Exceptions, assertions & testing

Lecture 13 of TDA 540 (Objektorienterad Programmering)

Carlo A. Furia Alex Gerdes

Chalmers University of Technology – Gothenburg University

Fall 2017

Exceptional behavior

Sometimes things do not go as planned during the execution of a program:

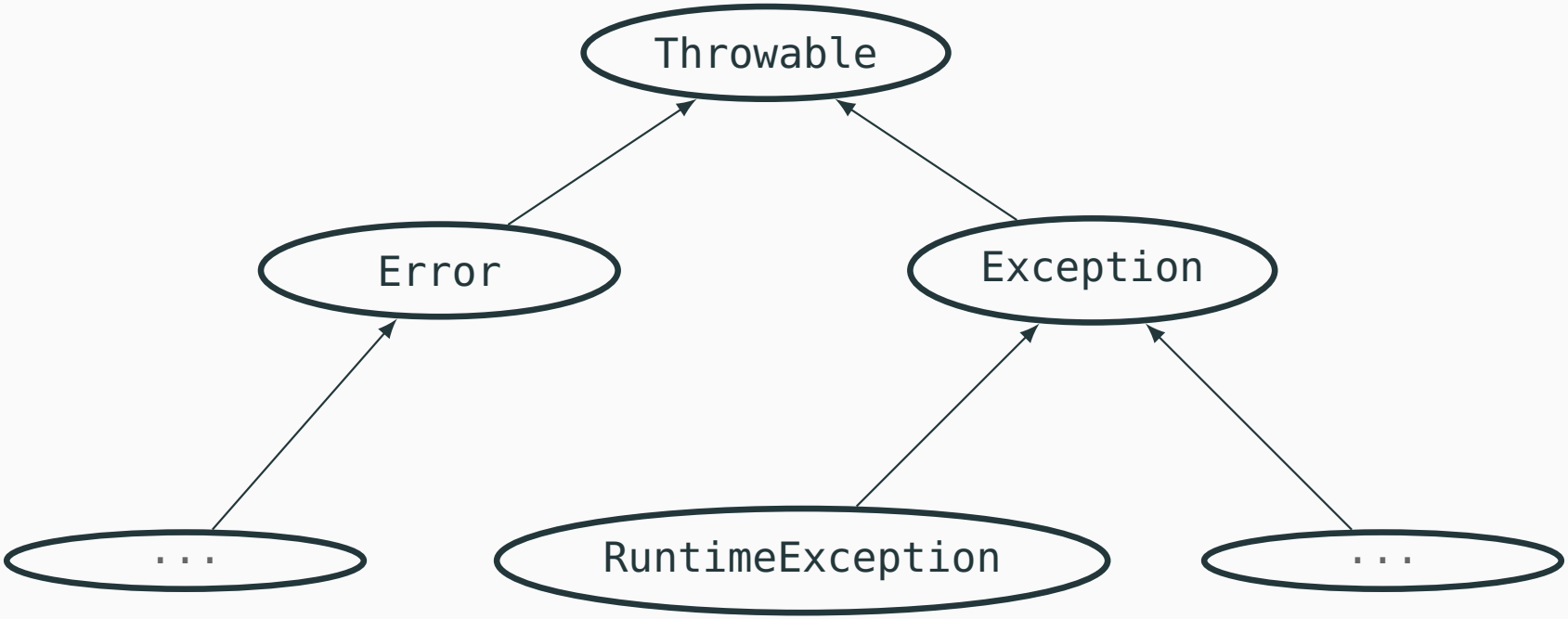
- the user provides invalid input
- the program runs out of memory
- a network connection cannot be established because a website is down
- ...

