

Introduction to Programming - Concepts and Tools

Lecture 11

Exceptions and Tests

Overview

- Errors and robust programs
- Exceptions
 - `try-catch{catch}`, `throws`, `throw`
 - Your own exceptions
- Testing
 - Psychological effects
 - Incompleteness
 - White-box (structural)
 - Black-box (functional)

Error Types

There are two types of errors

- **Static errors** (syntactical and type errors) are taken care of by the **compiler**.
- **Runtime errors** (semantical errors) are to be avoided by the programmer, and may be
 - Computing the wrong result
 - Unintended termination due to **exceptions**
 - Index out of bound, reading after EOF, or
 - Null pointer or arithmetical exceptions

Non-robust programs

Programs should be **robust**, this one isn't (may e.g. divide by zero)

```
class NotRobust {  
  
    public static int Divide(int a, int b) {return a/b;}  
  
    public static void main (String[] args) {  
  
        System.out.println("Type two positive"  
                            + "integers a and b");  
        int a = Console.in.readInt();  
        int b = Console.in.readInt();  
        System.out.println("a/b is: " + NotRobust.Divide(a,b));  
    }  
}
```

NotRobust.java

Jens Chr. Godskesen, ITU

4

Making Programs More Robust

Ideally a program should not terminate unintendedly. E.g. a program should deal gracefully with unexpected inputs.

- Why should the preceding program terminate when b is zero?
- Why should it terminate in case the user provides non integral inputs?

We would like to treat some **exceptions** not as **errors** (causing program termination) but as unexpected (but controllable) behaviour!

Making Programs More Robust

One way to obtain robustness is to make use of a class

```
class Status {  
    public boolean succes;  
}
```

and to let methods take status objects as parameters. Upon method return the status (succes or failure) decides the further program execution.

A more robust program

```
class MoreRobust {  
  
    public static int Divide(int a, int b, Status s) {  
        if (b == 0) s.succes = false;  
        else {b = a/b; s.succes = true;}  
        return b;  
    }  
  
    public static void main (String[] args) {  
        System.out.println("Type two integers a and b");  
        int a = Console.in.readInt();  
        int b = Console.in.readInt();  
        Status s = new Status();  
        int r = MoreRobust.Divide(a,b,s);  
        if (s.succes) System.out.println("a/b is: " + r);  
        else System.out.println("b is: zero");  
    }  
}  
MoreRobust.java
```

Inefficient and complex

Although we use ordinary control constructs this approach is not very elegant, we need to:

- Add an extra parameter to `Divide`
- Declare the extra variable `s`, and
- Test `s.succes` in every execution

Moreover, the complexity is increased in case of nested method calls, e.g.

```

class ToComplex {

    public static int Divide(int a, int b, Status s) {
        if (b == 0) s.succes = false;
        else {b = a/b; s.succes = true;}
        return b;
    }

    public static int ReadAndDivide(int a, Status s) {
        return ToComplex.Divide(a, Console.in.readInt(), s);
    }

    public static void main (String[] args) {
        System.out.println("Type integers a and b");
        int a = Console.in.readInt();
        Status s = new Status();
        a = ToComplex.ReadAndDivide(a, s);
        if (s.succes) System.out.println("a/b is: " + a);
        else System.out.println("b is: zero");
    }
}

```

Exception Handling

To avoid such complexity **Exception Handling** has been developed.

An **exception** is a condition in a program that is expected to occur only rarely if ever.

In the exceptional case, but only then, exception handling takes over control from the normal program flow.

Example

```
public static void main (String[] args) {  
  
    System.out.println("Type two integers a and b");  
    int a = Console.in.readInt();  
    boolean inputOk = false;  
    while (!inputOk) {  
        int b = Console.in.readInt();  
        try {  
            int r = a/b; // ArithmeticException may occur  
                        // then control leaves to catch  
            System.out.println("a/b is: " + r);  
            inputOk = true;  
        }  
        catch (ArithmeticException e) {  
            System.out.println("b is 0,input new b");  
        }  
    }  
}
```

```

public static void main(String[] args) throws IOException {
    DataInputStream input = null;
    if (args.length != 1) {
        System.out.println("Usage: java BinaryInput filename");
        System.exit(1);
    }
    try { input = new DataInputStream( new FileInputStream(args[0]) );}
    catch (IOException e) {
        System.out.println("Could not open " + args[0]);
        System.exit(1);
    }
    int count = 0;
    try {
        while (true) {
            int myData = input.readInt();
            count++;
            System.out.print(myData + " ");
            if (count % 4 == 0)
                System.out.println();
        }
    }
    catch (EOFException e) { }
    if (count % 4 != 0) System.out.println();
}

