JAVA

Weiterführende Spracheigenschaften

AGENDA

- Strings
- Exceptions
- Enums
- Generics

DIE KLASSE STRING

- Zeichenketten werden in Java als String repräsentiert
- Wie der Typ char sind auch Strings Unicode-fähig
- String hat viele Methoden zur Manipulation
- Keine Kenntnis über inneren Aufbau von Nöten

Vergleiche: http://commons.apache.org

1. - Ein String entspricht einer Reihung von Elementen des Typs char.
   -> Wie in anderen Sprachen auch.
   - Wiederholung: Wie groß ist ein char?
   - char Arrays sollten in Java nur noch in Ausnahmefällen verwendet werden.
3. - String liefert eine Vielzahl von Methoden zur Manipulation und Bestimmung von Eigenschaften der Zeichenkette mit.
   -> Mehr dazu gleich
   - Trotz dieser großen Vielzahl von Methoden gibt es eine Sammlung an Helfermethoden, welche einem den Umgang mit Strings noch weiter erleichtern.
   -> Diese sind aber leider kein Bestandteil dieser Vorlesung ;-). Vgl. http://commons.apache.org
4. - Anders als zum Beispiel bei C muss sich der Entwickler keine Gedanken über den inneren Aufbau von Strings Gedanken machen.
   - Keine Gedanken „verschwenden“, ob man es aktuell mit einem leeren Array von chars zu tun hat.
   - Weiterer Grund: Der innere Aufbau von Strings sind weitestgehend anonym und bedeutungslos für den Entwickler -> „Strings funktionieren einfach“.

WICHTIGSTE METHODEN VON STRINGS (1)

```
public StringBeispiel() {
    // Konstruktoren:

    // Ein leerer String
    String leerer = new String();
    // Duplizierung
    String duplikat = new String("Hallo");
    // Mittels char-Arrays
    char[] array = { 'H', 'a', 'l', 'l', 'o' };
    String charArray = new String(array);

    // Zeichenextraktion:

    // char charAt(int index)
    // Liefert das Zeichen am angegebenen nullbasierten Index.
    char charAt = duplikat.charAt(0);
    // String substring(int begin, int end)
    // Liefert den Teilstring von inklusiv Index begin bis exklusiv Index end.
    String subString1 = duplikat.substring(1, 3);
    // Eine Variante der Methode ohne end liefert stets den Teilstring bis zum Ende.
    String subString2 = duplikat.substring(3);
    // String trim()
    // Entfernt Leerzeichen am Anfang und Ende
    String trim = "    Hallo    ".trim();

    // Übung:
    System.out.println(leerer);
    System.out.println(duplikat);
    System.out.println(charArray);
    System.out.println(charAt);
    System.out.println(subString1);
    System.out.println(subString2);
    System.out.println(trim);
}
```


Ergebnis:

"

Hallo
Hallo
H
al
lo
Hallo

"

WICHTIGSTE METHODEN VON STRINGS (2)

```
public StringBeispiel2() {¶
    String string1 = "Hallo!";¶
    String string2 = "Wie gehts?";¶
    AudiQFuenf q5 = new AudiQFuenf();¶

    // Länge der Zeichenkette:¶
    ¶
    // int length()¶
    // Liefert die aktuelle Länge. Der Rückgabewert 0 bedeutet Leerstring.¶
    int length = string1.length();¶

    // Vergleich von Zeichenketten:¶

    // boolean equals(Object obj)¶
    // Analog zur gleichnamigen Methode der Klasse Object.¶
    // Vergleicht zwei Strings auf inhaltliche Gleichheit.¶
    boolean equals1 = string1.equals(string2);¶
    boolean equals2 = string1.equals("Hallo!"); // Was sollte man hier beachten?¶

    // Vergleich mit der String-Darstellung beliebiger Objekte möglich,¶
    // indem zuvor obj .toString() gerufen wird.¶
    boolean equals3 = q5.toString().equals("Ist das ein Q5?");¶

    // boolean equalsIgnoreCase(String s)¶
    // Wie equals, ignoriert jedoch die Unterschiede in Groß-/ Kleinschreibung.¶
    boolean equals4 = "hAllo!".equalsIgnoreCase(string1);¶
    ¶
    // Übung:¶
    System.out.println(length);¶
    System.out.println(equals1);¶
    System.out.println(equals2);¶
    System.out.println(equals3);¶
    System.out.println(equals4);¶
}¶
```