To make the program more robust about such exceptional events, we would like to define two separate behaviors:

1. normal behavior: what we have seen so far
2. exceptional behavior: using exceptions and exception handling

Exception objects

Exception handling in Java uses exception objects, which are instances of exception classes



Programming with exceptions

There are two sides to programming with exceptions:

suppliers throw (raise) exception objects to signal to clients that an exceptional event has occurred

clients catch (handle) exception objects and take counter-measures to work around the exceptional event

Programs with exception-handling have two control flows:

1. normal control flow: no exception occurs, exception-handling code is not executed
2. exceptional control flow: exceptions occur, exception-handling code is executed

Throwing exceptions

How to signal exceptional behavior:

- create an object `ex` of class that inherits from `Throwable`
  - one of the standard exception types in `java.lang`
  - or one new exception class (must be a heir of `Throwable`)
- raise the exception and pass control to the caller with **`throw`** `ex`

```
// parse nonnegative integer string
int stringToInt(String str) {
  int result;
  if (str == null) throw new NullPointerException();
  for (int i = 0; i < str.length(); i++) {
    if (!Character.isDigit(str.charAt(i)))
      throw new NumberFormatException(str + " is not an integer!");
  } // ... normal behavior ...
  return result;
}
```

Catching exceptions

Exception-handling code uses the `try`/`catch`/`finally` statements.

BLOCK	BEHAVIOR
`try { ... }`	execute the code in the block monitoring for raised exceptions
`catch(ET e) { ... }`	if an exception of type `ET` is raised while executing the corresponding `try` block, execute the code in the `catch` block, where `e` points to the raised exception object
`finally { ... }`	after executing the corresponding `try` block, and possibly after executing any `catch` block, execute the code in the `finally` block

A `try` block determines a "client" that may receive exception objects, and the two corresponding control flows (normal and exceptional).

A `finally` block is normally used to close/deallocate resources (such as files) regardless of whether an exception is thrown or not.

Exception handlers

Every `try` block is followed by zero or more `catch` blocks, zero or one `finally` block, or both. At least one `catch` block or one `finally` block is required (otherwise the `try` would be useless).

```
// parse nonnegative integer string
Integer stringToInteger(String str) {
  Integer result = null;
  try {
    result = Integer.parseInt(str);
  } catch (NumberFormatException e) {
      // if parseInt throws a NumberFormatException
      // print this error message:
      System.out.println(str + " is not a valid nonnegative int");
  }
  // returning -1 means that parsing a nonnegative integer failed
  if (result != null && result >= 0) return result; else return -1;
}
```

Exception handlers: catch blocks

Catch blocks use types to handle specific kinds of exceptions:

```
catch (ET e) { /* handler code */ }
```

- handle exceptions whose type is a subtype of `ET`
- `ET` must be a subtype of `Throwable`
- `e` behaves like a local variable inside the handler block
- if multiple catch blocks are defined, execute the first block whose type matches the raised exception's type

Multi-catch blocks:

```
catch (ET1 | ET2 | ET3 e) { /* handler code */ }
```

- handle exceptions whose type is a subtype of `ET1`, of `ET2`, or of `ET3`
- `ET1`, `ET2`, and `ET3` must not be related by inheritance
- `e` behaves like a constant inside the handler block

Nested exception handling blocks

When an exception of type `E` is thrown while executing the code inside a `try` block:

1. the first (in textual order) `catch` block whose type is a supertype of `E` executes
2. then, the `finally` block executes (if it exists)
3. then, execution continues after the `try` block

If no suitable `catch` block exists, or if a catch block raises an exception:

1. the `finally` block executes (if it exists)
2. then, the exception propagates to the next enclosing handler

If no enclosing handler exists:

1. the exception propagates to the `main` method
2. the program forcefully terminates

Catch, handle, and rethrow: example

Read an `n`-digit integer from a file with name `fn`:

```
int readNum(String fn, int n)
```

Exceptions to handle many things that can go wrong:

- a file with name `fn` doesn't exist
- the file exists but it cannot be opened
- the file's content is not a valid integer
- the file's content is an integer with fewer than `n` digits

