

# Erweiterte Konstrukte

- Sets
- Wert und Referenztypen
- Generics

Basiert auf Material von:

Kurt Bleisch

Stephan Neuhaus

Karl Rege

Marcela Ruiz

Jürgen Spielberger





# Sets

# Set: als ADT

- Das Set (ADT) ist eine ungeordnete Menge **ohne** Duplikate.  
Eine wichtige Operation ist **contains**, um herauszufinden, ob ein Objekt Teil der Menge ist.
- Interface Set: Es fehlen im Vergleich zur Liste folgende Methoden (da die Menge ungeordnet ist):
  - `get(index)`
  - `add(index, value)`
  - `remove(index)`
- Implementation durch:
  - **HashSet** bei grossen Datenbeständen (etwas) effizienter
  - **TreeSet** speichert die Elemente in alphabetischer Folge
  - und weitere...

Beide Implementationen sind sehr effizient.

# Set: Anwendungsbeispiel

1. Alle Wörter eines Texts sollen gesammelt werden, um dann später die Liste der Wörter aufzulisten.

Hinweis:

Geht anstatt mit Set auch mit Listen. Vor dem Einfügen überprüfen ob das Wort schon in der Liste vorhanden ist.

```
if (!list.contains(s)) list.add(s)
```

2. Frage: Was sind die gemeinsamen Wörter von zwei Texten?

Die Lösung wird mittels Mengenoperation ermittelt (siehe Folie 6).

# Set: Anwendungsbeispiel

- Folgendes Beispiel zeigt die Anwendung von **HashSet**:

```
Set stooges = new HashSet();
stooges.add("Larry");
stooges.add("Moe");
stooges.add("Curly");
stooges.add("Moe");    // duplicate, won't be added
stooges.add("Shemp");
stooges.add("Moe");    // duplicate, won't be added
System.out.println(stooges);
```

Output: [Moe, Shemp, Larry, Curly]

- Die Reihenfolge ist zufällig (später mehr)
- Falls **TreeSet** verwendet wird:

```
Set stooges = new TreeSet();
```

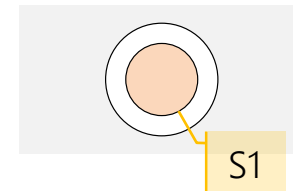
Output: [Curly, Larry, Moe, Shemp]

- Die Reihenfolge ist alphabetisch

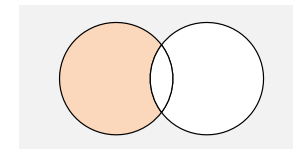
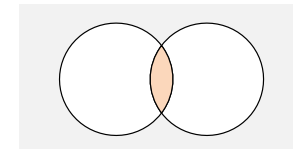
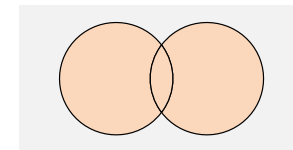
# Set: Operationen

Sets unterstützen die gängigen Mengenoperationen:

- Um zwei Sets miteinander zu vergleichen:
  - Subset: S1 ist eine Teilmenge von S2, wenn S2 jedes Element von S1 enthält.  
**containsAll** Testet für eine Teilmengenbeziehung.



- Mengenoperationen zu Sets:
  - Union: S1 vereinigt mit S2 enthält alle Elemente, die in S1 oder S2 sind.  
**addAll** führt Union aus.
  - Intersection: Die Schnittmenge von S1 und S2 enthält die Elemente, die sowohl in S1 als auch in S2 enthalten sind.  
**retainAll** führt Intersection aus.
  - Difference: Die Differenz von S1 und S2 enthält die Elemente, die in S1 sind, die aber nicht in S2 sind.  
**removeAll** führt Difference aus.





## Wert-Typen und Referenz-Typen

Vorbereitung zum  
nächsten Thema in der  
heutigen Vorlesung.

# Wert- und Referenz-Typen: Fragen

Welchen Wert hat a[0]?

```
int[] a = {3,1,2};  
int[] b;  
b = a;  
b[0] = 1;
```

Ist folgende Klassendeklaration korrekt?

```
public class C {  
    public C p;  
}
```

```
C c = new C();  
c.p = c;
```



# Wert- und Referenz-Typen

- Wert-Typen in Java sind:

- byte, short, int, long
- float, double
- char
- boolean

`int` ist Werttyp, `Integer` ist Referenztyp (Wrapper-Klasse).

`float` ist Werttyp, `Float` ist Referenztyp.