```

Try-Catch Blocks

The general format of a try-catch block is

```
try {  
    //code that may throw an exception  
}  
catch (ExceptionType identifier) {  
    // code to handle the exception  
}
```

If any statement in the try-block throws an exception of `ExceptionType` control is *immediately* transferred to the catch-block.

Throwing Exceptions

If exceptions are not caught and handled at the spot where they first occur they are **thrown** up one level, and so forth until they are caught. If they are never caught the program terminates.

This implies that methods supposed to return a value but where exceptions are not caught does not return a value.

```

public static int Divide(int a, int b) {return a/b;}
// may throw ArithmeticException and no value returned

public static void main (String[] args) {

    System.out.println("Type two positive"
                       + "integers a and b");
    int a = Console.in.readInt();
    boolean inputOk = false;
    while (!inputOk) {
        int b = Console.in.readInt();
        try {
            System.out.println("a/b is: " + Throwing.Divide(a,b));
            inputOk = true;
        }
        catch (ArithmeticException e) {
            System.out.println("b is zero, please input new b");
        }
    }
}

```

Dealing with several exceptions

Several types of exceptions may be caught by the same try-catch statement

```
try {  
    //code that may throw an exception  
}  
catch (ExceptionType1 identifier1) {  
    // code to handle the exception  
}  
catch (ExceptionType2 identifier2) {  
    // code to handle the exception  
}  
...
```

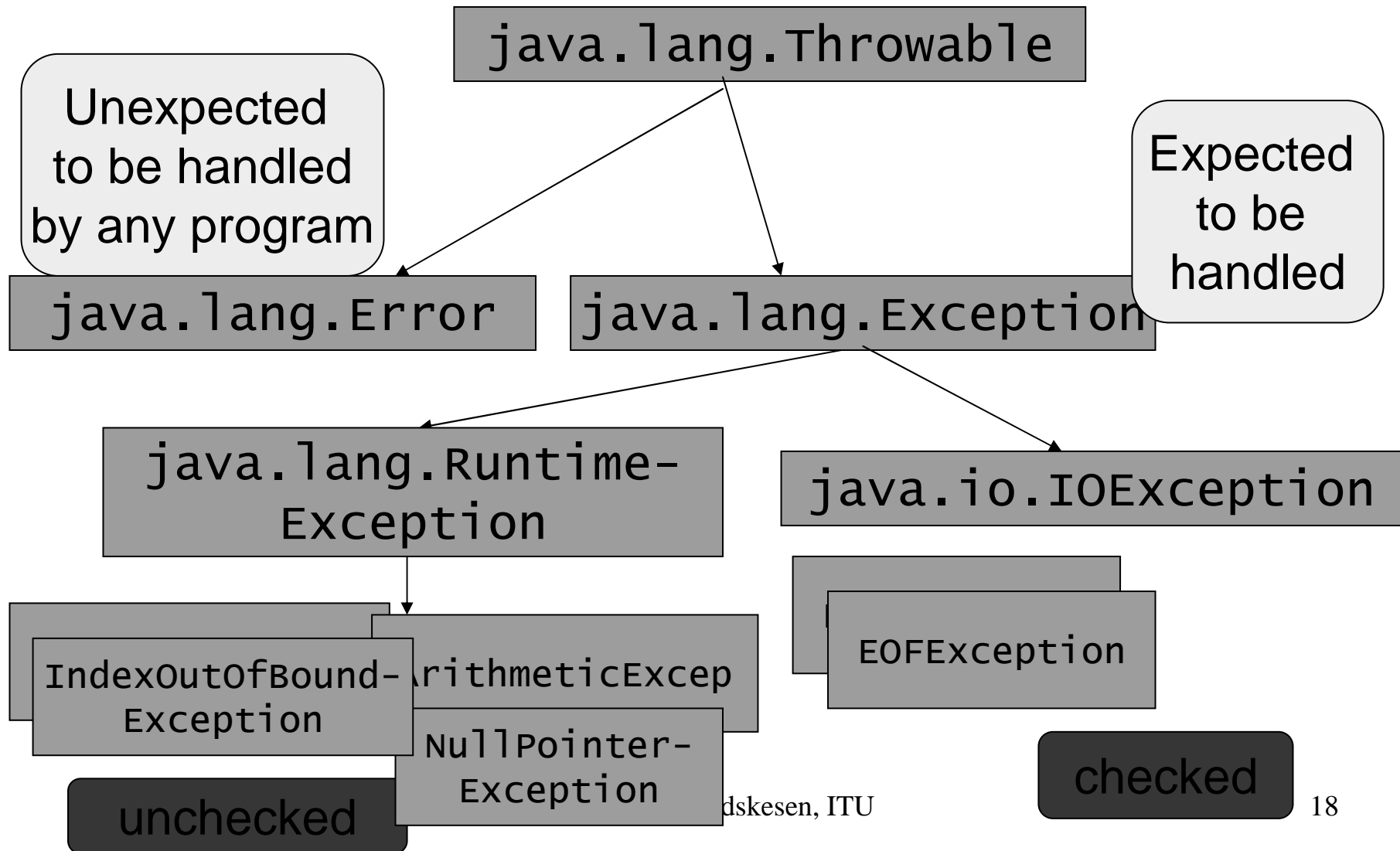
try-catch statements may be nested:

```

System.out.println("Type two positive integers a and b");
boolean inputOk = false;
try {
    int a = Console.in.readInt();
    while (!inputOk) {
        try{
            int b = Console.in.readInt();
            int r = a/b;
            System.out.println("a/b is: " + r);
            inputOk = true;
        }
        catch (ArithmeticException e) {
            System.out.println("b is zero, input new b");
        }
        catch (NumberFormatException e) {
            Console.in.readLine();
            System.out.println("b is not a number, input new");
        }
    }
}
catch (NumberFormatException e) {
    Console.in.readLine();
    System.out.println("a is not a number, try again..");
}