Read n-digit integer from file

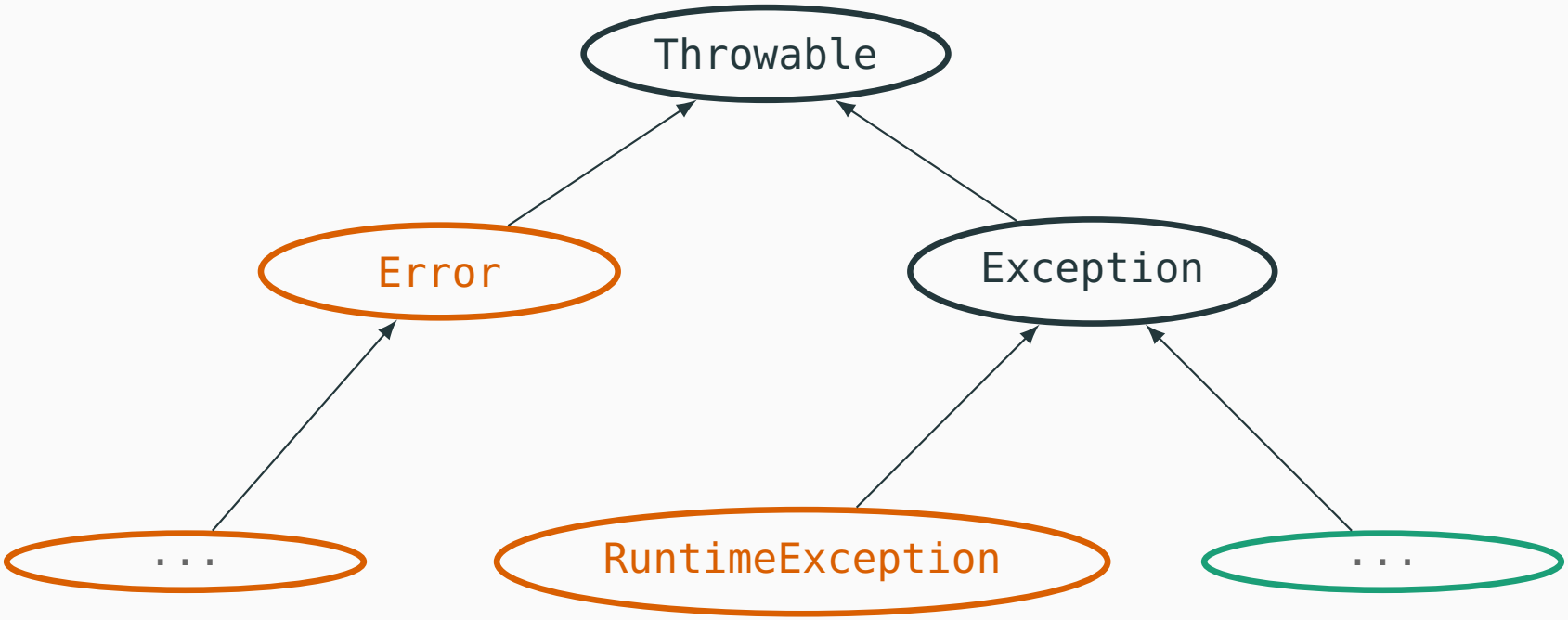
```
int readNum(String fn, int n)
  throws FileNotFoundException, IOException { // may be propagated
{ int result;
  BufferedReader br = null;
  try {
    br = new BufferedReader(new FileReader(fn));
    String str = br.readLine();
    if (str.length() < n)
        // throw too few digits exception
      throw new TooFewDigitsException(str.length());
    result = Integer.parseInt(str);
  } catch (FileNotFoundException e) { throw e; } // propagate except.
    catch (IOException e) { throw e; } // propagate exception
    catch (NumberFormatException e) { result = 0; } // handle except.
  finally { if (br != null) br.close(); } // close file
                          // even if an exception is propagated!
  return result; }
```

Client code of `readNum`

```
int x;
final String filename = "iamalldigits.txt";
try {
  // try to read 7-digit number
  x = readNum(filename, 7);
} catch (TooFewDigitsException e) {
  // try again, with number of characters read
  try { x = readNum(filename, e.numRead); }
  catch (Exception e)
  { System.out.println("No valid integer could be read"); }
} catch (Exception e) {
  System.out.println("Some IO error occurred"); }
}
```

Checked vs. unchecked exceptions

Java exception classes are partitioned in checked and unchecked



Checked vs. unchecked exceptions

Java exception classes are partitioned in checked and unchecked

CHECKED	UNCHECKED
declared in method signatures with `throws`	not declared explicitly (but normally still in documentation)
clients of methods using checked exceptions have to handle the exceptions, or declare that they may propagate them	clients may or may not handle the exceptions
the compiler checks that exceptions are handled	if unhandled exceptions occurs, the program terminates

Declaring a new exception class

The unchecked exception class `TooFewDigitsException` used in the previous example is declared using inheritance:

```
public class TooFewDigitsException extends Error {
   int nDigits;

   TooFewDigitsException(int nDigits) {
      this.nDigits = nDigits;
   }
}
```

Checked exceptions

Checked exceptions must be handled or declared with **`throws`**. For example, constructor of class `java.io.FileReader` declares checked exception `FileNotFoundException`.