Ergebnis

"
6
false
true
false
true
"

WICHTIGSTE METHODEN VON STRINGS (3)

```
public StringBeispiel3() {¶
    String string1 = "Hallo!";¶
    String stringA = "a";¶
    String stringB = "b";¶

    // Vergleich von Zeichenketten:¶

    // boolean startsWith(String s)¶
    // Testet, ob ein String mit der angegebenen Zeichenkette beginnt.¶
    boolean startsWith = string1.startsWith("Ha");¶
    // boolean endsWith(String s)¶
    // Testet, ob ein String mit der angegebenen Zeichenkette endet.¶
    boolean endsWith = string1.endsWith("o!");¶
    // int compareTo(String s)¶
    // Lexikalischer Vergleich beider Strings durch paarweisen Vergleich der einzelnen Zeichen¶
    // von links nach rechts. Ist der aktuelle String kleiner als s, wird ein negativer Wert¶
    // zurückgegeben. Ist er größer, wird ein positiver Wert zurückgegeben. Bei Gleichheit ist¶
    // der Rückgabewert 0. --> Wichtig für Sortierung und Collections!¶
    int compare1 = stringA.compareTo(stringB);¶
    int compare2 = stringB.compareTo(stringA);¶
    int compare3 = stringA.compareTo(stringA);¶

    // Übung:¶
    System.out.println(startsWith);¶
    System.out.println(endsWith);¶
    System.out.println(compare1);¶
    System.out.println(compare2);¶
    System.out.println(compare3);¶
}¶
```


Ergebnis:

"
true
true
-1
1
0
"

WICHTIGSTE METHODEN VON STRINGS (4)

```
public StringBeispiel4() {
    String string1 = "Hallo!";

    // Suchen in Zeichenketten:

    // int indexOf(String s)
    // Sucht das erste Vorkommen von s innerhalb der Zeichenkette. Wird s gefunden, wird der
    // Index des ersten übereinstimmenden Zeichens zurückgeliefert, ansonsten -1. Eine Variante
    // der Methode akzeptiert einen Parameter vom Typ char.
    int index1 = string1.indexOf("x");
    int index2 = string1.indexOf("l");
    // Fehlercode vs. explizite Ausnahme? Was ist euer Gefühl?

    // int indexOf(String s, int fromIndex)
    // Arbeitet wie die vorige Methode, beginnt allerdings mit der Suche erst bei fromIndex.
    // Auch hier akzeptiert eine Variante der Methode einen Parameter vom Typ char.
    int index3 = string1.indexOf("l", 3);

    // int lastIndexOf(String s)
    // Sucht nach dem letzten Vorkommen von s. Eine Variante der Methode akzeptiert einen
    // Parameter vom Typ char.
    int index4 = string1.lastIndexOf("l");

    // Übung:
    System.out.println(index1);
    System.out.println(index2);
    System.out.println(index3);
    System.out.println(index4);
}
```


Ergebnis:

"
-1
2
3
3
"

WICHTIGSTE METHODEN VON STRINGS (5)

```
public StringBeispiel5() {
    String string1 = "Hallo!";

    // Ersetzen von Zeichenketten:

    // String toLowerCase()
    // Wandelt die Zeichenkette in Kleinbuchstaben.
    String string2 = string1.toLowerCase();
    // String toUpperCase()
    // Wandelt die Zeichenkette in Großbuchstaben.
    String string3 = string1.toUpperCase();
    // String replace(char old, char new)
    // Einzelne Zeichen werden ersetzt: old durch new.
    String string4 = "Salat".replace("Salat", "Schnitzel");
    // String replaceAll(String regex, String new)
    // Ersetzt alle Teilstrings, die die Regular Expression regex trifft, durch den neuen
    // Teilstring new.
    String string5 = "Hallo123".replaceAll("(\\d)", "Zahl");
    // String replaceFirst(String regex, String new)
    // Ersetzt nur den ersten gefundenen Teilstring, den die Regular Expression regex trifft,
    // durch den neuen Teilstring new.
    String string6 = "Hallo123".replaceFirst("(\\d)", "Zahl");

    // Übung:
    System.out.println(string1);
    System.out.println(string2);
    System.out.println(string3);
    System.out.println(string4);
    System.out.println(string5);
    System.out.println(string6);
}
```


