6. Grundkonzepte der Objektorientierung (4):

Generische Klassen

Java-Beispiele:

StackForChar.java
BuildPairs.java
BuildPairsBounds.java
StackGen.java

Schwerpunkte

- Das Generalisierungsproblem
- Generalisierung durch Klasse ‚Object‘
- Wrapper-Klassen
- cast-Operator für Objekte
- Generische Klassen
- Generischer Typ

Das Generalisierungsproblem

Problem: Stack für unterschiedliche Elementtypen

- Für char-Elemente (bisherige Variante)

Stack.java

```
class Stack {
   private char [] stackElements;
   private int top;
      ...
   public void push(char x) {
      top++; stackElements[top] = x;
   }
   ...
```

austauschen

- Für int-Elemente
- Für Zeitangaben: Klasse 'Time'

→ 3 Klassen mit größtenteils identischem Code

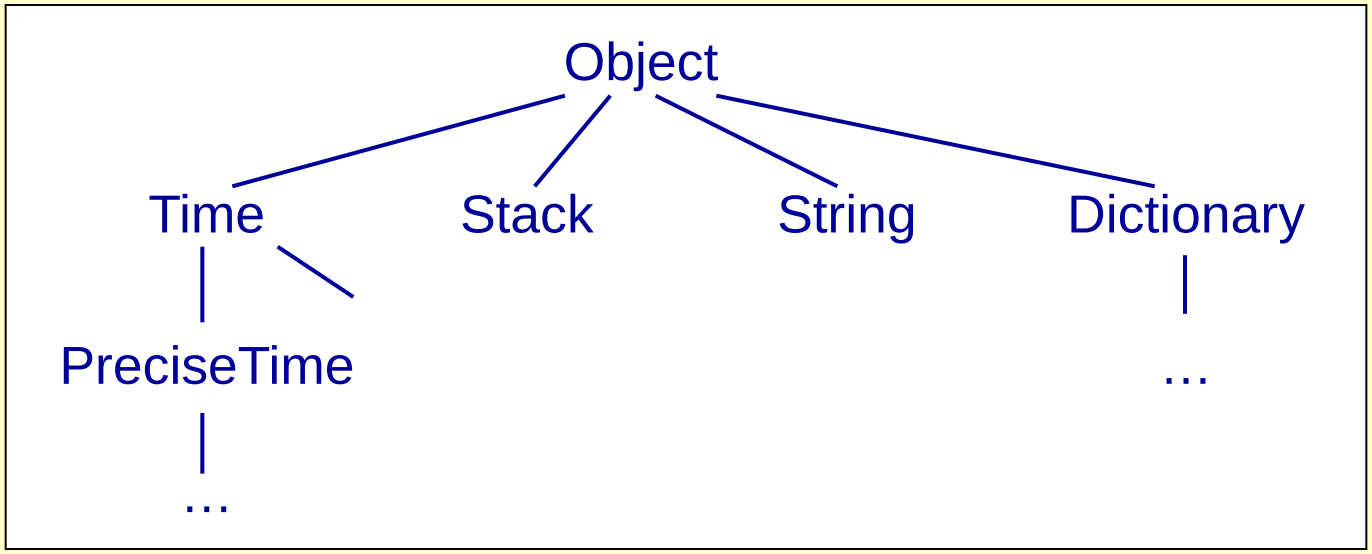
Gibt es eine elegantere Lösung?

Allgemeiner Stack: Elementtyp 'Object'

StackForChar.java

```
class Stack {
   private Object [] stackElements;
   private int top;

   public Stack(int n) {
      stackElements = new Object[n];
      top = -1;
   }

   public void push(Object x) {
      top++;
      stackElements[top] = x;
   }
   ...
}
```

Wiederholung: Kompatibilitätsregel der Klassenhierarchie



Objekte einer Unterklasse dürfen dort stehen, wo Objekte von Oberklassen zugelassen sind.

Beispiele:

```
Time t;
t = new PreciseTime(12, 10, 1);
t.printTime();
```

```
Object o;
o = new Time(12, 10);
o.printTime();
```

Allgemeiner Stack: Anwendung auf 'Time'

```
class Stack {
    private Object [] stackElements;
    private int top;

    public Stack(int n) {
        stackElements = new Object[n];
        top = -1;
    }

    public void push(Object x) {
        top++; stackElements[top] = x;
    }
    ...
}
```

Stack mit Basistyp Object

```
Time t1 = new Time(8, 30);
Stack timeStack = new Stack(10);

timeStack.push(t1.copy());
t1.addMinutes(10);
timeStack.push(t1.copy());
t1.addMinutes(30);
timeStack.push(t1.copy());
```

Statt Object wird Time übergeben

Was passiert bei `timeStack.push(t1)`?

indirekt:
`class Time extends Object`

Wrapper-Klassen

Allgemeiner Stack: Anwendung auf 'char'