```

Exceptions



Checked vs. Unchecked

Exceptions that aren't subclasses of `RuntimeException` (**checked exceptions**) must either be handled in the method where it can occur or *declared* by it.

```
class HelloFile {
    public static void main(String[] args)
        throws java.io.IOException // a throws declaration
    {
        PrintWriter out =
            new PrintWriter(new FileWriter("hello.txt"));
        out.println("Hello, file system!");
        out.close();
    }
}
```

Unchecked exceptions (subclasses of `RuntimeException`) need not be **declared** (they are common, occur anywhere).

Several catch matches

The first catch that either specifies the exception thrown or a super class of it is chosen, e.g. the last catch below will never be actuated (however, a compile error would occur)

```
try{
    int b = Console.in.readInt();
    int r = a/b;
    System.out.println("a/b is: " + r);
    inputOk = true;
}
catch (RuntimeException e) {
    System.out.println("b is zero, input new b");
}
catch (NumberFormatException e) {
    Console.in.readLine();
    System.out.println("b is not a number, input new");
}
```

My Own Exception

Exceptions may be defined extending (a subclass of) class `Throwable`.

```
class MyException extends Exception {  
    public MyException(String m) {super(m);};  
}
```

```
public static int Divide(int a, int b) throws MyException {  
    try {  
        return a/b;  
    }  
    catch (ArithmeticException e) {  
        throw new MyException("Divisor is zero");  
    }  
}
```

```

try {
    int a = Console.in.readInt();
    while (!inputOk) {
        try{
            int b = Console.in.readInt();
            System.out.println("a/b is: " +
                MyOwnException.Divide(a,b));
            inputOk = true;
        }
        catch (MyException e) {
            System.out.print(e);System.out.println(", input new b");
        }
        catch (NumberFormatException e) {
            Console.in.readLine();
            System.out.println("b is not a number, input new b");
        }
    }
}
catch (NumberFormatException e) {
    Console.in.readLine();
    System.out.println("a is not a number, try again ...");
}

```

Pre- and Post-conditions

A **pre-condition** for a method is a *requirement* that must be satisfied by the caller, otherwise the method is not guaranteed to compute as expected.