Ergebnis:

"
Hallo!
hallo!
HALLO!
Schnitzel
HalloZahlZahlZahl
HalloZahl23
"

KONVERTIERUNG VON STRING

- Oftmals müssen primitive Daten in Strings gewandelt werden
  - Die Klasse String liefert dazu entsprechende statische String.valueOf(..)-Methoden:

WEITERE EIGENSCHAFTEN VON STRINGS

- Die Klasse String ist final
- Strings sind Literale
- Verkettung durch + Operator
  - Beispiel: String string2 = "Hallo" + 123;
- Strings sind nicht dynamisch!
  - Inhalt und Länge Konstant
  - Jede Manipulation erzeugt ein neues Literal

1. - Eine eigene Ableitung ist nicht möglich
   - Es gibt also keine Möglichkeit, die vorhandenen Methoden auf „natürliche" Art und Weise zu ergänzen oder zu modifizieren.
   - Soll Beispielsweise die Methode „replace" etwas anderes machen, muss dies durch eine lokale Methode des Aufrufers geschehen, welche den String als Parameter bekommt.
   - Frage: Aus welchem Grund macht es Sinn, dass die Klasse String final ist?
   -> Einer der Gründe für diese Maßnahme ist die dadurch gesteigerte Effizienz beim Aufruf der Methoden von String-Objekten.
   -> Anstelle der dynamischen Methodensuche, die normalerweise erforderlich ist, kann der Compiler final-Methoden statisch kompilieren und dadurch schneller aufrufen. Daneben spielten aber auch Sicherheitsüberlegungen und Aspekte des Multithreading eine Rolle.
2. - Jedes String-Literal ist eine eigene Referenz auf ein Objekt
   - Der Compiler erzeugt für jedes Literal ein entsprechendes Objekt und verwendet es anstelle des Literals.
3. - Strings können durch den Plus-Operator verkettet werden.
   - Auch die Verkettung mit anderen Datentypen ist möglich. Es muss nur einer der Typen ein String sein.
   - Beispiel: String s = "Hallo" + 123;
4. - Strings werden wie gesagt Applikationsweit gespeichert und nur über Referenzen angesprochen.
   - D.h. bei der Initialisierung eines Strings wird der Inhalt (und somit auch die Länge) des Strings festgelegt und ist nicht mehr veränderbar.
   - Frage 1: Wie kann dann ein replace Funktionieren?
   -> Die replace Methode nimmt die Veränderung nicht auf dem Original-String vor, sondern erzeugt einen neuen String. Dieser ist mit dem gewünschten Inhalt gefüllt und gibt diese Referenz zurück.
   - Frage 2: Was passiert mit den „alten" String-Objekten?
   -> Strings sind Objekte und wenn keine Referenz mehr auf sie besteht, werden sie von der Garbage Collection weggeräumt.

STRINGBUFFER

- Arbeitet ähnlich wie String, repräsentiert jedoch dynamische Zeichenketten
- Legt den Schwerpunkt auf Methoden zur Veränderung des Inhaltes
- Das Wandeln von StringBuffer zu String ist jederzeit möglich
- Laufzeitvorteile!

```
public StringBufferBeispiel() {
    String string1 = "Hallo!";
    StringBuffer buffer = new StringBuffer(string1);
    // Konkatenierung in Schleife
    for (int i = 0; i < 20; i++) {
        buffer.append(i);          // string1 += "" + i;
    }
    // Wandlung
    String bufferString = new String(buffer);

    // Übung:
    System.out.println(bufferString);
    System.out.println(buffer);
}
```


2. - Hinzufügen, Löschen, Ersetzen einzelner Zeichen
   - Konkatenieren von Strings
4. - Werden Strings massiv verändert oder konkateniert (z.B. in langlaufenden Schleifen) ist die Verwendung von StringBuffer zu empfehlen!
   - Der Java Compiler ersetzt (laut Spezifikation) nach Möglichkeit String-Konkatenierungen durch StringBuffer!
   -> Dies erhöht die Lesbarkeit enorm und verschlechtert nicht das Laufzeitverhalten.

