# Programmierung 2

Kapitel 6 – Exceptions

Vielen Dank an Prof. Erik Buchmann
für die beigesteuerten Folien

# Definition

- Exception = Kurzform für "exceptional event"
- Exception ist ein Ereignis das während der Programmausführung auftritt und die normale Programmausführung unterbricht
- Arten von Exceptions
  - Errors
  - Exceptions
  - Runtime Exceptions

# Motivation

```
public class Mathematik {

    // Rekursive Berechnung der Fakultät einer Zahl
    public static int fak(int n) {
        if (n > 0) {
            return n * fak(n-1);
        } else {
            return 1;
        }
        // Was tun bei ungültigem n < 0?
    }

    public static double div(double wert, double durch) {
        if (durch != 0.0) {
            return wert/durch;
        } else {
            // zwar falsch, aber wie sonst mit dem Fehler umgehen?
            return 0.0;
        }
    }
}
```

# Fehlerquellen sind vielfältig

- Inkonsistente Variablenwerte, ungültige Parameterwerte
- Falsche Netzwerkdressen / ungültige Dateinamen
- Nicht erfüllte Vorbedingungen
  (Datei lesen erst nach Datei öffnen)
- ungültiger Array-Index,
  Zugriff auf Objekt über Objektvariable mit Wert null
- arithmetische Fehler
- kein Speicherplatz mehr vorhanden, Endlosrekursion
- Ressourcenkonflikte, mehrere parallele Tasks versuchen gleichzeitig dasselbe Objekt zu ändern
- Fehlende Berechtigungen, Hintergrundprozess versucht auf Bildschirm zu schreiben
- ...

# Umgang mit Fehlern?

- Schwere Fehler

  → Kontrollierter Programmabbruch
  - Z.B. kein Hauptspeicher mehr verfügbar
  - In Java: System.exit(-1);

- Leichte Fehler

  → Fehlerbehebung im Programm
  - Z.B. Nutzer um korrigierte Eingabe bitten, auf belegte Ressource warten, Netzwerkverbindung wiederherstellen, …

- Situationsabhängig, was schwere und leichte Fehler sind

  (kann ich Hauptspeicher selbst freigeben?)

# Bisher: Ungültigen Rückgabewert nutzen

- Aufrufer muss auf Rückgabe = -1 testen, korrekte Fakultät (n!) kann niemals den Wert -1 annehmen

```
public class Mathematik {

    // Rekursive Berechnung der Fakultät einer Zahl
    public static int fak(int n) {
        if (n > 0) {
            return n * fak(n-1);
        } else {
            if (n == 0) {
                return 1;
            } else {
                // Fehler
                return -1;
            }
        }
    }
}
```

Aufrufer darf nicht „einfach so“
mit dem Rückgabetyp weiterrechnen, sondern muss erst noch Fehlerprüfung durchführen
→ fehlerträchtig

# Funktioniert nicht immer

- Wertebereich voll ausgenutzt

```
public static double div(double wert, double durch) {
    if (durch != 0.0) {
        return wert/durch;
    } else {
        // zwar falsch, aber wie sonst mit dem Fehler umgehen?
        return 0.0;
    }
}
```

- Aufrufer kann nicht zwischen fehlerhaftem und fehlerlosem Ergebnis unterscheiden

# Idee: Ausgabe einer Fehlermeldung?

- Aufrufer muss Konsolenausgabe überwachen, Texte interpretieren
  → langsam, komplex und fehlerträchtig

```
public static int fak(int n) {
    if (n > 0) {
        return n * fak(n-1);
    } else {
        if (n == 0) {
            return 1;
        } else {
            // Fehlermeldung ausgeben
            System.err.println("ungueltiger Parameter");
            return -1;
        }
    }
}
```

# Idee: Globale Fehlervariable?

- Aufrufer muss sich kümmern → umständlich und fehlerträchtig
- nur eine static Variable pro Klasse → nicht parallelisierbar

```
private static boolean error; // Error-Variable

public static int fak(int n) {
    error = false;
    if (n > 0)  {
        return n*fak(n-1);
    }
    if (n == 0) {
        return 1;
    }
    error = true;
    return -1;
}
public static boolean hasError() {
    return error
};
```

