

Exception Handling



- Run-time errors
- The exception concept
- Throwing exceptions
- Handling exceptions
- Declaring exceptions
- Creating your own exception

Run-time Errors



Sometimes when the computer tries to execute a statement something goes wrong:

- Trying to read a file that doesn't exist.
- Trying to divide an integer by zero.
- Calling a method with improper arguments. Say, calling the setTime() method of a Clock object with arguments that does not refer to a valid hour.

In these cases the instruction would fail. We say that a run-time error had occurred.

Methods Failure



Calling a method can fail (wrong input parameters, calculation error, system call failure etc.) - What should we do?

Use a Boolean return value and write:

```
if (myClock.setTime(11,10,66) == true) {  
    if (myClock.hourElapsed() == true) {  
        if ... {  
        }  
    }  
}
```

Problems in This Approach?



- *Not possible to return a value other than Boolean*
- *Difficult to understand the code*
- *No indication what exactly went wrong*

Exceptions



In Java, run-time errors are indicated by exceptions.

If a method wants to signal that something went wrong during its execution it throws an exception.

Throwing an Exception



Throwing an exception involves:

- creating an **exception object** that encloses information about the problem that occurred
- use of the statement `throw` to notify about the exception



Throwing an Exception Example



```
public void setTime(  
    int hour, int minute, int second) {  
    if (hour<0 || hour>23 ||  
        minute<0 || minute>59 ||  
        second<0 || second>59) {  
        throw new IllegalArgumentException(  
            "Invalid time");  
    }  
    this.hour = hour;  
    this.minute = minute;  
    this.second = second;  
}
```

© Ariel Shamir

7

The Exception Object



The information about the problem that occurred is enclosed in a real object, the exception object.

This information includes:

- The type of the problem
- The place in the code where the exception occurred
- The state of the run-time stack
- ... other information

© Ariel Shamir

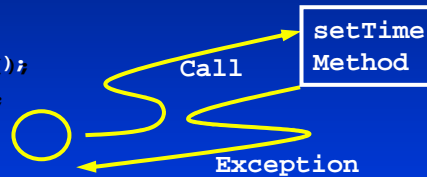
8

Who Receives the Exception?



The code that invoked the method will receive the exception object.

```
Clock myClock = new Clock();  
myClock.setTime(11,10,66);
```



It can then examine the information it carries and act accordingly (later).

The Type of the Exception



The most important information is the type of the exception

This is indicated by the class (i.e. type) of the exception object

The Java API defines classes for many types of exceptions

You can define more of your own

Illegal Argument Exception



The setTime() method used the exception IllegalArgumentException to signal that the values of the arguments were not valid.

IllegalArgumentException is a class defined in package java.lang. A method throws this type of exception if it wants to signal that an argument it got is not legal.

Examples of Exception Types



Several types of exceptions are defined in java.lang:

- **IllegalArgumentException** - passing an illegal argument to a method. Example: `Math.pow(0,0)`;
- **ArithmeticException** - division by zero.
- **NullPointerException** - trying to refer to an object through a reference variable whose value is null.

Occurrence of an Exception



When a program performs an illegal operation the following happens:

- An exception object is created and thrown
- The regular flow of the program stops
- The program may try to handle the exceptional situation (in a way we will explain shortly)
- If the program ignores the exception the program execution ceases. We sometimes say that the program crashes.

Occurrence of an Exception Example



```
class ClockProblem {  
    public static void main(String[] args) {  
        int hours, minuets, seconds;  
        // ...  
        Clock myClock = new Clock();  
        myClock.setTime(hours, minuets, seconds);  
        // ...  
    }  
}
```

What happens if hours < 0 or seconds > 59?

The Root of the Exception



```
public void setTime(  
    int hour, int minute, int second) {  
    if (hour<0 || hour>23 ||  
        minute<0 || minute>59 ||  
        second<0 || second>59) {  
        throw new IllegalArgumentException(  
            "Invalid time");  
    }  
    this.hour = hour;  
    this.minute = minute;  
    this.second = second;  
}
```

Exception Outcome



```
class ClockProblem {  
    public static void main(String[] args) {  
        int hours = -1;  
        int minuets,seconds;  
        // ...  
        Clock myClock = new Clock();  
        myClock.setTime(hours,minuets,seconds);  
        // ...  
    }  
}
```

hour = -1

A Clock must have a positive hour

Hey, no one cares to listen!

I'll crash the method!

Avoiding Exceptions



In the former example we could have checked if the parameters are legal.

Sometimes we cannot prevent an exceptional state. For example when reading from a diskette we cannot tell if the diskette is readable or not without trying to read from it.

Separating Exception Cases in Your Code



Even if we can check in advance if there is a possibility of running into an exceptional case we may want to handle this case outside the main block, so as not to complicate the readability of the handling of the regular case.

Handling Exceptions



In order to handle exceptional cases, you should put the code section that might throw an exception inside the body of a `try..catch` statement.

The parenthesis that follow the catch keyword, specify the type of exception we want to handle.