EXCEPTIONS

- Sinn von Exceptions
  - Strukturierte und separate (!) Behandlung von Fehlern, die während der Ausführung auftreten
  - Exceptions erweitern den Wertebereich einer Methode
- Beispiele: Array-Zugriff außerhalb der Grenze, Datei nicht gefunden, ...
- Begriffe: Exception, Throwing, Catching

2. - Ausnahmen, also Dinge, die eigentlich nicht passieren sollten, werden separat vom eigentlichen Programmcode behandelt und verarbeitet.
   - Das hat den Vorteil, dass der Quelltext wesentlich besser lesbar ist, da klar ist, was zur regulären Programmausführung und was zur Fehlerbehandlung gehört.
   -> Fehlercodes oft implizit und selbst sehr Fehleranfällig
3. - Exceptions erweitern somit den Wertebereich einer Methode.
   - Exceptions werden in die Signatur einer Methode aufgenommen. Schlüsselwort „throws XYZException“
4. - Wenn auf ein Element in einem Array zugegriffen werden möchte, dass außerhalb der Grenzen des Arrays liegt, wird das Programm nicht beendet.
   - Es fliegt eine IndexOutOfBoundsException. Diese kann separat vom regulären Programmcode behandelt werden.
5. - Exception: Die eigentliche Ausnahme, das Fehlerobjekt (!). Auch eine Exception ist ein Objekt.
   - Bsp: eine Instanz von IndexOutOfBoundsException oder FileNotFoundException.

   - Throwing: Auslösen bzw. werfen einer Exception.
   - Catching: Behandeln bzw. fangen einer zuvor geworfenen Exception.

ABLAUF BEIM VERWENDEN VON EXCEPTIONS (1)

1. Ein Laufzeitfehler oder eine Bedingung des Entwicklers löst eine Exception aus
2. Diese kann nun direkt behandelt und/oder weitergegeben werden
3. Wird die Exception weitergegeben, kann der Empfänger sie wiederum behandeln und/oder weitergeben
4. Wird die Exception gar nicht behandelt, führt sie zum Programmabbruch und Ausgabe einer Fehlermeldung

- Exceptions werden stets von „innen nach außen“ geworfen, d.h. eine Methode reicht die Exception nach außen an ihren Aufrufer weiter, usw.
- Sobald eine Exception auftritt wird der normale Programmablauf unterbrochen

AUSLÖSEN UND BEHANDELN VON EXCEPTIONS (2)

```
// Behandeln von Exceptions
public ExceptionsBeispiel() {
    // ... tue irgendwas
    try {
        tueEtwasMitEinerDatei();
    } catch (IndexOutOfBoundsException e) {
        // behandeln, weil etwas wegen einem Array kaputt ist
    } catch (FileNotFoundException e) {
        // behandeln, weil etwas wegen einer Datei kaputt ist
    } catch (Exception e) {
        // irgend etwas anderes ist kaputt gegangen.
    }
    // tue irgendwas anderes ...
}

// Auslösen von Exceptions
private void tueEtwasMitEinerDatei()
        throws IndexOutOfBoundsException,
               FileNotFoundException {
    // ...
}
```

FEHLEROBJEKTE

- Fehlerobjekte
  - Instanzen der Klasse Throwable oder ihrer Unterklasse
  - Das Fehlerobjekt enthält Informationen über die Art des aufgetretenen Fehlers
- Wichtige Methoden von Throwable
  - String getMessage(), String toString(), void printStackTrace()

```
java.lang.RuntimeException: Hier ist was dummes passiert ...
        at ba.java.weiteres.ExceptionsBeispiel.tueEtwasMitEinerDatei(ExceptionsBeispiel.java:28)
        at ba.java.weiteres.ExceptionsBeispiel.<init>(ExceptionsBeispiel.java:11)
        at ba.java.weiteres.ExceptionsBeispiel.main(ExceptionsBeispiel.java:32)
```


- Fehlerobjekte sind in Java recht ausgiebig behandelt und die Mutterklasse aller Ausnahmen ist Throwable:
Auszug aus der JavaDoc von Throwable:

```
/**
 * The {@code Throwable} class is the superclass of all errors and
 * exceptions in the Java language. Only objects that are instances of this
 * class (or one of its subclasses) are thrown by the Java Virtual Machine or
 * can be thrown by the Java {@code throw} statement. Similarly, only
 * this class or one of its subclasses can be the argument type in a
 * {@code catch} clause. ...
 */
```

- Ein Fehlerobjekt hat viele Informationen über die Art des aufgetretenen Fehlers.