# Fehler in Java

# Fehler in Java

1. Exception wird ausgelöst: Fehler tritt während der Ausführung einer Methode auf

   - Methode erzeugt ein Exception Objekt und gibt es an die Laufzeitumgebung
   - Exception Objekt beinhaltet Informationen über den aufgetretenen Fehler
     - Fehlertyp
     - Zustand des Programms als der Fehler auftrat, z.B. Stelle im Programm wo der Fehler auftrat

# Fehler in Java

2. Laufzeitumgebung sucht einen Weg um die Exception zu behandeln

   - Ablauf des Call Stack von Methode mit Fehler bis
     - zu einem passenden Exception Handler oder
     - der main-Methode des Programms

   - Falls Handler gefunden, wird dieser ausgeführt
     - Man nennt die Exception nun abgefangen

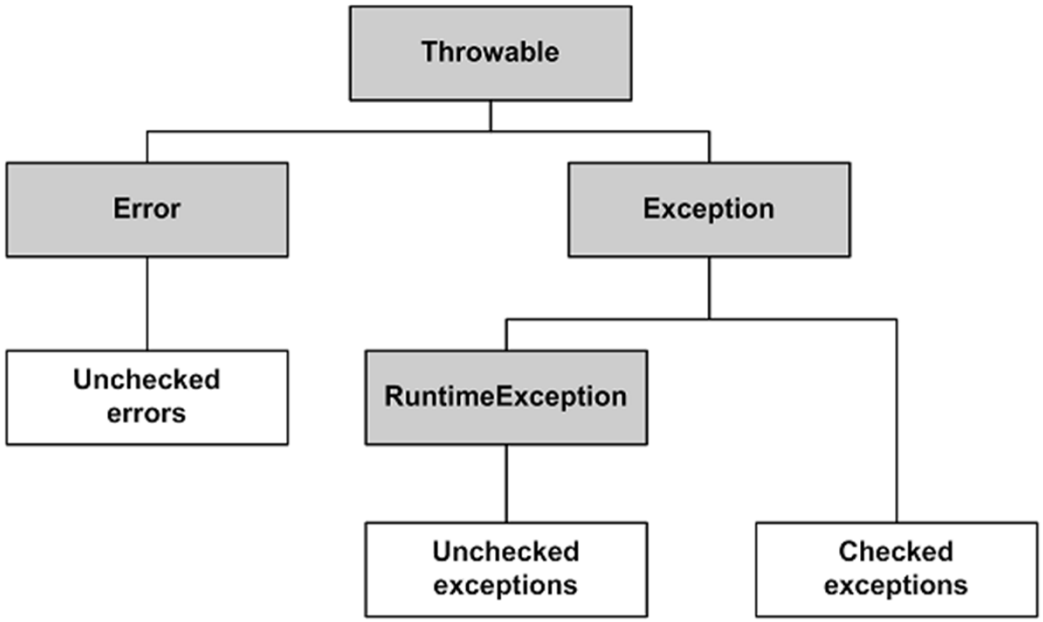
Falls kein Handler gefunden -> Laufzeitumgebung beendet Programm

# Abfangen oder Weiterleiten

- Code der evtl. eine Exception aufwirft muss von einem der beiden Möglichkeiten umschlossen sein
  - **try**-Block beschreibt wie mit Fehler umgegangen wird
  - **throws** Deklaration spezifiziert, dass Exception weitergeleitet wird

- Code der gegen die sogenannte "Catch or Specify" Anforderung verstößt, kann nicht erfolgreich kompiliert werden
  - Aber, nicht alle Exceptions sind an die Anforderung gebunden

# Fehlerklassen



# Checked Exceptions

- Checked Exceptions bedienen die "Catch or Specify" Anforderung
- Fehler die innerhalb des Programm zur Laufzeit auftreten
  - Z.B. IOException
- Typischerweise Fehler, die ein Programmierer vorhersehen und dementsprechend anfangen kann
- Checked Exceptions werden zur Compile-Zeit getestet