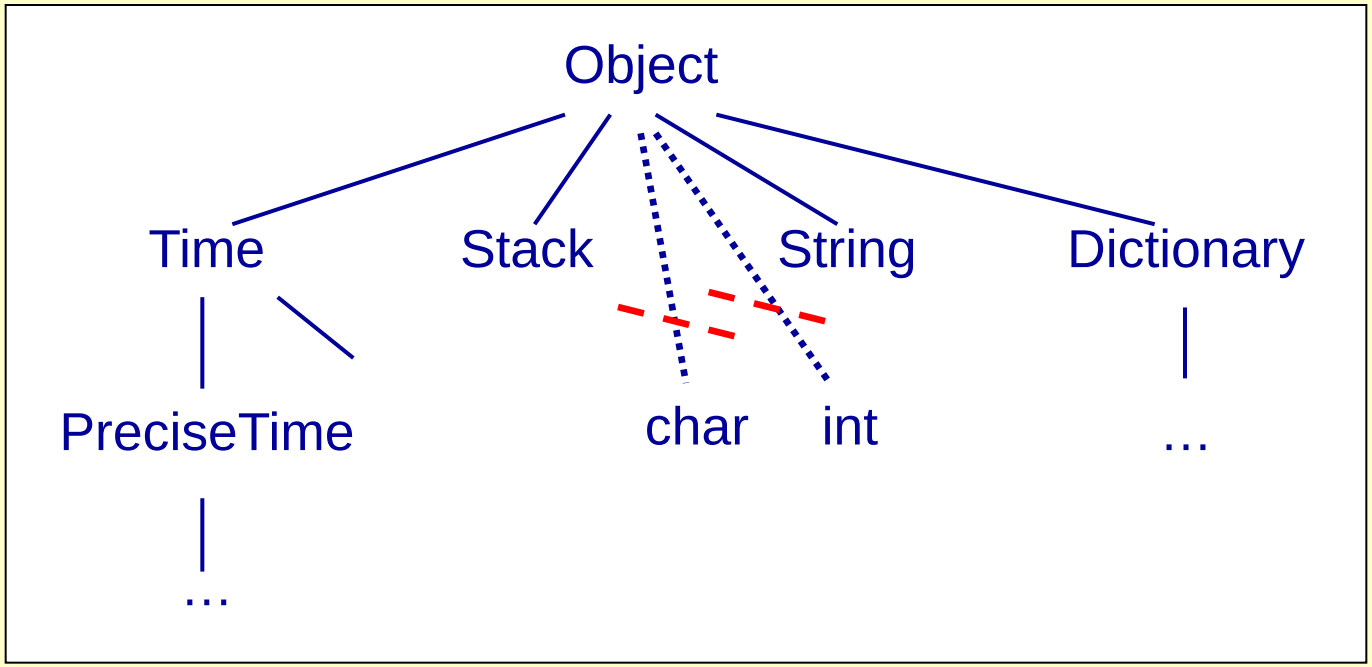
```
class Stack {
     private Object [] stackElements;
     private int top;

     public Stack(int n) {
          stackElements = new Object[n];
          top = -1;
     }

     public void push(Object x) {
          top++; stackElements[top] = x;
     }
     ...
}
```

```
char ch = 'A';

Stack s = new Stack(10);
s.push(ch);
```

erwartet Klasse Object o. Unterklasse

```
Stackchar.java:52: Incompatible type for method.
Can't convert char to java.lang.Object.
```

Klassenhierarchie



Objekte einer Unterklasse dürfen dort stehen, wo Objekte von Oberklassen zugelassen sind:

Aber elementare Typen sind keine Klassen.

Wrapper-Klassen

einfacher Typ → Klasse

einfacher Typ 'verpackt' in Klasse

wrap = verpacken

`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`double`	`Double`
`float`	`Float`
`boolean`	`Boolean`
`char`	`Character`
`void`	`Void`

Java-API: in java.lang

Jeweils Unterklasse von java.lang.Object
→ Können dort stehen, wo Object erlaubt ist.

Wrapper-Klasse ´Character´

java.lang

Class Character

java.lang.Object
  java.lang.Character

All Implemented Interfaces:

Comparable, Serializable

---

public final class **Character**
extends Object
implements Serializable, Comparable

The `Character` class wraps a value of the primitive type `char` in an object. An object of type `Character` contains a single field whose type is `char`.

In addition, this class provides several methods for determini

Character information is based on the Unicode Standard

The methods and data of class `Character` are defi ecifies various properties including name and general categ

The file and its description are available from the U

- http://www.unicode.org

Since:

1.0

See Also:

Serialized Form

Field Summary

static byte	COMBINING_SPACING_MARK	
static byte	CONNECTOR_PUNCTUATION	
static byte	CONTROL	General catego
static byte	CURRENCY_SYMBOL	Ge
static byte	DASH_PUNCTUATION	G
static byte	DECIMAL_DIGIT_NUMBER	
static byte	DIRECTIONALITY_ARABIC_	
static byte	DIRECTIONALITY_BOUNDARY	
static byte	DIRECTIONALITY_COMMON_NUM	
static byte	DIRECTIONALITY_EUROPEAN_NUMBER	Weak bidirectional character type "EN" in the Unicode specification.
static byte	DIRECTIONALITY_EUROPEAN_NUMBER_SEPARATOR	Weak bidirectional character type "ES" in the Unicode specification.
static byte	DIRECTIONALITY_EUROPEAN_NUMBER_TERMINATOR	Weak bidirectional character type "ET" in the Unicode specification.
	...	

The Character class wraps a value of the primitive type char in an object. An object of type Character contains a single field whose type is char.