A **post-condition** for a method is a *guarantee* that must be satisfied by the computation and result of the method.

If a pre- or post-condition is violated an error may be the result, but even better this unexpected behaviour may lead to exception handling taking over the program control flow.

Pre- and Post-conditions

```
public static int Divide(int a, int b) throws Exception {  
    //pre-condition, violation is an exception  
    if (!(b > 0)) throw new Exception("Divisor is zero");  
    else return a/b;  
}
```

```
public static int gcd ( ) throws Exception {  
  
    System.out.println("Type two natural numbers: ");  
  
    a = Console.in.readInt();  
    b = Console.in.readInt();  
  
    ...  
  
    //post-condition, violation is an exception  
    if (a < 0) throw new Exception("Invalid result ...");  
    System.out.println("The gcd is: " + a);  
}
```

What is (unit) testing? (I)

What does it mean to test:

```
public class Triangle {
    protected int a, b, c;

    public Triangle(int a, int b, int c) {
        this.a = a; this.b = b; this.c = c;}

    public boolean invalid() {
        return (a+b <= c) || (a+c <= b) || (b+c <= a);}

    public boolean equilateral() {
        if (invalid()) return false; return (a == b)&&(b == c);}
}
```

What is testing? (II)

Is testing the process of running the program to:

- demonstrate that errors are not present?
- show that the program performs its intended functions correctly?
- establish confidence that the program does what it is supposed to do, and *does not what it is not supposed to do?*

What is testing? (III)

In practice it's impossible to achieve the before mentioned goals because testing is **incomplete**, i.e. testing cannot guarantee the absence of errors

Instead, by testing (which is a costly process) we would like to increase the value of a program, i.e. find and remove errors (that we assume are there).

What is testing? (IV)

Testing is the process of executing a program with the intent of finding errors.

A **good** test case is one that has high probability of detecting an as-yet undiscovered error.

A **successful** test case is one that detects an as-yet undiscovered error (because it increases the value of the program)

The psychological effect (I)

Human beings are highly *goal oriented*.

- If the goal is to show **absence** of errors we tend to select test cases with **low** probability of finding errors.
- If the goal is to show **presence** of errors we tend to select test cases with **high** probability of finding errors.

The psychological effect (II)

Testing is a (sadistically) destructive process.

Most people are inclined to be creative and constructive, rather than destructive, so they find testing difficult.

The psychological effect (III)

- **A programmer should avoid attempting to test his or her own program**
 - Can programmers be adequately destructive towards something they constructed themselves?
 - Has the programmer understood the problem statement?

Similarity

- **A programming organization should not test its own programs**

The psychological effect (IV)

Test cases must be written for invalid and unexpected, as well as valid and expected input conditions

- Test cases representing unexpected and invalid input conditions have higher error-detection yield than do test cases for valid input conditions
- Examining a program to see if it does what it is supposed to do is not good enough, test also if it not does what it is *not supposed to do*. E.g. (5,5,5) is a valid triangle, whereas (1,2,3) is invalid.

Testing is incomplete (I)

**Testing cannot guarantee
the absence of errors**

Testing is incomplete (II)

If the source code is inaccessible (a black-box), then (black-box) test cases should be derived from the **problem specification**.

Finding all errors requires **exhaustive input testing**, i.e. trying all possible inputs, valid as well as invalid.

`Triangle.java` requires (in principle) infinitely many test cases.

Testing is incomplete (III)

If the source code is accessible (a white-box), then (white-box) test cases may be derived from the **internal structure of the program** (as well as the specification).

The analog to exhaustive input testing is usually chosen to be **exhaustive path testing**.

Testing is incomplete (IV)

```
int i = 0;
while (i++ < 20)
    if (a*a == b*b + c*c) ... // rectangular
    else if (b*b == a*a + c*c) ... // rectangular
    else if (c*c == b*b + a*a) ... // rectangular
    else if (a == b && b == c) ... // equilateral
    else if (a == b) ... // isosceles
    else if (a == c) ... // isosceles
    else if (b == c) ... // isosceles
```

Infeasible to test all paths ($\sim 7^{20}$)

Testing is incomplete (V)

Moreover, exhaustive path testing may not

- guarantee conformance with the program specification.
 - We may have written the wrong program
- detect absence of missing paths
 - What if the case ($a*a == b*b + c*c$) where missing
- uncover data-sensitivity errors
 - what if the case ($a*a == b*b + c*c$) where mistyped ($a*a == a*b + c*c$) and $a = b$ in the test case

Testing strategy

Exhaustive input testing is superior to exhaustive path testing, but neither are feasible strategies.