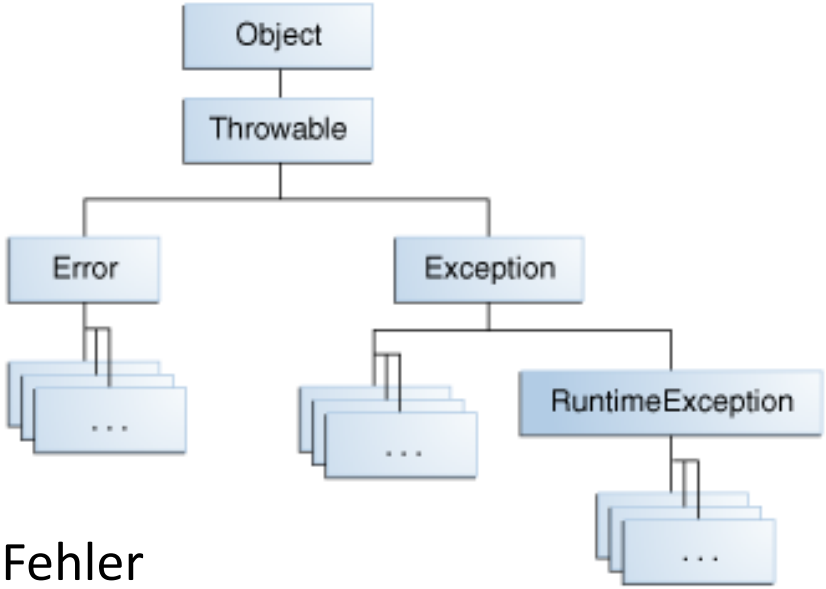
# Unchecked Exceptions: Errors

- Unchecked Exceptions sind nicht an die "Catch or Specify" Anforderung gebunden
- Schwerwiegende Fehler die außerhalb der Anwendung verursacht werden und schwer vorhersehbar sind
  - Hardwarefehler
  - Fehler während der Ausführung von Betriebssystem-Funktionen
- Behandlung
  - Benachrichtigung des Nutzers oder
  - Abbruch des Programms
- Einfache Programme behandeln Errors typischerweise nicht
- Unchecked Exceptions werden nicht zur Compile-Zeit getestet

# Unchecked Exceptions: Runtime Exceptions

- Runtime Exceptions sind nicht an die "Catch or Specify" Anforderung gebunden
- Schwerwiegende Fehler die innerhalb der Anwendung auftreten und schwer vorhersehbar/behebbar sind
  - Logikfehler oder fehlerhafte Anwendung einer API
  - NullPointerExceptions, ArrayIndexOutOfBoundsException
- Behandlung
  - Anwendungen können diese Fehler abfangen
  - Allerdings kann mit der Entfernung des verursachenden Bugs, der Fehler vermieden werden
- Unchecked exceptions werden nicht zur Compilezeit getestet

# Zusammengefaßt




- Fehlerobjekte (Exceptions)
  - sind Objekte von Fehlerklassen
  - alle von Throwable abgeleitete Klassen
  - Error und RuntimeException: schwerwiegende Fehler
- Klasse Exception: hier können Sie eigene Fehlerklassen ableiten
  - Name einer Fehlerklasse sollte aussagekräftig sein!

```
package java.lang;
public class Exception extends Throwable {
    public Exception();
    public Exception(String message);
    public String getMessage(); // liefert Beschreibung
    public String toString();   // ruft getMessage auf }
```

# Fehlerbehandlung in Java

# Fehlerbehandlung in Java

- Bei der Klassen-/Methodendefinition:
  - überlegen, was für Fehler auftreten können
  - Definition und Implementierung geeigneter Fehlerklassen
  - Erweiterung der Methodensignatur um Angabe möglicher Fehler
  - erzeugen und liefern von Fehlerobjekten

- Bei der Klassen-/Methodennutzung:
  - beim Methodenaufruf ermitteln, welche Fehler auftreten können
  - Methode auszuführen
  - falls kein Fehler auftritt, normal weitermachen
  - mögliche Fehlerobjekte abfangen, Fehlerbehandlung

# Auf einen Blick

- Ableitung einer Exception

```
public class InvalidParameter extends Exception {
    public InvalidParameter() {
        super("Ungültiger Parameter");
    }
}
```