4. - Wichtige Methoden von Throwable sind:
   1. getMessage(): Liefert eine für den Menschen lesbare (!) Fehlernachricht zurück.
   2. toString(): Liefert eine String Repräsentation des Fehlers zurück: Name der Exceptionklasse und Fehlernachricht.
   3. printStackTrace()
      Schreibt den Trace des aktuellen Fehlerstacks auf den Standart Error Stream (System.err). Über den StackTrace lässt sich der Ursprung des Fehlers zurückverfolgen.

FEHLERKLASSEN VON JAVA

- Alle Laufzeitfehler sind Unterklassen von Throwable
- Unterhalb von Throwable existieren zwei große Hierarchien:
  - Error ist Superklasse aller schwerwiegender Fehler
  - Exception ist Superklasse aller abnormalen Zustände
- Es können eigene Klassen von Exception abgeleitet werden

1. - Alles was von Throwable erbt, kann in einem catch Block gefangen werden.
3. - Error sollte die Applikation nicht selber fangen, sondern vom System verarbeiten lassen.
   - Auszug aus der JavaDoc von Error.

```
/**
 * An {@code Error} is a subclass of {@code Throwable}
 * that indicates serious problems that a reasonable application
 * should not try to catch.
 */
```

4. - Eine Exception ist ein abnormaler Zustand, den die Applikation selber fangen und behandeln kann (an sinnvoller Stelle).
   - Auszug aus der JavaDoc von Exception:

```
/**
 * The class {@code Exception} and its subclasses are a form of
 * {@code Throwable} that indicates conditions that a reasonable
 * application might want to catch.
 */
```

5. - Es ist zu vermeiden, einfach eine Instanz von Exception zu schmeißen
   - Damit würden sich Fehler nicht im catch Block unterscheiden lassen und das Konzept verliert an Mächtigkeit.

FANGEN VON FEHLERN...

- Es können einerseits durch verschiedene catch-Blöcke unterschiedliche Exception-Typen unterschieden werden
- Andererseits ist es üblich alle Fehler mittels der Oberklasse Exception gemeinsam zu behandeln, wenn eine Unterscheidung an dieser Stelle nicht nötig ist

1. - Sehr gute Unterscheidung der Fehlerfälle.
   - Manchmal nicht sinnvoll auf jede Fehlerart unterschiedlich zu reagieren.
2. - Beispiel GUI: Im Fehlerfall bei einer Aktion „Speichere Datei“ wird dem Benutzer nur angezeigt:
     - „Sorry, konnte nicht gespeichert werden. Versuch es bitte erneut.“
     - Obwohl es unterschiedliche Gründe haben kann.
   - Daher ist diese feingranulare Unterscheidung manchmal nicht sinnvoll.

DIE FINALLY KLAUSEL

- Nach dem try-catch kann optional eine finally-Klausel folgen
- Der Code in finally wird immer ausgeführt
- Die finally-Klausel ist damit der ideale Ort für Aufräumarbeiten

```
private void tueEtwas() throws FileNotFoundException {¶
    File file = null;¶
    try {¶
        file = new File("irgend/ein/pfad");¶
        String string = leseDatei(file);¶
        System.out.println(string);¶
    } catch (FileNotFoundException e) {¶
        // Fehlerbehandlung¶
        System.err.println(e.getMessage());¶
        throw e;¶
    } finally {¶
        schliesseDatei(file);¶
    }¶
}¶

private void schliesseDatei(File file) {¶
    // Gebe Referenz auf Datei frei¶
}¶

private String leseDatei(File file) throws FileNotFoundException{¶
    // Datei lesen ein und gebe Inhalt zurück¶
    return null;¶
}¶
```


1. - In der finally-Klausel werden üblicherweise Aufräumarbeiten getätigt.
   - Zum Beispiel: Schließen von Dateien, Freigeben von Objekten, ...
2. - Auch wenn die eigentliche Methode durch ein re-throw oder return verlassen wurde.
3. - Daher finden dort die Aufräumarbeiten statt, die unbedingt sein müssen.
   - Vorsicht: Der Zugriff auf Variablen, die im try-Block deklariert wurden ist nicht möglich.
   - Vorsicht2: Der Zugriff auf Variablen, die im try-Block instanziiert wurden, ist mit Vorsicht zu genießen! Auf null prüfen!