What subset of all possible test cases has the highest possibility of detecting the most errors?

Combine elements of both black-box and white-box testing to obtain a reasonable (but incomplete) testing strategy.

Structural (white-box) testing

White-box testing is about the degree to which test cases **cover** the source code. Path coverage is the ultimate although (in most cases) an infeasible criteria. Instead we consider:

- Decision/Condition coverage, and
- Loop coverage

Condition coverage

Each **condition** in all **decisions** takes each possible value at least once (and each statement must be covered at least once).

```
if (a > 1 && b == 0) x = x/a;  
if (a == 2 || x > 1) x = x + 1;
```

The conditions are $a > 1$, $b == 0$, $a == 2$, and $x > 1$.

The decisions are the conj. $a > 1 \ \&\& \ b == 0$
and the disj. $a == 2 \ || \ x > 1$

$(a=1, b=0, x=4)$ and $(a=2, b=2, x=1)$ is a condition coverages.

Decision/Condition coverage

Each **decision** and each **condition** in all decisions takes each possible value at least once (and each statement must be covered at least once).

```
if (a > 1 && b == 0) x = x/a;  
if (a == 2 || x > 1) x = x + 1;
```

(a=2, b=0, x=4) and (a=1, b=2, x=1) is a decision/condition coverage.

Loop Coverage

Each **loop** must be executed:

zero, one, and more times

```
for (int i = 0; i < x; i++) {  
    ...  
}
```

(x=0), (x=1) and (x=17) is a loop coverage.

Example

```
//The program receives some integers and prints
  the smallest and largest
int mi, ma;
if (args.length == 0)                                // (1)
    System.out.println("No numbers");
else {
    mi = ma = Integer.parseInt(args[0]);
    for (int i = 1; i < args.length; i++) {          // (2)
        int obs = Integer.parseInt(args[i]);
        if (obs > ma) ma = obs;                       // (3)
        else if (mi < obs) mi = obs;                  // (4)
    }
    System.out.println("Max: " + ma + "Min: mi");
}
```

Choice	Input data	Input property
(1) True	{}	No numbers
(1) False	{17}	At least one number
(2) zero times	{17}	Exactly one number
(2) once	{27,29}	Exactly two numbers
(2) more than once	{49,47,48}	At least 3 numbers
(3) True	{27,29}	obs > ma
(3) False	{39,37}	obs <= ma
(4) True	{49,47,48}	obs <= ma && mi < obs
(4) False	{49,48,47}	obs <= ma && mi >= obs

Input	Expected Output
{}	"No numbers"
{17}	17 17
{27,29}	27 29
{39,37}	37 39
{49,47,48}, {49,48,47}	47 49

Choice	Input data	Input property
(1) True	{}	No numbers
(1) False	{17}	At least one number
(2) zero times	{17}	Exactly one number
(2) once	{27,29}	Exactly two numbers
(2) more than once	{49,47,48}	At least 3 numbers
(3) True	{27,29}	obs > ma
(3) False	{39,37}	obs <= ma
(4) True	{49,47,48}	obs <= ma && mi < obs
(4) False	{49,48,47}	obs <= ma && mi >= obs

Input	Actual Output
{}	"No numbers"
{17}	17 17
{27,29}	27 29
{39,37}	39 39
{49,47,48}, {49,48,47}	49 49

Corrected Example

```
int mi, ma;
if (args.length == 0) // (1)
    System.out.println("No numbers");
else {
    mi = ma = Integer.parseInt(args[0]);
    for (int i = 1; i < args.length; i++) { // (2)
        int obs = Integer.parseInt(args[i]);
        if (obs > ma) ma = obs; // (3)
        else if (obs < mi) mi = obs; // (4*)
    }
    System.out.println("Max: " + ma + "Min: mi");
}
```

Functional (Black-box) testing

Functional testing is about how to,

*based on the **problem specification***, choose a suitable small subset of all possible input tests with a high probability of finding errors

We consider the techniques:

- **Equivalence partitioning**
- **Boundary value analysis**

Equivalence partitioning (I)

The problem specification:

The program reads three integer values, no one greater than 100. The three values are interpreted as representing the lengths of sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

Gives rise to (at least) the following 12 **valid** or **invalid equivalence classes**. Equivalence classes are assigned a unique number and the valid ones are written in boldface.