In addition, this class provides several methods for determining a characters's category (lowercase letter, digit, etc.) and for converting characters from uppercase to lowercase and vice versa.

Constructor Summary

`Character(char value)`	Constructs a newly allocated Character

Method Summary

char	`charValue()` Returns the value of this Character object.
int	`compareTo(Character ...`
int	`compareTo(Object o)` Compares this
static int	`digit(char ch, int radix)` Returns the nu
boolean	`equals(Object obj)` Compares this object aga
static char	`forDigit(int digit, int radix)` Determines the character representation for a specific digit in the specified radix.
static byte	`getDirectionality(char ch)` Returns the Unicode directionality property for the given character
static int	`getNumericValue(cha...`
static int	`getType(char ch)` Re
int	`hashCode()` Returns a ha
static boolean	`isDefined(char ch)`
static boolean	`isDigit(char ch)` Determines if the specified character is a digit.
static boolean	`isIdentifierIgnorable(char ch)`
static boolean	`isISOControl(char ch)` Determi
static boolean	`isJavaIdentifierPart(char ch)`
static boolean	`isJavaIdentifierStart(char ch)`
static boolean	`isJavaLetter(char ch)` **Deprecated.** *Replaced by isJavaIdentifierStart(char).*
static boolean	`isJavaLetterOrDigit(char ch...`
static boolean	`isLetter(char ch)` Determi
static boolean	`isLetterOrDigit(char ch)`
static boolean	`isLowerCase(char ch)` Determines if the specified character is a lowercase character.
static boolean	`isMirrored(char ch)`
static boolean	`isSpace(char ch)` D
static boolean	`isSpaceChar(char ch)`
static boolean	`isTitleCase(char ch)`
	...

public Character(char value)

Constructs a newly allocated Character object that represents the specified char value.

public char charValue()

Returns the value of this Character object.

public static int getNumericValue(char ch)

Returns the int value that the specified Unicode character represents.

public static boolean isDigit(char ch)

Determines if the specified character is a digit.

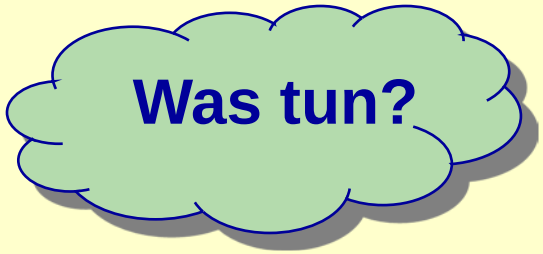
public static boolean isLetter(char ch)

Determines if the specified character is a letter.

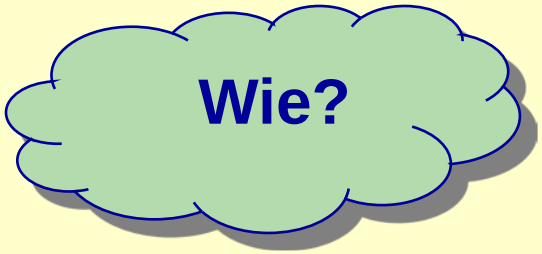
public static boolean isLowerCase(char ch)

Determines if the specified character is a lowercase character.

Annahme:
Java-API ohne Wrapper-Klassen



➔ selbst implementieren

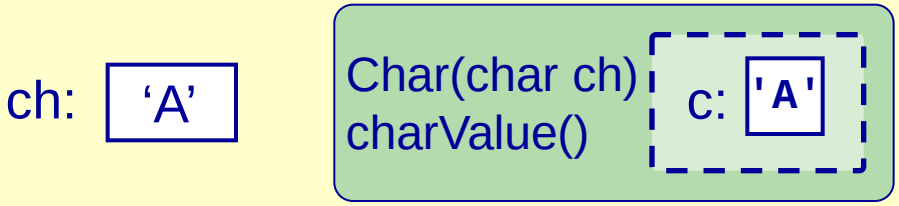


Wrapper 'Char' für 'char': selbst implementiert

```
class Char {
   private char c;

   public Char(char ch) {
      c = ch;
   }
   public char charValue() {
      return c;
   }
}
```

StackForChar.java

```
Char ch;

ch = new Char('A');
char c = ch.charValue();
```



Allgemeiner Stack: Anwendung auf Wrapper-Klasse 'Character'

```
class Stack {
     private Object [] stackElements;
     private int top;

     public Stack(int n) {
          stackElements = new Object[n];
          top = -1;
     }

     public void push(Object x) {
          top++; stackElements[top] = x;
     }

     public Object top() {…}
     ...
}
```

```
Character ch;
Stack s = new Stack(10);

ch = new Character(Keyboard.readChar());
s.push(ch);
ch = (Character)s.top();
char c = ch.charValue();
```

Typ 'char'

nach 'Character'

Von ‚Character' nach 'char'

Kompatibilität und Typ-Cast

```
class Stack {
    private Object [] stackElements;
    private int top;

    public Stack(int n) {
        stackElements = new Object[n];
        top = -1;
    }

    public void push(Object x) {
        top++; stackElements[top] = x;
    }

    public Object top() {…}
    ...
}
```

