1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Lecture 18: User-defined exceptions

Why user defined exceptions?

- The built-in exceptions cover almost all the general types of exceptions in programming
- However, in some cases custom exceptions can be beneficial:
  - To catch specific subsets of existing Java exceptions
  - To handle “business logic exceptions” not related to program errors, but e.g., data errors specific to the application
  - Custom exceptions allow handling at specific level of the program
- User defined exceptions can be created simply by inheriting from the existing exceptions.

Example of user defined exception (1)

```
1   import java.util.*;
2   class IntOverflowException extends Exception {
3     public IntOverflowException(String str) {
4         super(str);
5     }
6   }
7   public class TestCustomException {
8     static int fact(int x)
9                   throws IntOverflowException {
10      int y=1;
11      for(int i=1; i<=x; y *= i++) {
12        if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
13          throw new IntOverflowException("integer overflow");
14        }
15      }
16      return y;
17    }
18    static int computeC(int n, int r)
19                  throws IntOverflowException {
20      int res = fact(n)/(fact(r)*fact(n-r));
21      return res;
22    }
```

```
23    public static void main(String args[])
24    {
25      try  {
26        computeC(50,10); // compute C(n,k)
27      }
28      catch (IntOverflowException ex) {
29        System.out.println(ex.getMessage());
30      }
31      System.out.println("continue...");
32    }
33  }
```

Example of user defined exception (2)

```
1   import java.util.*;
2   class IntOverflowException extends Exception {
3     public IntOverflowException(String str) {
4         super(str);
5     }
6   }
7   public class TestCustomException {
8     static int fact(int x)
9                   throws IntOverflowException {
10      int y=1;
11      for(int i=1; i<=x; y *= i++) {
12        if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
13          throw new IntOverflowException("integer overflow");
14        }
15      }
16      return y;
17    }
18    static int computeC(int n, int r)
19                  throws IntOverflowException {
20      int res = fact(n)/(fact(r)*fact(n-r));
21      return res;
22    }
```

```
23    public static void main(String args[])
24    {
25      try  {
26        computeC(50,10); // compute C(n,k)
27      }
28      catch (IntOverflowException ex) {
29        System.out.println(ex.getMessage());
30      }
31      System.out.println("continue...");
32    }
33  }
```

User defined exception. Note that constructor and call to `super()` is not obligatory, but it helps to implement the default functionality.

Example of user defined exception (3)

```
1   import java.util.*;
2   class IntOverflowException extends Exception {
3     public IntOverflowException(String str) {
4         super(str);
5     }
6   }
7   public class TestCustomException {
8     static int fact(int x)
9                 throws IntOverflowException {
10      int y=1;
11      for(int i=1; i<=x; y *= i++) {
12        if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
13          throw new IntOverflowException("integer overflow");
14        }
15      }
16      return y;
17    }
18    static int computeC(int n, int r)
19                throws IntOverflowException {
20      int res = fact(n)/(fact(r)*fact(n-r));
21      return res;
22    }
```

```
23    public static void main(String args[])
24    {
25      try  {
26        computeC(50,10); // compute C(n,k)
27      }
28      catch (IntOverflowException ex) {
29        System.out.println(ex.getMessage());
30      }
31      System.out.println("continue...");
32    }
33  }
```

You need to use keyword `throws` to indicate which methods could throw the custom exception. Alternatively, you can inherit your exception from `RuntimeException`.

Example of user defined exception (4)

```
1   import java.util.*;
2   class IntOverflowException extends Exception {
3     public IntOverflowException(String str) {
4         super(str);
5     }
6   }
7   public class TestCustomException {
8     static int fact(int x)
9                   throws IntOverflowException {
10      int y=1;
11      for(int i=1; i<=x; y *= i++) {
12        if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
13          throw new IntOverflowException("integer overflow");
14        }
15      }
16      return y;
17    }
18    static int computeC(int n, int r)
19                  throws IntOverflowException {
20      int res = fact(n)/(fact(r)*fact(n-r));
21      return res;
22    }
```

```
23    public static void main(String args[])
24    {
25      try  {
26        computeC(50,10); // compute C(n,k)
27      }
28      catch (IntOverflowException ex) {
29        System.out.println(ex.getMessage());
30      }
31      System.out.println("continue...");
32    }
33  }
```

Exception is thrown if variable y (int) will overflow in the next round.