- Erweiterung der Methode, die den Fehler produzieren könnte

```
public static int fak(int n) throws InvalidParameter {
    if (n > 0) return n * fak(n-1);
    if (n == 0) return 1;
    throw new InvalidParameter();
}
```

- Abfangen des Fehlers beim Aufruf der Methode

```
try {
    System.out.println ("Die Fakultät ist "+fak(zahl));
} catch (InvalidParameter e) {
    // Fehlerbehandlung
}
```

# Beispiel: Berechnung der Fakultät

- Ursprüngliche Methode:
- Fehler: n<0

```
public static int fak(int n) {
    if (n > 0) return n * fak(n-1);
    if (n == 0) return 1;
    // Fall n<0 abfangen
}
```

- Ableitung einer passenden Fehlerklasse von Exception

```
public class InvalidParameter extends Exception {
    // fehlerhafter Parameter
    int value;
    // Konstruktor
    public InvalidParameter(int value) {
        super("Ungültiger Parameter: " + value);
        this.value = value;
    }
    // fehlerhaften Wert zurückgeben
    public int getErrValue() {
        return value;
    }
```

# Erweitern Methode um die Exception

- Ursprüngliche Methode:

```
public static int fak(int n) {
    if (n > 0) return n * fak(n-1);
    if (n == 0) return 1;
    // Fall n<0 abfangen
}
```

- Methode „wirft“ die eben definierte Exception
- throws in der Signatur
- throw new Exception im Methodenrumpf

```
public static int fak(int n) throws InvalidParameter {
    if (n > 0) return n * fak(n-1);
    if (n == 0) return 1;
    throw new InvalidParameter(n);
}
```

# Fehlerbehandlung

- Ursprüngliche Methode

```
static void berechneFakultaet() {
    Scanner sc = new Scanner();
    int zahl = Integer.parseInt(sc.nextLine());
    int result = fak(zahl);
    System.out.println ("Die Fakultät ist "+result);
}
```

- Abfangen des Fehlers mit **try**
- Behandeln des Fehlers mit **catch**

```
static void berechneFakultaet() {
    int zahl = Integer.parseInt(new Scanner().nextLine());
    try {
        int result = fak(zahl);
        System.out.println ("Die Fakultät ist "+result);
    } catch (InvalidParameter e) {
        System.err.println("Ungültige Eingabe: "+e.getErrValue());
        // erneuter Versuch
        berechneFakultaet();
    }
}
```

# Fehlertypdeklaration

- Deklaration von Fehlertypen einer Methode in der Methodensignatur mittels Schlüsselwort **throws**
- Eine Methode muss alle Fehlertypen deklarieren, die sie werfen kann, oder sie muss sie selbst abfangen

```
public class Ex1 extends Exception {}
public class Ex2 extends Exception {}
public int f() throws Ex1, Ex2 {
    ... if (...) throw new Ex1();
    ... if (...) throw new Ex2();
}
public int g() throws Ex1 {
    try { ... int i = f(); ... }
    catch (Ex2 e) { ... }
}
```

- Fehlerobjekte werden "weitergeleitet" an aufrufende Methode
- Error- und RuntimeException-Fehler müssen nicht deklariert werden

# Überschreiben von Fehlertypdeklaration

- Die Fehlertypdeklaration gehört mit zur Signatur einer Methode
- Methoden mit Fehlertypen in abgeleiteten Klassen überschreiben
  - entweder: Fehlertypdeklarationen stimmen überein
  - oder: Fehlerklassen angeben, die von den Fehlerklassen der überschriebenen Methode abgeleitet sind

```
public class Ex1 extends Exception {}
public class Ex2 extends Exception {}
public class Ex3 extends Ex2 {}

public class A {
    public int f(float v) throws Ex1, Ex2 { ... }
}

public class B extends A { //eine der Alternativen für f()
    public int f(float v) throws Ex1, Ex2 { ... }
    public int f(float v) throws Ex1, Ex3 { ... }
}
```

# Abfangen von Fehlerobjekten

```
try {
    //...
} catch (XException obj) {
    //...
} catch (YException obj) {
    //...
} finally {
    //...
}
```