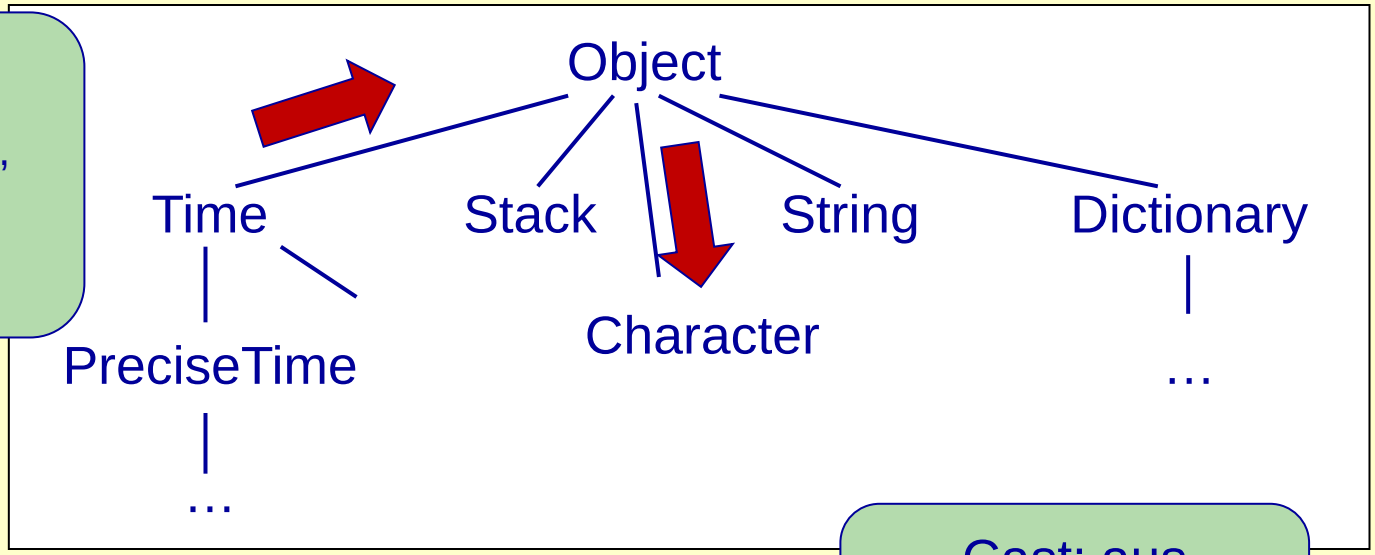
```
Character ch;
Stack s = new Stack(1

ch = new Character(Keyboard.readChar());
s.push(ch);
ch = (Character)s.top();
char c = ch.charValue();
```

Kompatibilitätsregel:
Objekte einer Unterklasse dürfen dort stehen, wo Objekte von Oberklassen zugelassen sind.

Umgekehrte Richtung (Oberklasse → Unterklasse):
Typ-Cast von 'Object' nach 'Character' notwendig

Kompatibilität: automatisch – mit cast-Opertator

Objekte einer Unterklasse dürfen dort stehen, wo Objekte von Oberklassen zugelassen sind.



Cast: aus allgemeinem Typ wird spezieller

Standard: spezieller Typ wird allgemeinem Typ zugeordnet

```
Time t;
t = new PreciseTime(12, 10, 1);
t.printTime();
```

```
Character ch;
Stack s = new Stack(10);

ch = (Character)s.top();
char c = ch.charValue();
```

Wrapper-Klasse ´Character'

java.lang

Class Character

java.lang.Object
  java.lang.Character

All Implemented Interfaces:
Comparable, Serializable

public final class **Character**
extends Object
implements Serializable, Comparable
The `Character` class wraps a value of the primitiv

In addition, this class provides several methods for d

Character information is based on the Unicode Stand

The methods and data of class `Character` are defi
various properties including name and general categ

The file and its description are available from the Ur

- http://www.unicode.org

Since:
1.0

See Also:
Serialized Form

- The Character class wraps a value of the primitive type char in an object. An object of type Character contains a single field whose type is char.
- In addition, this class provides several methods for determining a characters's category (lowercase letter, digit, etc.) and for converting characters from uppercase to lowercase and vice versa.

Field Summary

static byte	**COMBINING_SPACING_MARK** General category "Mc" in the Unicode specification.
static byte	**CONNECTOR_PUNCTUATION** General category "Pc" in the Unicode specification.
static byte	**CONTROL** General category "Cc" in the Unicode specification.
static byte	**CURRENCY_SYMBOL** General category "Sc" in the Unicode specification.
static byte	**DASH_PUNCTUATION** General category "Pd" in the Unicode sp
static byte	**DECIMAL_DIGIT_NUMBER** General category "Nd" in the Un
static byte	**DIRECTIONALITY_ARABIC_NUMBER** Weak bidire
static byte	**DIRECTIONALITY_BOUNDARY_NEUTRAL** Weak bidi
static byte	**DIRECTIONALITY_COMMON_NUMBER_SEPARATOR** W
static byte	**DIRECTIONALITY_EUROPEAN_NUMBER** Weak bidirectional charact
static byte	**DIRECTIONALITY_EUROPEAN_NUMBER_SEPARATOR** Weak bidirectional character type "ES" in the Unicode specification.
static byte	**DIRECTIONALITY_EUROPEAN_NUMBER_TERMINATOR** Weak bidirectional character type "ET" in the Unicode specification.
	…

Logische Struktur in Ordnung?