TRY WITH RESOURCES


```
public void tryWithResources() {
    try (InputStream fileInput =
            new FileInputStream(new File("pfad/zur/meiner/Datei"))) {
        // Hier wird die Datei eingelesen
    } catch (Exception e) {
        // Und hier der Fehler behandelt
    }
}
```


1. - Durch den try-with-resources Block werden die Ressourcen in der try(..) Klausel automatisch geschlossen.
   - Schlüsselwort ist das AutoCloseable Interface, welches implementiert werden muss.
   - In diesem Beispiel würde fileInput nach Ende des Applikationscodes (oder des Ausnahmecodes) automatisch geschlossen werden.

CATCH-OR-THROW REGEL

- Jede Exception muss behandelt oder weitergegeben werden
  - Eine Exception abzufangen und nicht weiter zu behandeln ist im Allgemeinen nicht sinnvoll und sollte begründet werden.

WEITERGABE VON EXCEPTIONS

- Soll eine Exception weitergegeben werden,
  - muss diese Exception durch throws angegeben sein
  - kann es entweder eine neu instanziierte Exception
  - oder eine Exception sein, die von „weiter unten kommt“, welche explizit behandelt wurde und mittels throw weitergeworfen wurde
  - oder eine Exception sein, die von „weiter unten kommt“, welche nicht von explizit im catch Block behandelt wurde

3. - Es passiert irgendwas im Programmablauf, was dazu führt, dass irgendwo im Code „throw new XYZException()“ steht.
4. - Eine Exception wird im catch Block gefangen, eine rudimentäre Behandlung durchgeführt (zum Beispiel geloggt) und dann die gleiche Exceptions mittels „throw existierendeException;“ weitergeworfen werden.
5. - Eine Exception wird nicht im catch Block gefangen. Dann wird diese automatisch weitergeworfen, ohne dass der folgende Code etwas davon mitbekommt.
   - Doch wie kann das sein? Exceptions müssen doch immer angegeben werden?
   -> Die RuntimeException macht es möglich! Siehe nächste Folie.

RUNTIMEEXCEPTION

- Unterklasse von Exception
- Superklasse für alle Fehler, die nicht behandelt werden müssen
  - Entwickler kann entscheiden ob er diese Exception fängt
- Ausnahme von der catch-or-throw Regel
- Muss nicht in throws deklariert werden
- Eingeständnis zum Aufwand, aber gefährlich!

2. - Eine RuntimeException (und alle Klasse, die davon erben) können behandelt werden in einem catch-Block.
   - Sie müssen aber nicht behandelt werden!
3. - Einem Entwickler ist also die Freiheit gegeben, ob er eine RuntimeException fängt, oder eben nicht.
   - Es kann eine Methode, die eine RuntimeException wirft, benutzt werden, ohne dass ein catch-Block von Nöten ist.
4. - Genau aus diesem Grund ist die RuntimeException auch eine Ausnahme von der catch-or-throw Regel.
   - Wiederholung: Was besagt die catch-or-throw Regel?
   - Warum ist es also eine Ausnahme von dieser Regel?
5. - Zudem kommt der unglaublich ungünstige Umstand, dass RuntimeExceptions nicht in die Methodensignatur aufgenommen werden müssen.
   - Warum ist dies problematisch?
6. - Die RuntimeException war ein Eingeständnis an die Java Entwickler, damit der Aufwand für Fehlerbehandlung nicht zu groß wird.
   - In meinen Augen eines der gefährlichsten Features von Java. (Seit JDK1.0)
   - Die Applikation kann „auseinander brechen“, ohne dass es dem verwendeten Entwickler überhaupt klar ist was passieren kann.
   -> In meiner Erfahrung hat die RuntimeException zu sehr großen Systemabstürzen geführt!
   -> Einfach aus dem Grund der Unwissenheit!