- Fehlerobjekt abfangen
  - Nur innerhalb eines **try**-Blocks
- Fehlerbehandlung
  - In catch-Block mit dem passenden Fehlerobjekt
  - Wird Fehlerobjekt geworfen (**throw**), so wird **try**-Block abgebrochen
- Existiert **catch**-Block mit passendem Fehlerobjekt, wird dieser ausgeführt
- Existiert ein **finally**-Block, wird dieser immer ausgeführt
- Kein **catch**-Block mit passendem Fehlerobjekt?
  - **finally**-Block ausführen falls vorhanden
  - aktuelle Methode abbrechen
  - Fehlerobjekt an aufrufende Methode weiterleiten, bis zu main()

# Catch-Block

- Vergleichbar zu Methodenaufruf mit einen Objekt als Parameter
- Muss **try**-Block oder anderen **catch**-Block folgen
- Parameter ist ein von Klasse Throwable abgeleitetes Objekt, das im **try**-Block geworfen werden kann
- Fehlerobjektübergabe identisch zu Parameterübergabe bei Methoden, d.h. Fehlerobjekte
  - werden wie lokale Variablen des **catch**-Blockes behandelt
  - werden mit dem geworfenen Fehlerobjekt initialisiert
- **catch**-Block passt zu einem Fehlerobjekt
  - Wenn die Klasse des Fehlerobjektes wie angegeben
  - Wenn die Klasse eine Oberklasse des Fehlerobjektes ist (Polymorphie)

# Finally-Block

- Parameterlos
- Schließt einen try-{catch}-Block ab
  - hinter keinem/einem/einer Menge von catch-Blöcken definiert
- Ist ein finally-Block hinter einem try-{catch}-Block vorhanden, so wird dieser immer ausgeführt, d.h.
  - wenn kein Fehlerobjekt im try-Block geworfen wurde
  - wenn ein Fehlerobjekt im try-Block geworfen und nicht abgefangen wurde
    - nach dem finally-Block wird in diesem Fall die Funktion verlassen und das Fehlerobjekt an die aufrufenden Funktion weitergeleitet
  - wenn Fehlerobjekt im try-Block geworfen und abgefangen
    - nach der Ausführung des passenden catch-Blockes

# Anmerkungen

- In einem try-Block geworfene Exceptions können vom anschließenden catch-Block direkt wieder gefangen werden
- try-{catch}-finally- Blöcke lassen sich schachteln
- Sofortiger Abbruch eines Programms:

```
class Ex1 extends Exception {...}

public void f() {
    try {
        ... throw new Ex1();
        ...
    } catch (Exception obj) {
        ...
    }
}

try {
    ...
} catch (Exception o1) {
    try {
        ...
    } catch (Exception o2) {
        ...
    }
}

System.exit(-1);
```

# Stack Trace und Logging

- Sammeln von Informationen über fehlerhafte Programmabläufe in einer Log-Datei
  - Auswertung komplexer Programmabläufe
  - Auffinden versteckter Abhängigkeiten

- java.util.logging API

```
try {
    Handler handler = new FileHandler("OutFile.log");
    Logger.getLogger("").addHandler(handler);
}
catch (IOException e) {
    Logger logger = Logger.getLogger("package.name");
    StackTraceElement elements[] = e.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        logger.log(Level.WARNING, elements[i].getMethodName());
    }
}
```

# Zusammenfassung

- Exceptions
  - zur Behandlung von potentiellen Laufzeitfehlern
  - Trennung von "Normalfall" und möglichen Fehlerfällen
  - Instanzen von Klassen, die von Throwable abgeleitet sind

- Im Fehlerfall wird von einer Methode eine Exception geworfen

- Aufrufende Methoden können diese Exception abfangen und eine Fehlerbehandlung einleiten

# Kontrollfragen

- Warum ist es keine allgemein praktikable Vorgehensweise, Fehlersituationen über (globale) Variablen mit Fehlercodes oder über ungültige Rückgabewerte einer Funktion an den Aufrufer zu kommunizieren?
- Beschreiben Sie den Ablauf beim Auslösen und Verarbeiten einer Exception.
- Nennen Sie alle Bedingungen, die erfüllt sein müssen damit ein bestimmter catch-Block ausgeführt wird.
- Welche Exceptions müssen unter welchen Umständen mit throws bei der Deklaration einer Methode mit angegeben werden, welche nicht?