Constructor Summary

`Character(char value)`	Constructs a newly allocated Character

Method Summary

char	`charValue()` Returns the value of this Character object.
	… o) Compares this …
	…int radix) Returns the nu…
	…bj) Compares this object aga…
	…git, int radix) Determines the character representation for a specific digit in the specified radix.
static byte	`getDirectionality(char ch)` Returns the Unicode directionality property for the given character.
static int	`getNumericValue(char ch)` Returns the int value that the specified Unicode character represents.
	…h) Determines if the specified character is a digit.
	…orable(char ch)
	…ar ch) Determi…
	…rPart(char ch)
	…rStart(char ch)
	…ar ch) **Deprecated.** *Replaced by isJavaIdentifierStart(char).*
	…orDigit(char ch)
	…char ch) Determines if the specified character is a lowercase character.
	…ch)
static boolean	`isTitleCase(char ch)`
	…

public Character(char value)
Constructs a newly allocated Character object that represents the specified char value.

Wrapper

public char charValue()
Returns the value of this Character object.

Service: Arbeit mit Zeichen

public static int getNumericValue(char ch)
Returns the int value that the specified Unicode character represents.

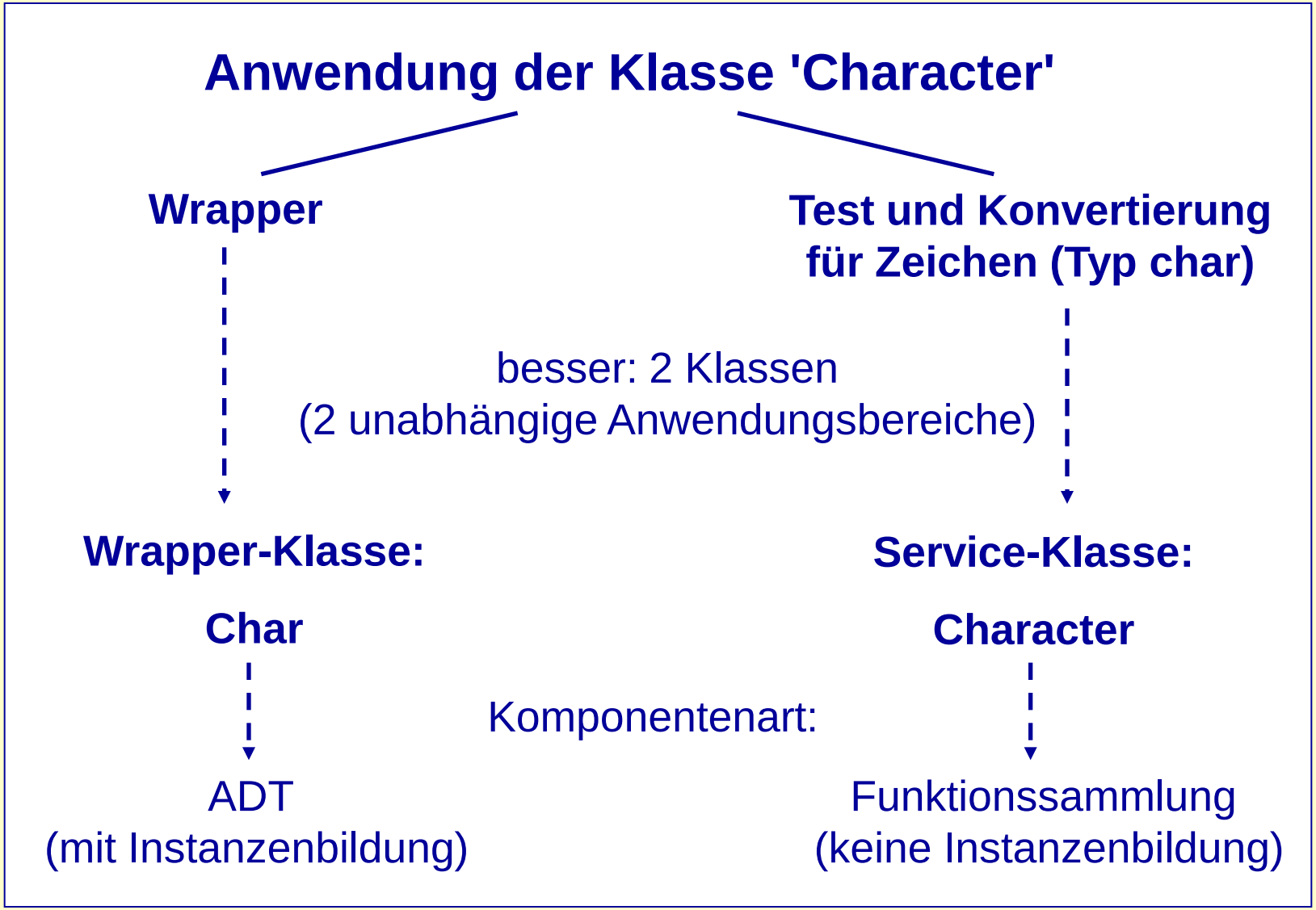
public static boolean isDigit(char ch)
Determines if the specified character is a digit.

public static boolean isLetter(char ch)
Determines if the specified character is a letter.

public static boolean isLowerCase(char ch)
Determines if the specified character is a lowercase character.