Equivalence partitioning (II)

Number of inputs: = (1) **3**, (2) < 3 , (3) > 3

Input type: (4) **0 < integer ≤ 100**,

(5) integer ≤ 0 , (6) integer > 100

Scalene: (7) **(a,b,c)** with a, b, c distinct, $a > 0$, $b > 0$, $c > 0$, and with $a + b > c$, $a + c > b$, and $b + c > a$

(8) (a,b,c) with $a > 0$, $b > 0$, $c > 0$, and either $a + b \leq c$, $a + c \leq b$, or $b + c \leq a$

Isosceles: (9) **(a,a,b)**, **(a,b,a)**, or **(b,a,a)**, with a, b distinct, and $a > 0$, $b > 0$, $a + a > b$

(10) (a,b,c) with a, b, c distinct, $a > 0$, $b > 0$, $c > 0$, and with either $a + b > c$, $a + c > b$, or $b + c > a$

Equilateral: (11) **(a,a,a)** with $a > 0$.

(12) (a,b,c) with at least two of a, b, c distinct, $a > 0$, $b > 0$, $c > 0$, and either $a + b > c$, $a + c > b$, or $b + c > a$

Test case identification (I)

- Until all **valid** eq. classes have been covered, write a new test case covering as many of the uncovered valid equivalence classes as possible
- Until all **invalid** eq. classes have been covered, write a test case that covers **one, and only one**, of the uncovered invalid equivalence classes.

Invalid equivalence classes have their own test case because they may otherwise **mask** each other, e.g. (0,1,1,1) may fail due the number of inputs and not due to a zero length.

Test case identification (II)

(5,5,5) covers **(1)**, **(4)**, **(11)**

(3,4,5) covers **(7)**, (10)

(5,5,3) covers **(9)**, (12)

(2) covers (2)

(2,3,4,5,6) covers (3)

(-5,3,6) covers (5)

(3,4,125) covers (6)

(8,2,3) covers (8)

Boundary-value analysis (I)

Boundary conditions have high pay-off

- Rather than selecting *any* element in an equivalence class, select one or more such that each **edge** of the class is the subject of a test.

But what is an edge?

Boundary-value analysis (II)

- 2 and 4 are edges in eq. class (2) and (3)
- 1 and 100 are edges for each input in eq. class (4)
- 0 and 101 are edges for each input in eq. class (5) and (6) respectively
- In (7) (and likewise for (10) and (12)) we should distinguish between
 - (a,b,c) with $a>0, b>0, c>0, a+b>c$
 - (a,b,c) with $a>0, b>0, c>0, a+c>b$
 - (a,b,c) with $a>0, b>0, c>0, b+c>a$
- Similar consideration for (8) and (9)

Boundary-value analysis (III)

Concerning **output conditions**:

- In the triangle example we should make sure that equilateral triangles are not reported as being scalene or isosceles, and similarly that isosceles triangles are not taken for being scalene.

Boundary-value analysis (IV)

$(1,1,1)$, $(100,100,100)$ covers **(1)**, **(4)**, **(11)**

$(1,100,100)$, $(100,1,100)$, $(100,100,1)$ covers **(4)**, **(9)**

$(3,4,5)$ covers **(7)** and **(10)**

$(5,5,3)$, $(3,7,7)$, $(9,3,9)$ covers **(9)**, **(12)**

$(2,3)$ covers **(2)**

$(2,3,4,5)$ covers **(3)**

$(0,0,0)$, $(1,0,1)$, $(1,1,0)$ covers **(5)**

$(101,66,77)$, $(66,101,77)$, $(66,77,101)$ covers **(6)**

$(1,2,3)$, $(1,3,2)$, $(3,2,1)$ covers **(8)**