Example of user defined exception (5)

```
1   import java.util.*;
2   class IntOverflowException extends Exception {
3     public IntOverflowException(String str) {
4         super(str);
5     }
6   }
7   public class TestCustomException {
8     static int fact(int x)
9                   throws IntOverflowException {
10      int y=1;
11      for(int i=1; i<=x; y *= i++) {
12        if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
13          throw new IntOverflowException("integer overflow");
14        }
15      }
16      return y;
17    }
18    static int computeC(int n, int r)
19                  throws IntOverflowException {
20      int res = fact(n)/(fact(r)*fact(n-r));
21      return res;
22    }
```

```
23    public static void main(String args[])
24    {
25      try  {
26        computeC(50,10); // compute C(n,k)
27      }
28      catch (IntOverflowException ex) {
29        System.out.println(ex.getMessage());
30      }
31      System.out.println("continue...");
32    }
33  }
```

Our `try…catch` block. From experience, we know that computing *C*(50,10) will cause `int` overflow.

Example of user defined exception (6)

```
1   import java.util.*;
2   class IntOverflowException extends Exception {
3     public IntOverflowException(String str) {
4         super(str);
5     }
6   }
7   public class TestCustomException {
8     static int fact(int x)
9                   throws IntOverflowException {
10      int y=1;
11      for(int i=1; i<=x; y *= i++) {
12        if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
13          throw new IntOverflowException("integer overflow");
14        }
15      }
16      return y;
17    }
18    static int computeC(int n, int r)
19                  throws IntOverflowException {
20      int res = fact(n)/(fact(r)*fact(n-r));
21      return res;
22    }
```

```
23    public static void main(String args[])
24    {
25      try  {
26        computeC(50,10);
27      }
28      catch (IntOverflowException ex) {
29        System.out.println(ex.getMessage());
30      }
31      System.out.println("continue...");
32    }
33  }
```

```
$ java TestCustomException
integer overflow
continue...
$
```

Custom unchecked exception example

```
1   import java.util.*;
2   class IntOverflowException extends RuntimeException {
3   }
4   public class TestCustomException2 {
5     static int fact(int x) {
6       int y=1;
7       for(int i=1; i<=x; y *= i++) {
8         if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
9           throw new IntOverflowException();
10        }
11      }
12      return y;
13    }
14    static int computeC(int n, int r) {
15      int res = fact(n)/(fact(r)*fact(n-r));
16      return res;
17    }
```

```
18    public static void main(String args[])
19    {
20      try  {
21        computeC(50,10);
22      }
23      catch (IntOverflowException ex) {
24        System.out.println("int overflow...");
25      }
26      System.out.println("continue...");
27    }
28  }
```