Die Klasse ‘Charakter‘ spielt keine Rolle.

Java-API: logische Struktur ok?



Anwendungen von 'Character'

Anwendung der API-Klasse 'Character'

Wrapper

Instanzenbildung

```
ch = new Char('A');

char c = ch.charValue();
```

Instanzmethoden

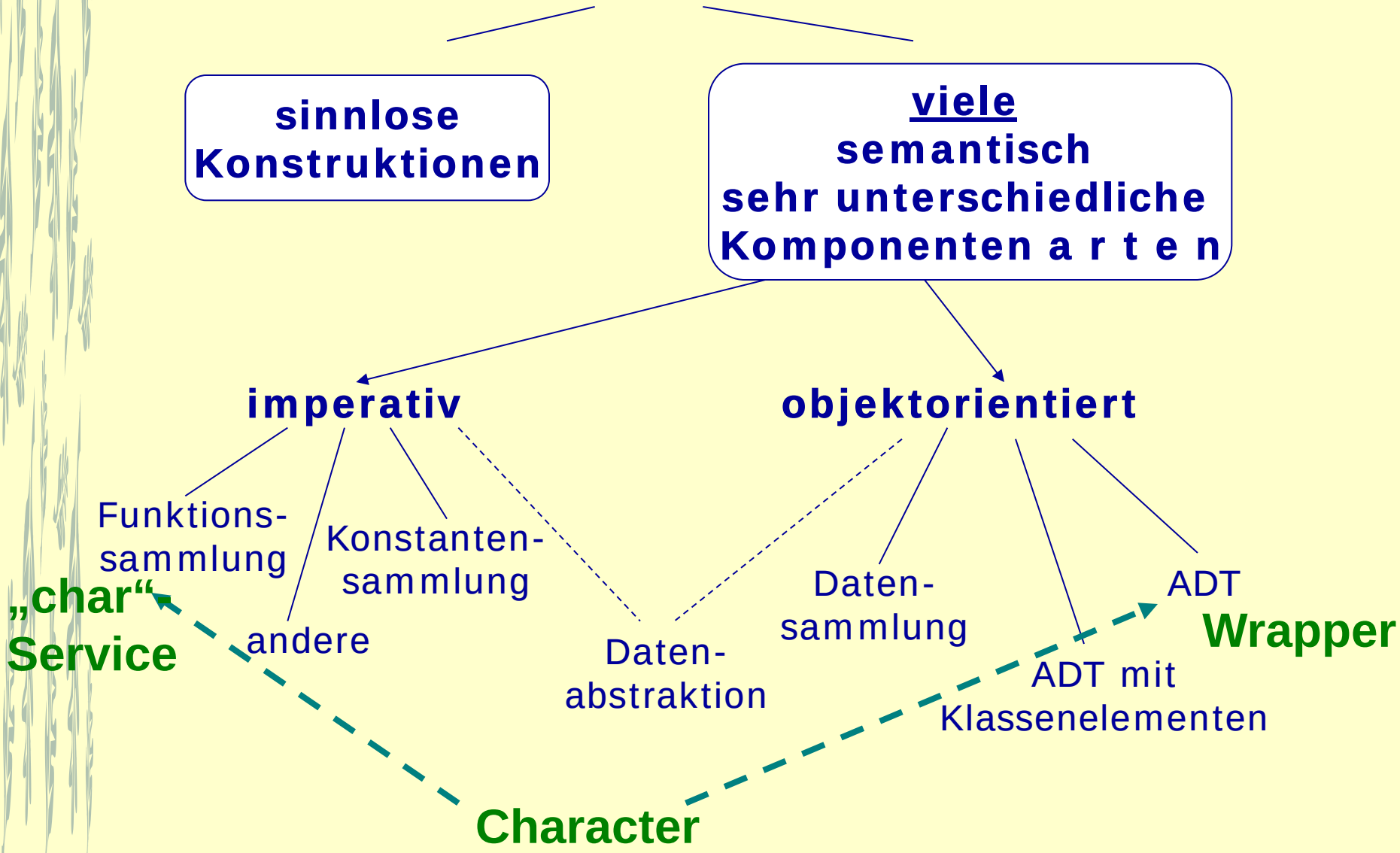
Test und Konvertierung für Zeichen (Typ char)

keine Instanzenbildung

```
Character.isLetter('A');

Character.isDigit(c);
```

Klassenmethoden

Platz der API-Klasse ´Character‘



Generische Klassen

Immer "reine" Stacks: derselbe Basistyp garantiert (z.B. Character)?