Declare checked exception:

```
void printFile(String filename)
  throws FileNotFoundException
{
  FileReader fr;
  // may throw exception,
  // which is propagated
  fr = new FileReader(filename);
  // ...
}
```

Handle checked exception:

```
void printFile(String filename)
{
  FileReader fr;
  try {
    fr = new FileReader(filename);
  } // handle exception if thrown
  catch (FileNotFoundException e) {
    System.out.println("Fail!");
  }
}
```

Exceptions: checked or unchecked?

Java tends to prefer checked exceptions:

- unchecked exceptions are behavior that is not explicit (in the method signature)
- clients can be prepared to deal with checked exceptions

However, checked exceptions have their own disadvantages:

- proliferation of exception-handling code
- complex logic to decide which exceptions to propagate and which to handle
- changing exceptions may change the public interface of methods

How to choose in practice between checked and unchecked exceptions?

- use a checked exception if the client can do something to recover from the exception
- document the usage of unchecked exceptions too
- usually prefer checked exceptions to error codes

Writing correct programs

Programming means writing instructions that achieve a certain functionality. How do we know if a program is correct? And what does it even mean that a program is correct?

To this end, we distinguish between implementation and specification:

- The implementation consists of the actual code that is written, compiled, and executed
- The specification is a description of what the program should do, usually more abstract than the implementation

Implementation:

```
int sum(int[] a)
{ int sum = 0;
  for (int v : a)
    sum += v;
  return sum; }
```

Specification:

method `sum` takes a non-null reference `a` to an array of integers, and returns the sum of all values in `a`

Method specifications

Let us focus on input/output specifications of individual methods.
Such specifications consist of two parts:

1. precondition: a constraint that defines the method's valid inputs
2. postcondition: a functional description of the output after executing the method

Implementation:

```
int sum(int[] a)
{ int sum = 0;
  for (int v : a)
    sum += v;
  return sum; }
```

Specification:

1. precondition: `a != null`
2. postcondition:
   `sum` $== \sum_{0 \leq k < \texttt{a.length}} \texttt{a}[k]$

Method specifications in object-oriented programs

In object-oriented programs, the input and output of a method also include the object state before and after executing the method.

Implementation:

```
class BankAccount {
  int balance;

  void deposit(int amount)
  { balance += amount; }
}
```

Specification:

1. precondition: `amount >= 0`, no constraint on `balance`
2. postcondition:
   "after" `balance` == "before" `balance + amount`

Pre/postconditions in Java

Java does not have support for writing pre/postcondition specifications in the source file.

JML is a system for annotating Java programs in special comments.

```
class BankAccount {
  int balance;

  //@ requires amount >= 0;
  //@ ensures balance == \old(balance) + amount;
  void deposit(int amount)
  { balance += amount; }
}
```

Assertions

Even if Java does not have support for writing pre/postcondition specifications in the source file, it supports assertions, which are a more primitive way of expressing specifications in the source file of a program.

When execution reaches the statement:

```
assert condition;
```

the Boolean `condition` is evaluated on the current program state:

1. if `condition == true`, execution continues
   (the assertion passes: no effects)
2. if `condition == false`, an exception `AssertionError` is thrown
   (the assertion fails)

Important: assertion checking is disabled by default (`assert` statements are skipped during execution). To enable it run your program with `java -ea MyProgram`.

Assertions and invariants

Assertions encode invariants: conditions on the program state that a correct programs should satisfy whenever execution reaches the location of the assertion.

- assertions encode the assumptions the program relies on
- in a correct program, assertions always evaluate to true (and thus no effect)
- an assertion evaluating to false indicates that there is a mismatch between assumptions and actual execution (probably an error)

```
void deposit(int amount) {
  assert amount >= 0;
  int old_balance = balance;
  balance += amount;
  assert balance >= old_balance;
}
```

```
account = new BankAccount();
assert account.balance == 0;
int square = x * x;
assert square >= 0;
account.deposit(square);
assert account.balance >= 0;
```

Assertions and exceptions

Both assertions and exceptions are means to deal with unwanted behavior. In Java, failing assertions throw exceptions, so there is a clear connection between the two.