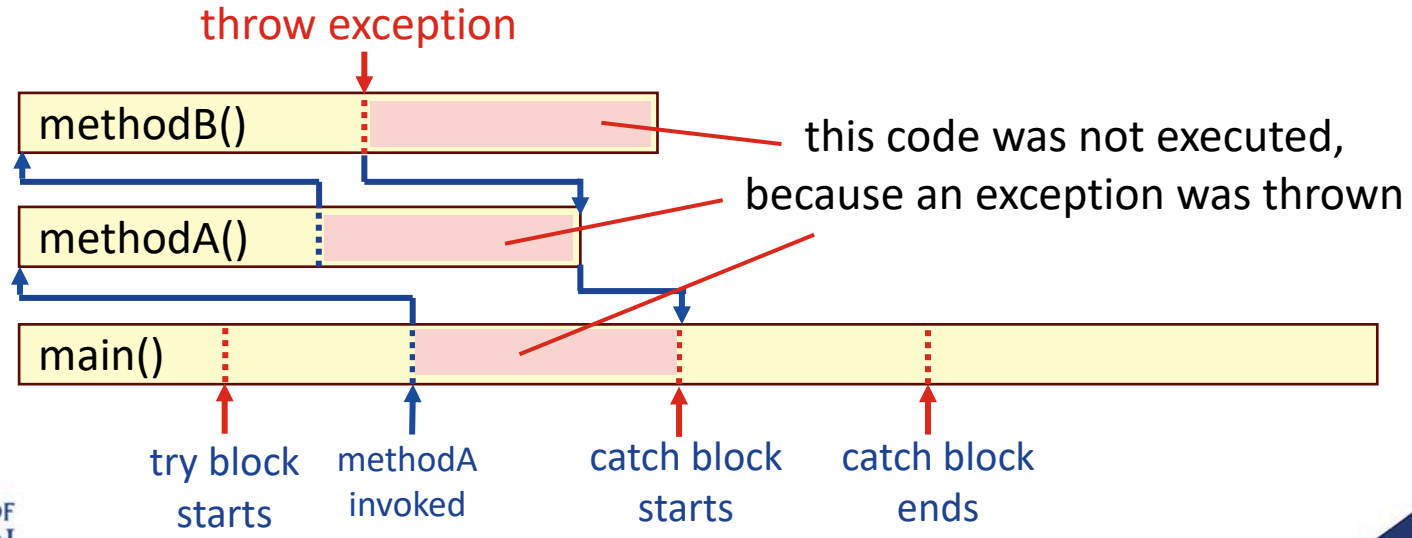
```
$ java TestCustomException2
int overflow
continue...
$
```

Simplified class without constructor inherited from `RuntimeException`

Keyword throws not required for unchecked exceptions

Code ignored due to an exception

- Note that when an exception is thrown, it is propagated through the call stack, until the exception is handled: some data may not be properly initialised!



Variables with no valid data assigned

```
1   import java.util.*;
2   class IntOverflowException extends RuntimeException {
3   }
4   public class TestCustomException3 {
5     static int fact(int x) {
6       int y=1;
7       for(int i=1; i<=x; y *= i++) {
8         if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
9           throw new IntOverflowException();
10        }
11      }
12      return y;
13    }
14    static int computeC(int n, int r) {
15      int res = fact(n)/(fact(r)*fact(n-r));
16      return res;
17    }
```

```
18    static int C;
19    public static void main(String args[])
20    {
21      try  {
22        C = computeC(10,5);
23      }
24      catch (IntOverflowException ex) {
25        System.out.println("int overflow");
26      }
27      System.out.println("C = " + C);
28    }
29  }
```

```
$ java TestCustomException3
int overflow
C = 0
$
```

Because an exception was thrown, `int C` does not have valid value!

UNIVERSITY OF ABERDEEN

Caveats of generic exception handlers

- A generic exception handler catching `Exception` superclass can give misleading information about the underlying problem
  - You should not expect that exceptions are always thrown for the same reason!
- Sometimes custom exceptions are thrown after the code catches a standard exception
  - You should provide a constructor that preserves the details of the error from the standard exception

Misinterpreted exception

```java
import java.util.*;
public class TryCatchTest2 {
  public static void divide() {
    Scanner input = new Scanner(System.in);
    System.out.print("Give x: ");
    int x = input.nextInt();
    System.out.print("Give y: ");
    int y = input.nextInt();
    System.out.println("x / y = " + x/y);
  }
 public static void main(String[] args) {
    try {
      divide();
    }
    catch(Exception e1) {
      System.out.println("y can't be zero!");
    }
  }
}
```

```
Give x: 5
Give y: abc
y can't be zero!
```

In this case, input is not numeric, and the exception thrown is `InputMismatchException`, not `ArithmeticException`

Custom exception example with cause (1)

```
1  import java.util.*;
2  class DivisionException extends Exception {
3    public DivisionException(String msg,
4                             Throwable cause) {
5      super(msg + cause.toString());
6    }
7  }
8  public class CustomExceptionTest4 {
9    public static void divide()
10             throws DivisionException {
11     try {
12       Scanner input = new Scanner(System.in);
13       System.out.print("Give x: "); int x = input.nextInt();
14       System.out.print("Give y: "); int y = input.nextInt();
15       System.out.println("x / y = " + x/y);
16     }
17     catch(Exception e) {
18       throw new DivisionException("division() failed due to ", e);
19     }
20   }
```

```
21   public static void main(String[] args) {
22     try {
23       divide();
24     }
25     catch(DivisionException e) {
26       System.out.println(e.getMessage());
27     }
28   }
29 }
```