Beispiel:

```
int a, b;  
a = 3; b = a;
```

- Eingebaute Referenz-Typen in Java sind:

- Arrays
- Alle von Object abgeleiteten Klassen
- String (verhält sich bei Operationen wie ein Wert-Typ, `s = s + "aa" + "b"`)

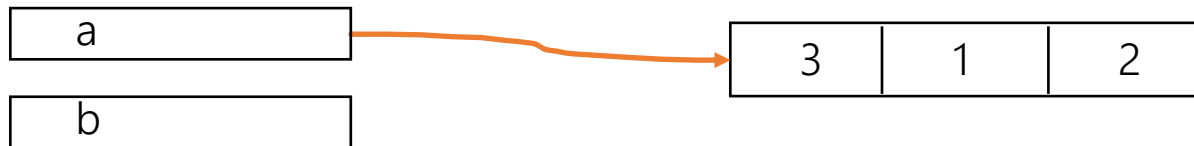
Beispiel:

```
int[] a = {3,1,2};  
int[] b;  
b = a;  
b[0] = 1; // a[0] = 1
```

# Wert- und Referenz-Typen

Objekt-Variablen sind lediglich Referenzen (Zeiger) auf Objekte.

```
int[] a= {3,1,2};  
int[] b;
```



am Anfang zeigen diese nirgendwo hin: null

```
int[] a,b;  
a[0] = 3; // Fehler
```

können mittels der Zuweisung gesetzt:

```
a = new int[3];  
b = a;
```

oder wieder zu null gesetzt:

```
a = null;
```

# Wert- und Referenz-Typen

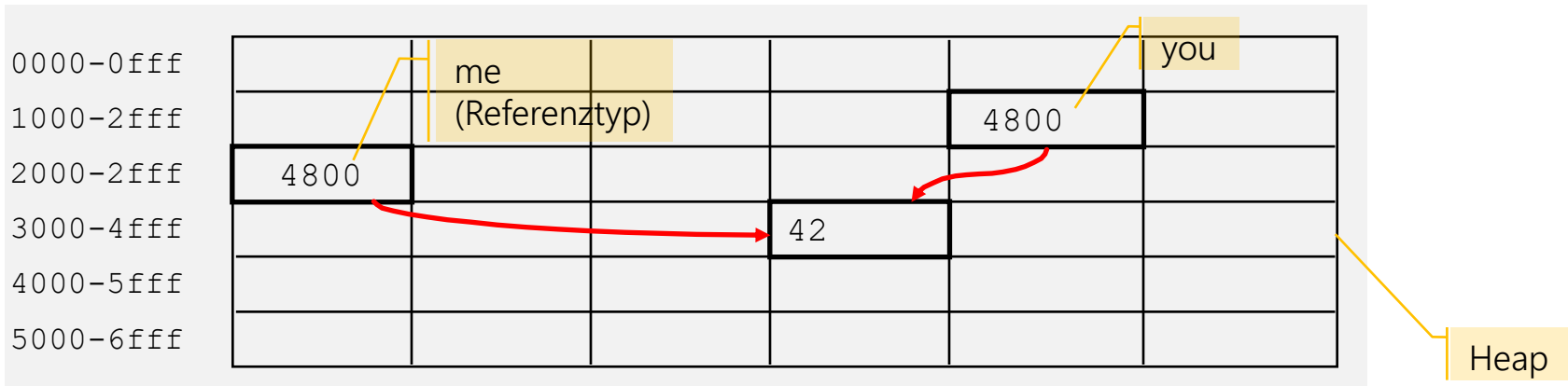
```
int radius = 42;
```

0000-0fff						
1000-2fff		radius (Werttyp)				
2000-2fff	42					
3000-4fff						
4000-5fff						
5000-6fff						

## Begriffe:

- Variable      benannter Speicherplatz für einen Wert, statt Adresse 2000 können wir radius schreiben
- Wert            «Nutz»-Daten, z.B. 42
- Zuweisung    Deponieren eines Wertes in einer Variablen, radius = 42
- Ausdruck      Liefert einen Wert: radius = 6\*7;

# Wert- und Referenz-Typen



Zeiger (Referenzen) sind lediglich Verweise auf den eigentlichen Wert

```
class MyClass {
    int val;
}

MyClass me = new MyClass(); me.val = 42;
MyClass you = me;
```



# Java Generics

# Java-Generics im Collection-Framework

Bisher:

```
List list = new LinkedList();  
list.add(new Integer(99));  
list.add(new String("a"));  
Integer i = (Integer)list.get(1);
```

- Gefährlich:
  - kein Schutz gegen Einfügen von Elementen vom «falschen» Typ
  - **ClassCastException** zur Laufzeit möglich bei falschem Typcast
- Lösung:  
try-catch, Befüllung der List absichern oder sonstige Klammzüge
- Wünschenswert:  
Collection Klassen spezialisiert auf bestimmte Elementtypen.

# Java-Generics im Collection-Framework

Typparameter, müssen  
Referenztypen sein.

```
List<Integer> list = new LinkedList<>();  
list.add(new Integer(99));
```

Wrapperklasse «wandelt»  
in Referenztyp um.

- Generische Datentypen  
Bsp.: **List<Integer>**, **LinkedList<Integer>**
- Duden: Generisch = «Die Gattung betreffend», «Gattungs...»
- Man liest auch: «parametrisierter Datentyp» oder «parametrischer Datentyp»
- Deklaration des Datentyps, den die Collection aufnehmen soll: hier, der **Typparameter** ist vom Typ Integer.
- Regeln sind sehr restriktiv, z.B.

```
list.add(new Object()); ➡ Compile-Error
```

Jede Pgm.-Sprache hat  
etwas andere Regeln.