- exceptions should signal exceptional but possible behavior
  - the exceptional behavior requires a special handling
  - but exceptions may occur even in a perfectly correct program: for example, invalid user input, or I/O errors
- assertions should encode the specification of correct behavior
  - when the specification is satisfied, nothing special happens
  - assertions should never fail in a correct program

In practice, however, since Java does not check assertions by default and uses exceptions extensively, exceptions are used also in cases where an assertion would be more appropriate.

Verification

Verification is the process of checking that a program is correct. This means that, in addition to the implementation, there is also some form of specification (possibly only informal).

Two main techniques to do verification:

- testing: run the program using many different inputs, check that every run satisfies the specification
- formal verification: mathematically prove that every possible execution of the program satisfies the specification

Unit testing

Testing in a nutshell:

- run the program using many different inputs
- check that every run satisfies the specification

Unit testing: testing one method in isolation.

Method `deposit` under test:

```
class BankAccount {
  int balance;

  void deposit(int amount)
  { balance += amount; }
}
```

Testing code:

```
BankAccount ba = new BankAccount();
ba.deposit(0);
assert ba.balance == 0;
ba.deposit(121);
assert ba.balance == 121;
ba.deposit(3);
assert ba.balance == 121 + 3;
```

How many inputs can we test?

If we could check all valid inputs of a method, testing would be equivalent to proving correctness. But is this feasible in practice?

- Test all possible input arguments to `deposit`: $\simeq 2^{31} \simeq 2.1 \cdot 10^9$

```
// runs in 0.05 seconds
for (int v = 0; v < Integer.MAX_VALUE; v++) {
  BankAccount ba = new BankAccount();
  ba.deposit(v);
  assert ba.balance == v;
} // must test Integer.MAX_VALUE separately: why?
```

- Test all possible input states to `deposit`: $\simeq 2^{32} \cdot 2^{31} \simeq 9.2 \cdot 10^{18}$

```
// runs in > 500 days
for (int u = Integer.MIN_VALUE; u < Integer.MAX_VALUE; u++)
 for (int v = 0; v < Integer.MAX_VALUE - ((u > 0) ? u : 0); v++)
 { BankAccount ba = new BankAccount();
   ba.balance = u;
   ba.deposit(v);
   assert ba.balance == u + v; }
```

Testing in practice

Testing cannot realistically try out all possible inputs. Instead, its main purpose is to try out a good number of varied inputs in a way that has a good chance of exposing errors.

Testing heuristics for selecting inputs:

- boundary values
  - for `int`: `Integer.MAX_VALUE`, `Integer.MIN_VALUE`
  - for `String`: `null`, `""`
- partition the input values, and pick one element per partition
  - for `int`: `n < 0`, `n == 0`, `n > 0`
  - for `String`: string of digits, string of alphabetic characters, string of non-alphanumeric characters, . . .
  - partition according to the conditions in the method under test
- pick some inputs at random
- regression testing: pick inputs that triggered errors previously (and now should have been fixed)

Thou shall test your code!

Systematically testing your code is a good practice that every programmer should follow.

- Test extensively: try to write unit tests for all `public` methods of all your classes
- Test early: as soon as a class has a public interface, you can write tests for it, which reflect the specification (the tests will fail until you have implementations)
- Test often: as soon as you change anything in a class implementation, rerun the tests for it to check that everything works
- Test for regressions: for every error that you discover, add a test that exposes the error to your collection of tests

Unit testing example

```
class Account {
  int balance;
  List<String> owners;

  // add amount to balance
  void deposit(int amount) {
    balance += amount;
  }

  // add account owner if not null
  void addOwner(String name) {
    if (name != null)
      owners.add(name);
  }
}
```

```
class AccountTest {
  static void testDeposit() {
    Account ba = new Account();
    ba.deposit(10);
    assert ba.balance == 10;
    ba.deposit(-10);
    assert ba.balance == 0;
  }
  static void testAddOwner() {
    Account ba = new Account();
    ba.addOwner("Jane Doe");
    assert ba.owners.size() == 1;
    ba.addOwner(null);
    assert ba.owners.size() == 1;
  }
}
```