Allgemeiner Stack: Elementtyp 'Object'

StackForChar.java

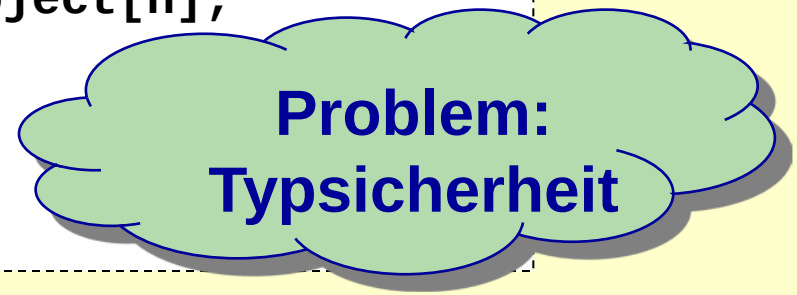
```
class Stack {
   private Object [] stackElements;
   private int top;

   public Stack(int n) {
      stackElements = new Object[n];
      top = -1;
   }
   ...
}
```

bisher: Stack von Time, Stack von Character, …

Gemischte Stacks möglich?

"Gemischte" Stacks möglich

```
class Stack {
   private Object [] stackElements;
   private int top;

   public Stack(int n) {
      stackElements = new Object[n];
      top = -1;
   }
   ...
}
```




```
Stack mixedStack = new Stack(10);

mixedStack.push(new Char('A'));
mixedStack.push(new Time(8,30));
mixedStack.push(new Integer(20));
```

Oft will man Stacks mit demselben Basistyp, kann es aber so nicht garantieren.

Schön wäre es, wenn es Typ-Parameter gäbe … (1)

Anstelle von:

```
class Stack {
    private Object [] stackElements;
    private int top;

    public Stack(int n) {
        stackElements = new  Object [n];
        top = -1;
    }

    public void push(Object x) {
        top++; stackElements[top] = x;
    }

    public Object top() {
        . . .
    }
  . . .
}
```

Schön wäre es, wenn es Typ-Parameter gäbe … (2)

nun:

```
class Stack<T> {
    private T [] stackElements;
    private int top;

    public Stack(int n) {
        stackElements = new  T [n];
        top = -1;
    }

    public void push(T x) {
        top++; stackElements[top] = x;
    }

    public T top() {
       . . .
    }
  . . .
}
```

T steht für **beliebigen**, aber **festen** Typ

… mit einer Aktualisierung durch Typargumente

```
Stack<Time> tStack;
Stack<Character> chStack;

tStack = new Stack<Time> ();
chStack = new Stack<Character> ();

tStack.push(new Time(1,20));
chStack.push(new Character('A'));
chStack.push(new Time(7,10));
```

Typfehler: Compiler erwartet Parameter vom Typ ‘Character’

Damit: keine „gemischten“ Stacks erlaubt

Anmerkung:
Dieses Beispiel klappt so nicht ganz.

Generische Klasse: Deklaration

```
class Pair<T> {
    private T first;
    private T second;

    public Pair(T fst, T scd) {
       first = fst;
       second = scd;
    }

    public T getFirst() {
      return first;
    }

    public T getSecond() {
      return second;
    }
  }
```

Typvariable

Generische Klasse

BuildPairs.java

T: beliebiger, aber fester Typ

Damit keine gemischten Tupel: z. B. (Integer, String)

Generische Klassen: Ab Java 2 (Version 1.5)

Generischer Typ: Typvariablen werden aktualisiert

BuildPairs.java

```
class BuildPairs {

  public static void main (String[] args) {
    Pair<Integer> pi;
    Pair<String>  ps;
    Integer i, j;

    i = new Integer(99);
    j = new Integer(100);
    pi = new Pair<Integer> (i, j);
    ps = new Pair<String> ("Hallo", "World");

    System.out.println(ps.getFirst() + " "
           + ps.getSecond());
    System.out.println(pi.getFirst().intValue()
           + " " + pi.getSecond().intValue());
  }
}
```

Generischer Typ

Typargument

Generischer Typ = Generische Klasse + Typargument

Generischer Typ: Verwendung wie normaler Typ

BuildPairs.java

```
class BuildPairs {

  public static void main (String[] args) {
    Pair<Integer> pi;
    Pair<String>  ps;
    Integer i, j;

    i = new Integer(99);
    j = new Integer(100);
    pi = new Pair<Integer> (i, j);
    ps = new Pair<String> ("Hello", "World");

    System.out.println(ps.getFirst() + " "
           + ps.getSecond());
    System.out.println(pi.getFirst().intValue()
           + " " + pi.getSecond().intValue());
  }
}
```

