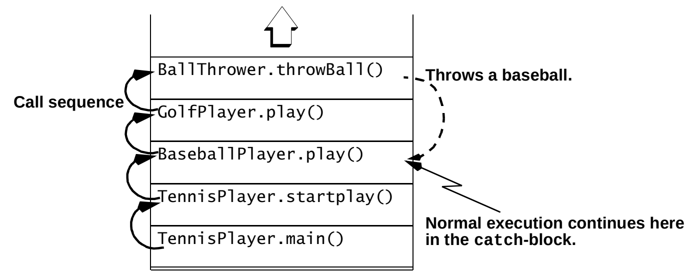
Note that the activation of both `GolfPlayer.play()` and `BaseballPlayer.play()` are "unwound".

Example II: Exception Propagation (cont.)

Output when ballChoice = 2 (i.e. a baseball):

```
Baseball is fun!
Nobody wants to play tennis!
```

Method Activation Stack



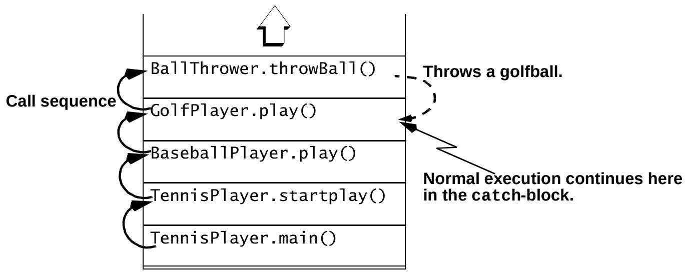
Note that the activation of `GolfPlayer.play()` is "unwound".

Example II: Exception Propagation (cont.)

Output when ballChoice = 3 (i.e. a golfball):

```
Golf is fun!
Nobody wants to play baseball!
Nobody wants to play tennis!
```

Method Activation Stack

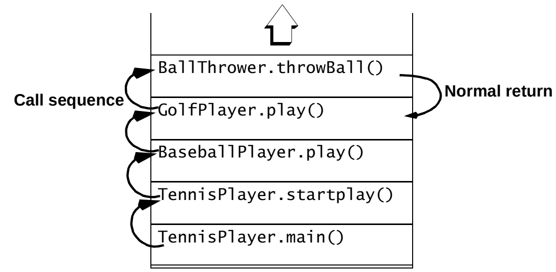


Example II: Exception Propagation (cont.)

Output when ballChoice > 3 or < 1:

```
Nobody wants to play golf!
Nobody wants to play baseball!
Nobody wants to play tennis!
```

Method Activation Stack



`finally` block

- The `finally` block is *always* executed whether or not an exception was thrown in the corresponding `try` block.
- The `finally` block is usually used for "cleanup" code.

Scenarios through the `try`-`catch`-`finally` block

- No exception is thrown in the `try` block.
  - All `catch` blocks are skipped, then the `finally` block is executed, and normal execution continues.
- Exception is thrown in the `try` block, and caught in a `catch` block.
  - Only the appropriate `catch` block is executed, then the `finally` block is executed, and normal execution continues.
- Exception is thrown in the `try` block, but not caught in a `catch` block.
  - All `catch` blocks are skipped, then the `finally` block is executed and *the exception propagated*.

If an exception is thrown in the `try` block, *rest of the* `try` *block is not executed*.

The `try-catch-finally` construct can be nested.

Example using `finally` block

```
// (Checked) Exception definitions
class Tennisball extends Exception {};
class Baseball extends Exception {};
class Golfball extends Exception {};

// Tennis player
public class TennisPlayer {
    void startplay() {
        try {
            new BaseballPlayer().play();
            System.out.println("Nobody wants to " + "play tennis!");
        } catch (Tennisball tb) {
            System.out.println("Tennis is fun!");
        } finally {
            System.out.println("I only play tennis!");
        }
    }

    public static void main(String args[]) {
        new TennisPlayer().startplay();
    }
}
```

```
// Baseball player
class BaseballPlayer {
    void play() throws Tennisball {
        try {
            new GolfPlayer().play();
            System.out.println("Nobody wants to " + "play baseball!");
        } catch (Baseball bb) {
            System.out.println("Baseball is fun!");
        } finally {
            System.out.println("I only play baseball!");
        }
    }
}

// Golf player
class GolfPlayer {
    void play() throws Tennisball, Baseball {
        try {
            new BallThrower().throwBall();
            System.out.println("Nobody wants to" + " play golf!");
        } catch (Golfball gb) {
            System.out.println("Golf is fun!");
        } finally {
            System.out.println("I only play golf!");
        }
    }
}
```



```
// Ball thrower
class BallThrower {
    void throwBall() throws Tennisball, Baseball, Golfball {
        int ballChoice = 2;
        switch (ballChoice) {
            case 1: throw new Tennisball();
            case 2: throw new Baseball();
            case 3: throw new Golfball();
        }
    }
}
```

Output from the program:

```
I only play golf!
Baseball is fun!
I only play baseball!
Nobody wants to play tennis!
I only play tennis!
```

Compare this output with the earlier example without the `finally` *block.*

Exceptions and Inheritance

- An *overridden method* in a subclass can only specify in its `throws` clause *all or a subset of exceptions* specified in the `throws` clause of the method definition in the superclass.

```
// Class A
    public void aMethodWithManyExceptions()
            throws Exception1, Exception2, Exception3 {...}
...
// In subclass B which extends class A
    public void aMethodWithManyExceptions()
            throws Exception1, Exception3 {...}
```

- Since all exceptions are objects, a `catch` block can specify an *exception superclass* as parameter.
  - A `catch` block can *shadow* other `catch` blocks which specify a *subclass*.
  - Compiler flags such situations as errors.

```
...
try { ... }
catch (Exception e) {...}                  // superclass shadows
catch (ArithmeticException a) {...}        // subclass.
...
```


Summary of Exception Handling

- "*Throw or Declare*"-rule for *checked* exceptions:
  - The checked exceptions which a method can throw but does not catch must be declared in a `throws` clause of the method.
- A method can catch exceptions which it throws and handle them in `catch` blocks.
- A method can explicitly throw an exception using the `throw`-statement.

```
... aMethod(...) throws Exception_1, Exception_2, ... , Exception_n {
  try {
      // statements which can lead to one of the following exceptions
      // being thrown: Exception_1, Exception_2, ... , Exception_n and
      //               Exception_a, ..., Exception_m
      throw new Exception_i();   // throw an exception
  } catch (Exception_a a) {      // exception handler for Exception_a
      // code to handle exception a
  }
  ...
    catch (Exception_m m) {      // exception handler for Exception_m
      // code to handle exception m
  } finally {
      // code to cleanup
  }                                          try-catch-finally block
}
```