Custom exception example with cause (2)

```
1  import java.util.*;
2  class DivisionException extends Exception {
3    public DivisionException(String msg,
4                             Throwable cause) {
5      super(msg + cause.toString());
6    }
7  }
8  public class CustomExceptionTest4 {
9    public static void divide()
10             throws DivisionException {
11     try {
12       Scanner inpu
13       System.out.p
14       System.out.p
15       System.out.p
16     }
17     catch(Exceptio
18       throw new Di
19     }
20   }
```

```
21   public static void main(String[] args) {
22     try {
23       divide();
24     }
25     catch(DivisionException e) {
26       System.out.println(e.getMessage());
27     }
28   }
29 }
```

Define constructor that preserves the cause of the exception (original general exception catched)

Custom exception example with cause (3)

Catch the general exception and throw the custom (business) exception

```
1  import java.util.*;
2
3
4
5
6
7
8  public class CustomExceptionTest4 {
9    public static void divide()
10           throws DivisionException {
11     try {
12       Scanner input = new Scanner(System.in);
13       System.out.print("Give x: "); int x = input.nextInt();
14       System.out.print("Give y: "); int y = input.nextInt();
15       System.out.println("x / y = " + x/y);
16     }
17     catch(Exception e) {
18       throw new DivisionException("division() failed due to ", e);
19     }
20   }
```

```
21   public static void main(String[] args) {
22     try {
23       divide();
24     }
25     catch(DivisionException e) {
26       System.out.println(e.getMessage());
27     }
28   }
29 }
```

Custom exception example with cause (4)

```
1  import java.util.*;
2  class DivisionException extends Exception {
3    public DivisionException(String msg,
4                             Throwable cause) {
5      super(msg + cause.toString());
6    }
7  }
8  public class CustomExceptionTest4 {
9    public static void divide()
10             throws DivisionException {
11     try {
12       Scanner input = new Scanner(System.in);
13       System.out.print("Give x: "); int x = input.n
14       System.out.print("Give y: "); int y = input.n
15       System.out.println("x / y = " + x/y);
16     }
17     catch(Exception e) {
18       throw new DivisionException("division() failed due to ", e);
19     }
20   }
```

```
21   public static void main(String[] args) {
22     try {
23       divide();
24     }
25     catch(DivisionException e) {
26       System.out.println(e.getMessage());
27     }
28   }
29 }
```

Catch the custom (business) exception and print out the underlying cause exception

Custom exception example with cause (5)

```
1  import java.util.*;
2  class DivisionException extends Exception {
3    public DivisionException(String msg,
4                             Throwable cause) {
5      super(msg + cause.toString());
6    }
7  }
8  public class CustomExceptionTest4 {
9    public static void divide()
10             throws DivisionException {
11     try {
12       Scanner input = new Scanner(System.in);
13       System.out.print("Give x: "); int x = input.n
14       System.out.print("Give y: "); int y = input.n
15       System.out.println("x / y = " + x/y);
16     }
17     catch(Exception e) {
18       throw new DivisionException("division() faile
19     }
20   }
21   public static void main(String[] args) {
22     try {
23       divide();
24     }
25     catch(DivisionException e) {
26       System.out.println(e.getMessage());
27     }
28    }
29 }
```

```
$ java CustomExceptionTest4
Give x: 56
Give y: 0
Division failed due to
java.lang.ArithmeticException: / by zero
$ java CustomExceptionTest4
Give x: abc
Division failed due to
java.util.InputMismatchException
```

Summary

- Variables in Java are either *primitive* or *reference* type variables
  - Primitive type variables contain the value directly, reference type variables refer to the variable data stored elsewhere in the memory
- In Java, variables are passed to the methods *by value*
  - Invoked method makes a copy of the variable
- Calls to methods are stored in a *call stack* in first-in-first-out manner
- In Java, errors and other exceptional situations throw exceptions handled by `try…catch` structure
  - User defined (custom) exceptions can also be defined and handled

Questions, comments?