ENUMERATION

- In der Praxis treten Datentypen mit kleinen und konstanten Wertemengen relativ häufig auf
- Verwendung wie andere Typen
  - z.B. Anlegen von Variablen des Typs
  - Variablen können nur vorgegebene Werte annehmen

```
// Lokale Enumeration
enum Farbe {
    ROT, GRUEN, BLAU
}

public EnumBeispiel() {
    // Deklaration
    Farbe farbe;
    // Initialisierung
    farbe = Farbe.ROT;
}
```


ENUMS SIND KLASSEN

- Werte von Enums sind Objekte und werden auch so genutzt
- Alle Werte von enums sind singletons
- Enums selbst besitzen weitere Eigenschaften
  - values(), valueOf(String), toString(), equals(..)
  - Werte sind direkt in switch Anweisung verwendbar
- Es gibt spezielle Collections für Enums: EnumSet, EnumMap

1. - Vergleiche Beispiel von eben.
   - Enums werden deklariert und zugewiesen.
   - Namenskonvention: - Schreibweise von Enums an sich wie bei Klassen (Großbuchstabe + CamelCase)
     - Schreibweise von Werten in Großbuchstaben und Unsterstrich
2. - Allerdings sind alle Werte von Enums singletons.
   - D.h. alle Werte existieren genau ein mal pro Applikation.
   - Wiederholung: Was hat das also zur Folge? Wie oft ist eine Farbe ROT also in einer Applikation vorhanden?
4. - values() liefert einen Array von allen Werten die für das Enum definiert sind.
   - valueOf(„ROT“) liefert den Enum Wert, welcher zur Farbe ROT gehört (Schreibweise wichtig!).
     -> Wird kein Wert zu dem angegebenen String gefunden, wird eine IllegalArgumentException geworfen (=RuntimeException).
   -> die Klasse enum implementiert damit das Interface: Compareable und Serializable.
5. - Werte sind direkt in einer switch Anweisung verwendbar.
   -> War bis Java7 ein echtes Feature, mittlerweile sind aber auch Strings in einer switch Anweisung verwendbar.
   - Von daher eher für alte Java Versionen interessant.
6. - Warum würde eine normale Liste wenig Sinn machen bei Enums (also Aufzählungen)?
   -> Sie sind singletons, eine Liste mit 100 Elementen die alle auf das eine Objekt „Rot“ zeigen macht wenig sinn..

Hinweis: Die Interfaces Compareable, Serializable und Collections werden wir später besprechen!

ENUMS KÖNNEN ERWEITERT WERDEN

- Enums sind Klassen ..
  - .. und lassen sich auch so definieren

```
public enum Farbe {
    ROT(255, 0, 0), GRUEN(0, 255, 0), BLAU(0, 0, 255);

    private int r, g, b;

    private Farbe(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public String toRGB() {
        return "r: " + r + ", g: " + g + ", b: " + b;
    }
}
```


1. - Modifier des Konstruktors ist auf private limitiert.
   - Frage Warum?
   -> Wenn er public wäre, könnten sich externe Klassen ja diesem Konstruktor bedienen und sich eigene Rot Objekte machen.
   - Modifier von Werten des Enums ist implizit public.

BEDARF NACH GENERICS

- bis Java 1.4
  - Wenn man einen Typ nicht explizit einschränken wollte, musste man auf Object arbeiten
  - Dies ist nicht typsicher -> Rückgabewert Object
  - Daher wurde sehr viel gecastet
  - Vor allem bei Collections ein prinzipielles Problem