Variablen vereinbaren

Konstruktor aufrufen

Methoden aufrufen

```
% java BuildPairs

Hello World
99 100
```

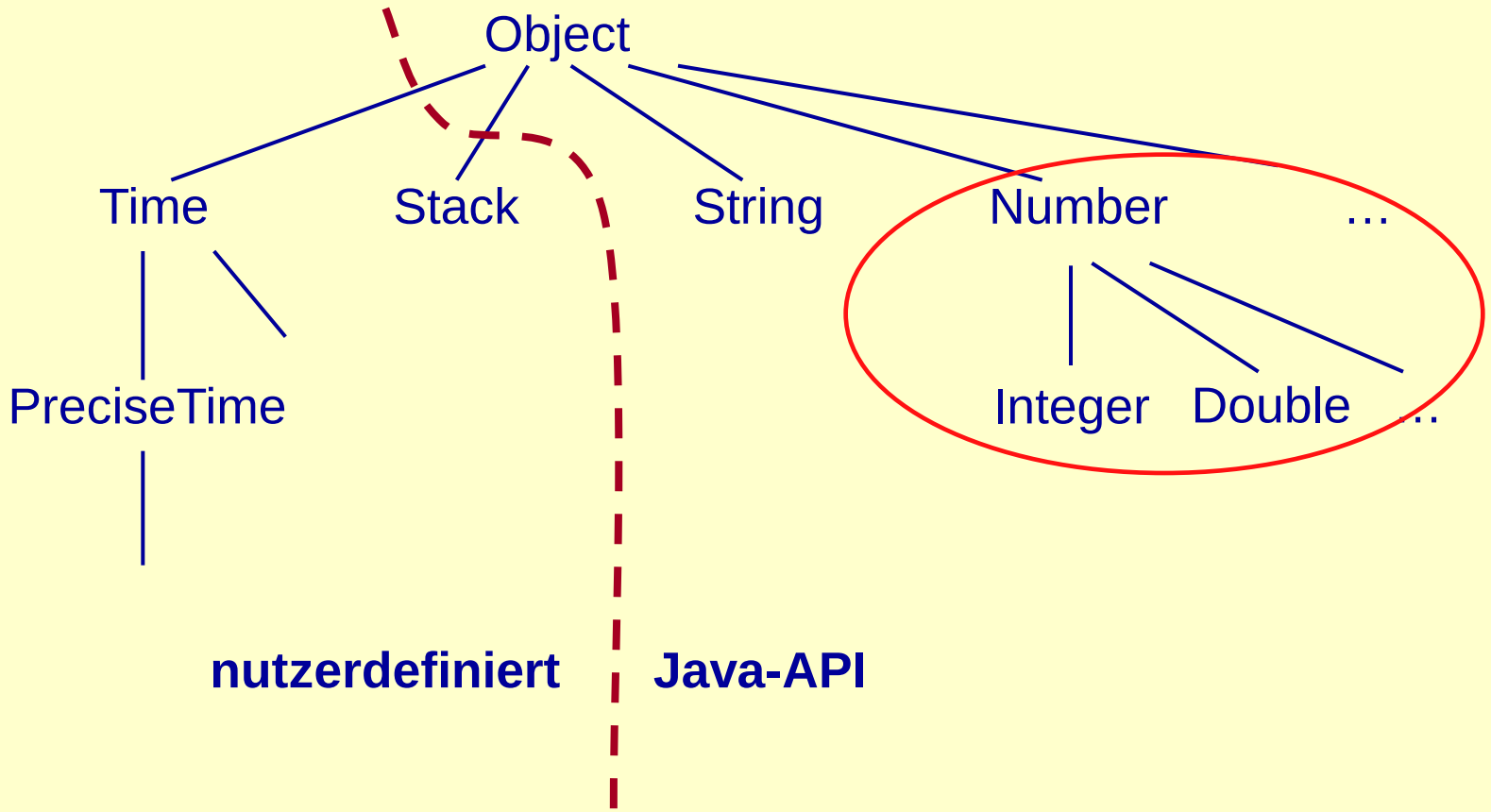
Typbounds: Einschränkung von Typargumenten

BuildPairsBounds.java

```
class PairNumber <T extends Number> {
  private T first;
  private T second;

  PairNumber(T fst, T scd) {
     first = fst;
     second = scd;
  }

  public T getFirst() {
    return first;
  }

  public T getSecond() {
    return second;
  }

  public double add () {
    return first.doubleValue() + second.doubleValue();
  }
}
```

Typargument muss von Number abgeleitet sein: Integer, Double … (Wrapper-Klassen, die Zahlen repräsentieren)

Klasse ‚Number': Oberklasse von „Zahlen"-Wrapperklassen



Typbounds: Einschränkung von Typargumenten

BuildPairsBounds.java

```
class BuildPairsBounds {

  public static void main (String[] args) {
    PairNumber<Integer> pi;
//  PairNumber<String>  ps;
    Integer i, j;

    i = new Integer(99);
    j = new Integer(100);
    pi = new PairNumber<Integer> (i, j);

    System.out.println(pi.getFirst().intValue()
            + " " + pi.getSecond().intValue() + " "
            + pi.add());
  }
}
```

Integer von Number abgeleitet

Fehler!

```
% java BuildPairsBounds

99 100 199.0
```

Erinnerung: Schön wäre es, wenn es Typ-Parameter gäbe …

```
class Stack  <T> {
    private T [] stackElements;
    private int top;

    public Stack(int n) {
        stackElements = new  T [n];
        top = -1;
    }

    public void push(T x) {
        top++; stackElements[top] = x;
    }

    public T top() {
      . . .
    }
 . . .
}
```

Nicht erlaubt: Typvariable als Array-Elementtyp

Ausweg: Statt Arrays verwende man generische API-Klasse ArrayList

ArrayList

StackGen.java

Kombination aus Array und Liste

```
ArrayList<T> stackElements;
```

```
statt: T[] stackElements;
```

- Als Array: Zugriff über Index

  ```
  stackElemements.get(top)
  ```

- Als Liste: beliebige Länge

  Modifikation durch

  ```
  stackElements.set(top,x)
  ```
  existierendes Element ändern

  ```
  stackElements.add(top,x)
  ```
  erweitern an der Stelle top