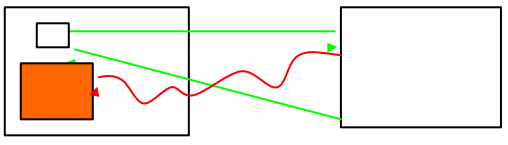
Exceptions in Java

- **Unvorhergesehene Zustände können immer auftreten**
  - Datei nicht vorhanden die gelesen werden soll
  - Verbindung zum Server bricht ab
  - Dateneingabe falsch
  - Zugriff auf Feld mit falschem Index
- **Reaktion?**
  - Programmabbruch
    - Keine Möglichkeit zu reagieren
  - Resultatparameter (boolean, null)
    - müssen getestet werden / können ignoriert werden
    - führen zu "geschachteltem" Code
    - java unterstützt die Rückgabe von mehreren Resultaten schlecht (keine ref-para)
  - Exceptions
    - strukturierte Behandlung von Fehlern

Exceptions in Java

- **Exception**
  - Fehler (oder vom Entwickler gewollte Bedingung) löst Ausnahme aus
  - Kontrollierter Sprung an Stelle im Programm an welcher der Fehler behandelt wird (Exception Handler)



  - Aufrufer kann nach dem Behandeln der Ausnahme
    - auf andere Art fortfahren, Aktion wiederholen
    - Ausnahme weitergeben oder eigene Ausnahme auslösen
    - Programm abbrechen

Beispiel: Bankzugriff mit Fehlercodes

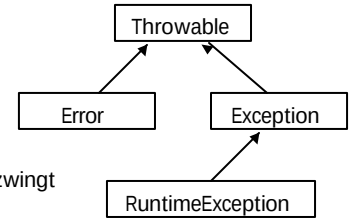
```
static boolean withdraw( String name, String nr, double val){
   boolean ok = false;
   Bank bank = Banks.connectBank(name);
   if (bank != null){
      Account account = bank.getAccount(nr);
      if(account != null){
         int err = account.withdraw(2560.00);
         ok = err == 0;
         if( !ok ){ System.err.println(getErrorText(err)); }
      }
      else {
         System.err.println("Konto nicht vorhanden");
      }
   }
   else {
      System.err.println("Bank nicht vorhanden");
   }
   return ok;
}
```


Beispiel: Bankzugriff mit Ausnahmen

```
static boolean withdraw(String name, String nr, double val){
   try {
      Bank bank = Banks.connectBank(name);
      Account account = bank.getAccount(nr);
      account.withdraw(2560.00);
      return true;
   }
   catch(LimitOverflowException e){
      System.err.println("Saldo zu klein");
   }
   catch(AccountException e) {
      System.err.println("Konto nicht vorhanden");
   }
   catch(BankException e) {
      System.err.println("Bank nicht vorhanden");
   }
   return false;
}
```

Ausnahmen

- **Klassen**
  - **Error**
    - schwerwiegende Fehler
    - sollten nicht behandelt werden
  - **Exception**
    - Fehler deren Behandlung der Compiler erzwingt
  - **RuntimeException**
    - Programmierfehler
    - Compiler erzwingt deren Behandlung nicht
    - Dürfen ohne "Ankündigung" geworfen werden
  - **Throwable**: Methoden
    - getMessage() Fehlertext
    - printStackTrace() Schreibt Aufrufstack auf Console
    - toString() Fehlerobjekt in lesbarer Form



Ausnahmen in der Java Library

- **RuntimeException**
  - NullPointerException
  - IndexOutOfBoundsException
  - ClassCastException
  - ArithmeticException
  - IllegalArgumentException
  - SecurityException
  - SystemException
  - UnsupportedOperationException
  - NoSuchElementException
- **Exception**
  - ClassNotFoundException
  - AlreadyBoundException
  - ApplicationException
  - AWTException
  - BadLocationException
  - CloneNotSupportedException
  - DataFormatException
  - ExpandVetoException
  - GeneralSecurityException
  - IllegalAccessException
  - InstantiationException
  - InterruptedException
  - IntrospectionException
  - IOException

Definition von neuen Ausnahmen

- **Ausnahme = Java Klasse**
  - Unterklasse von **Exception** oder **RuntimeException** (oder einer existierenden Exceptionklasse)
  - Üblicherweise werden folgende Konstruktoren bereitgestellt:
    - Default Konstruktor
    - Konstruktor mit String Argument (=Fehlerbeschreibung)
    - Konstruktor mit Throwable Argument (=Verkettung) **[1.4]**
    - Konstruktor mit String und Throwable Argument **[1.4]**
  - Klasse darf weitere, Exception-spezifische Methoden anbieten