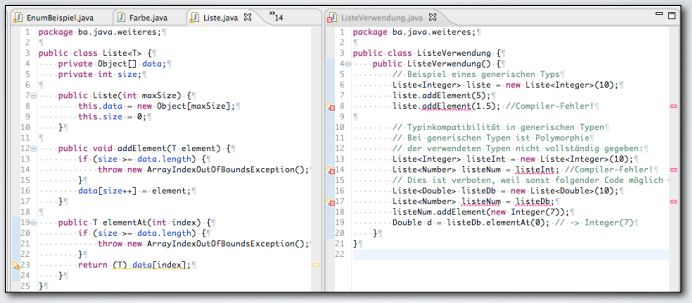
1. - Ohne Generics konnten Klassen, die andere Typen verwenden diese aber nicht explizit einschränkten wollte, lediglich auf Object arbeiten
2. - Dies ist nicht typsicher, das heißt: Die Klasse kann in ihren Methoden zwar alle möglichen Objekte aufnehmen, bei der Rückgabe von gespeicherten Objekten kann der Typ der Objekte allerdings nicht mehr unterschieden werden.
   - Beispiel: Ich möchte eine Liste wo nur „ähnliche Objekte“ drin sind. Also zum Beispiel nur Autos oder Unterklassen.
   -> Nun ist es Schwachsinn für Jede Klasse eine eigene Listenimplementierung zu machen, genau so wie es Schwachsinn ist in diesem Beispiel Autos zusammen mit Kuchen zu speichern (was bei Object allerdings möglich wäre).
3. - Da nur auf Object gearbeitet wurde, musste viel von Object z.B. nach Auto und von Auto nach Object gecastet werden.
4. - Wie bereits in den Beispielen angesprochen ist dies vor allem bei Collections ein sehr großes Problem.
   - Eine Collection ist im Prinzip wie ein Array, nur wesentlich komfortabler.

SINN VON GENERICS

- Ab Java 5
  - Mittels Generics können typsichere Klassen unabhängig vom konkreten verwendeten Typ definiert werden
  - Die Klasse wird so implementiert, dass sie mit jedem Typ zusammenarbeiten kann (Kann eingeschränkt werden)
  - Mit welchem Typ sie zusammen arbeiten muss, wird bei Initialisierung festgelegt
- Generics ersetzen die Arbeit mit Object dennoch nicht (ganz)

3. - Mit welcher konkreten Klasse eine Klasse zusammen arbeiten muss, kann bei der Initialisierung festgelegt und eingeschränkt werden.
5. - Beim Instanziieren einer generischen Klasse muss der konkrete Typ, mit dem gearbeitet werden soll, angegeben werden.
   - Danach kann diese Instanz nur mit diesem Typ arbeiten.
   - Es gibt jedoch immer wieder Fälle, in denen der konkrete Typ irrelevant ist.
   -> In diesen Fällen ist es nach wie vor gut, allgemein auf Object zu arbeiten.
   Beispiel: List<Object> meinEigentum;

BEISPIEL



Anmerkung:
Man kann in Java keinen neuen generischen Array anlegen. Daher wird hier auf Objekt gearbeitet.

WILDCARD ? ALS TYPPARAMETER

- „?“ Kann anstelle eines konkreten Typs angegeben werden, wenn der Typ selbst keine Rolle spielt
- Damit geht die Typsicherheit verloren, allerdings explizit
- Nur bei lesendem Zugriff erlaubt. Kein new Liste<?> möglich!

```
public void printAll(List<?> l) {
    for (Object o : l) {
        System.out.println(o);
    }
}
```


GEBUNDENE WILDCARDS

- Abgeschwächte Form der ? Wildcard durch Einschränkung auf Subklassen einer gegebenen Superklasse mittels extends
- Es ist auch eine Einschränkung nach oben durch super möglich
- Auch die gebundene Wildcard ist nur lesend erlaubt

```
// Vorsicht: Das ist nicht unsere Klasse Liste!
public void printAllNumber(List<? extends Number> list) {
    List<? extends Number> liste2; // Auch dies ist lesend!
    for(Number numb : list) {
        System.out.println(numb.doubleValue());
    }
}
```


2. - Aber äußerst selten!

PROJECT COIN


```
public void projectCoin() {
    List<Integer> integerList = new LinkedList<Integer>();
    Map<Integer, List<Integer>> integerZuIntegerListMap =
            new HashMap<Integer, List<Integer>>();
    Map<Integer, Map<Integer, List<Integer>>> blabla =
            new HashMap<Integer, Map<Integer, List<Integer>>>();
    // Das ist zu viel! Daher ProjectCoin ab Java7
    // Der Compiler weiß schon am Besten, was ich initialisieren will,
    // kümmert er sich auch um die Generic Handling
    List<Integer> integerList7 = new LinkedList<>();
    Map<Integer, List<Integer>> integerZuIntegerListMap7 = new HashMap<>();
    Map<Integer, Map<Integer, List<Integer>>> blabla7 = new HashMap<>();
}
```


126