Try.. Catch Block



The idea is to "try" and execute some statements.

- If whatever you tried succeeds, the catch statements are ignored.
- If an exception is thrown by any of the statements within the try block, the rest of the statements are skipped and the corresponding catch block is executed.

Try..Catch Syntax



```
try {
    // code statements
} catch (ExceptionType e) {
    // handler statements
}
```

Each catch clause has an associated exception type. When an exception occurs, processing continues at the first catch clause that matches the exception type.

Handling Exceptions Example



```
class ClockProblem {
    public static void main(String[] args) {
        int hour, minuets, seconds;
        // ...
        Clock myClock = new Clock();
        try {
            myClock.setTime(hours, minuets, seconds);
        } catch (IllegalArgumentException iae) {
            // act accordingly...
        }
        // ...
    }
}
```

Call → setTime Method

Exception

Handling Other Exceptions



```
class ClockProblem {
    public static void main(String[] args) {
        // ...
        Clock myClock = new Clock();
        try {
            myClock.setTime(hours, minuets, seconds);
        } catch (ArithmeticException ae) {
            // ...
        } catch (IllegalArgumentException ia) {
            output.println("Illegal input!");
        }
        // ...
    }
}
```

Declaring for Exceptions



A method can declare exceptions it may throw. We will see later that sometimes this is a must

The declaration for exceptions a method can throw is done using the `throws` keyword

The user of the method is warned against possible exceptions this method can throw

Declaring Exceptions Using 'throws'



```
public void setTime(  
    int hour, int minute, int second)  
    throws IllegalArgumentException {  
    if (hour<0 || hour>23 ||  
        minute<0 || minute>59 ||  
        second<0 || second>59) {  
        throw new IllegalArgumentException(  
            "Invalid time");  
    }  
    this.hour = hour;  
    // ...  
}
```

© Ariel Shamir

25

Documenting Exceptions



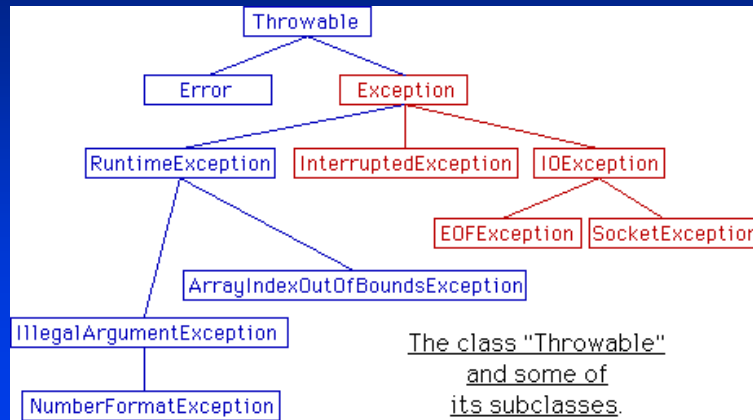
The exceptions that might be thrown by a method should also be documented with the @exception tag or @throws :

```
/**  
 * Sets the time of the clock.  
 * @param hour, minute, second The new time.  
 * @throws IllegalArgumentException If the  
 * parameters don't represent a valid time.  
 */
```

© Ariel Shamir

26

Exception Hierarchy



The class "Throwable" and some of its subclasses.

© Ariel Shamir

27

RuntimeException



Every exception is a runtime error.

Exceptions of type `RuntimeException` do not have to be declared.

A routine can throw an `IllegalArgumentException` without announcing the possibility.

A program that calls that routine is free either to catch or to ignore such exception.

© Ariel Shamir

28

Mandatory Declaration



For those exception classes that require mandatory handling, the situation is different. If a subroutine can throw such an exception, that fact must be announced in a `throws` clause in the routine definition. Failing to do so is a syntax error that will be reported by the compiler.

Mandatory Handling



The calling method must either catch the mandatory exception or declare that it can throw it using the `throws` clause.

This is because if this exception is not caught, it will be delegated to the next level of method call.

Creating Your Own Exception



```
public class EmpNotFoundException
    extends Exception {
    public EmpNotFoundException(String name)
    {
        super("Employee " + name +
            " is not found in company");
    }
}
```

Throwing Your Exception



```
public class CompanyE {
    private String[] names;
    // more...
    public void pay(String employeeName)
        throws EmpNotFoundException {
        int index = lookFor(employeeName);
        if (index == -1) throw new
            EmpNotFoundException(employeeName);
        Employee e = getEmployee(index);
        e.pay();
    }
}
```

Finally Block



```
try {  
    // statements (may include return)...  
    // Executed normally  
} catch (Exception e) {  
    // statements...  
    // Executed when exceptions are thrown  
} finally {  
    // statements...  
    // Executed after all catch cases  
    // Also after the try statements  
    // Even before (if it includes a return)  
    // we exit from the method!  
}
```

Advantages of Exceptions



- *Reduces programming complexity ("clean code")*
- *Prevents ignoring errors*
- *Refers errors to where they can be dealt with*
- *Frees the method return value mechanism*
- *Provides more informative than just a return value*