Definition von neuen Ausnahmen

- **LimitOverflowException**

```
class LimitOverflowException extends Exception {
   private double limit;
   public  double getLimit(){return limit;}
   public  void   setLimit(double limit){this.limit = limit;}

   public LimitOverflowException(){ }
   public LimitOverflowException (String s){super(s);}
   public LimitOverflowException (String s, Throwable cause){
      super(s, cause);
   }
}
```

Behandlung von Ausnahmen

- **try-catch-finally Anweisung**

  try { … } **catch** (Exception e) { … }

  - try-Block enthält eine oder mehrere Anweisungen
  - sobald ein Fehler auftritt wird Programmausführung unterbrochen
  - catch-Block welcher der Ausnahme entspricht wird ausgeführt
  - es wird nur *ein* catch-Block ausgeführt
  - im catch-Block steht die Ausnahme als Parameter zur Verfügung

```
try {
    n = Integer.parseInt(s);
}
catch(NumberFormatException e){
    System.err.println(s + " ist keine gültige Zahl");
    n = 0;
}
```


Behandlung von Ausnahmen

- **mehrere catch-Blöcke**
  - es können mehrere catch-Blöcke angegeben werden
  - ausgeführt wird der *erste* Block, dessen Argument der aufgetretenen Ausnahme entspricht (d.h. Exception ? Argument-Typ)

```
try {
   n = Integer.parseInt(s);
}
catch(NumberFormatException e){
   System.err.println(s + " ist keine gültige Zahl");
   n = 0;
}
catch(NullPointerException e){
   System.err.println("String nicht definiert");
}
catch(Exception e){
   // catch all
}
```

Behandlung von Ausnahmen

- **finally-Block**
  - der finally-Block enthält Anweisungen welche **immer** ausgeführt werden wenn der try-Block betreten wurde, unabhängig davon, ob eine Ausnahme auftritt oder nicht; wird verwendet um Aufräumarbeiten auszuführen

```
FileReader in = null;
try {
   FileReader in = new FileReader("test.txt");
   int ch = in.read();
   while(ch != -1){consume(ch); ch = in.read();}
   return true;
}
catch(ConsumptionException e){
   System.out.println(e); return false;
}
finally{
   if(in != null) in.close();
}
```


Deklaration von Ausnahmen

- **Ausnahmen müssen im Methodenkopf deklariert werden**

```
public void withdraw(float amount)
        throws LimitOverflowException {
    …
}
```

  - *Ausnahme*: Exceptions die von der Klasse RuntimeExpeption abgeleitet sind müssen nicht deklariert werden

- **Compiler prüft, ob deklarierte Ausnahmen behandelt werden**
  - Klient muss alle Ausnahmen behandeln (oder diese selber als mögliche Ausnahmen deklarieren)
  - *Ausnahme*: Exceptions die von der Klasse RuntimeException abgeleitet sind müssen nicht behandelt werden

Werfen von Ausnahmen

- **Werfen einer neuen Exception**
  - **throw** new LimitOverflowException(“Limit too small (“+limit +”)”);
- **Weitergeben einer geworfenen Exception**
  - **explizit**

```
try{
    account.withdraw(1234);
}
catch(LimitOverflowException e){
    Logger.logException(e);
    throw e;
}
```


Werfen von Ausnahmen

- **Weitergeben einer geworfenen Exception**
  - **implizit**

```
boolean withdraw(String name, String nr, double val)
    throws BankException,AccountException,
           LimitOverflowException {
   Bank bank = Banks.connectBank(name);
   Account account = bank.getAccount(nr);
   return account.withdraw(2560.00);
}
```

- **try-finally**

```
Object read (InputStream in) throws IOException {
   try {
     in = new ObjectInputStream(in);
     return in.readObject();
   }
   finally{
      in.close();
   }
}
```

Verlorene Ausnahmen

- **No return statements in a finally block!**

```
int foo(){
   try {throw new NullPointerException();}
   finally {return 0;}
}
```

  - Was ist das Resultat von foo() ?

- **No throw statements in a finally block!**

```
void deposit(){
   try {throw new LimitOverflowException();}
   finally { throw new IllegalArgumentException(); }
}
```

  - Welche Ausnahme wird geworfen wenn deposit() aufgerufen wird?

Exceptions & Vererbung

- **Polymorphismusgedanke**
  - Was gilt, wenn eine Methode in einer Unterklasse überschrieben wird
    - müssen alle Ausnahmen der überschriebenen Methode in der thows Deklaration aufgeführt werden?
    - dürfen Ausnahmen weggelassen werden?
    - dürfen neue Ausnahmen deklariert werden welche spezifisch für die Spezialisierung sind?

- **Mehrfachvererbung auf Schnittstellen**
  - Was gilt, wenn eine Methode mit derselben Signatur aus verschiedenen Schnittstellen geerbt wird und unterschiedliche Ausnahmen deklariert?