# Java-Generics im Collection-Framework

Ab Java 5, Sep. 2004:

```
List<Integer> list = new LinkedList<>();  
list.add(new Integer(99)); // oder mit Boxing: list.add(99);  
Integer i = list.get(0);   // oder mit Unboxing: int i = list.get(0);  
list.add(new Double(3.1415));      ⇒ Compile-Error
```

Werttyp

Autoboxing um automatisch Wert- in Referenztypen umzuwandeln und umgekehrt.



# Java-Generics - Vorteile

- Java-Generics-Erweiterung des Collection Frameworks:
  - Erhöht die Effizienz von Collections, macht sie vielseitiger einsetzbar.
  - Senkt die Gefahr von Fehlern bei Typprüfungen zur Laufzeit.
  - Erhöht die Lesbarkeit, Aussagekraft und erleichtert Verständnis des Codes.
  - Einfügen von anderen Typen werden zur **Compile-Zeit** schon abgelehnt.
  - Cast beim Lesen kann entfallen, da garantiert keine anderen Typen enthalten sind.
- Erleichtert so die Implementierung von Datenstrukturen wie «Containern».
- Möglichkeit zur Erstellung von eigenen Klassenschablonen und Methodenschablonen.
- Erleichtert das Schreiben von Algorithmen, die mit verschiedenen Datentypen funktionieren.



# Entwicklung von generischen Klassen ohne Typparameter

# Generizität

- Datentypen wurden in die Programmiersprachen eingeführt, um die Anwendung von Programmiersprachen sicherer zu machen
  - Assembler und frühe C Versionen kannten keine Datentypen -> oft Quelle von Fehlern
- Mit dem Datentyp wird die Menge der Operationen (Operatoren), die auf Werte dieses D.T. angewandt werden können, eingeschränkt (constraint)
  - z.B.  $6 * 8$ , aber nicht `"hallo" * "world"` oder `int i = "hallo"`
- Es soll aber trotzdem möglich sein, eine Datenstruktur / einen Algorithmus auf Werte verschiedener Datentypen anzuwenden, z.B.:
  - Datenstrukturen: Liste, Stack, Queue für Integer, Float, etc.
  - Algorithmen: Sortieren, Suchen, etc.

Einen Algorithmus, der auf Werte von unterschiedlichen Datentypen angewandt werden kann, nennt man **generisch**.

# Generizität: ohne Typparameter

Generizität erreicht durch:

## 1. Überladen von Methoden

```
int max(int i, int j);  
double max(double d, double e);
```

## 2. Object als Parameter (als Oberklasse aller Klassen)

```
class Box {  
    private Object val;  
  
    void setValue(Object val ) {  
        this.val = val; }  
  
    Object getValue() {  
        return val; }  
}
```

```
Box.setValue(new Integer(32));  
int i = (Integer)Box.getValue();
```

- Beim Lesen muss ein Cast zum gewünschten Typ durchgeführt werden.
- Es können Werte von beliebigem Typ eingefügt werden, obwohl das u.U. keinen Sinn macht: Laufzeitfehler beim Lesen (TypeCastException).

# Generizität: ohne Typparameter

3. Für jeden Datentyp eine eigene Klasse.

Bsp.: Container für eine einfache Zahl (Datentyp int):

```
class IntBox {  
    private int val;  
  
    void setValue(int val) {  
        this.val = val; }  
  
    int getValue() {  
        return val; }  
}
```

```
class StringBox {  
    private String val;  
  
    void setValue(String val ) {  
        this.val = val; }  
  
    String getValue() {  
        return val; }  
}
```

Das Gleiche für String,  
Integer, Float, Double, etc.



# Entwicklung von generische Klassen und Interfaces mit Typparametern

# Generische Klassen

- Für den Typ wird lediglich ein Platzhalter, z.B. T, E, V eingesetzt.
- Der Typ kann später bei der Instanzierung (Variablendeklaration) festgelegt werden.

```
class Box<T> {  
    private T val;  
  
    void setValue(T val ) {  
        this.val = val;  
    }  
  
    T getValue() {  
        return val;  
    }  
}
```

Platzhalter durch < >  
gekennzeichnet.

- Beispiele für Deklarationen (für T nur Referenztypen erlaubt):

```
Box<String> box = new Box<>();  
Box<Integer> box = new Box<>();  
Box<Point> box = new Box<>();
```

# Generische Klassen: Verwendung

- Es sind keine Typenumwandlungen (Casts) notwendig
- Fehler werden während der Übersetzung erkannt.

```
Box<String> box = new Box<>();  
box.setValue("hallo");  
String x = box.getValue();
```

- Funktioniert auch mit einfachen Typen, dank Autoboxing

```
Box<Integer> box = new Box<>();  
box.setValue(42);  
int x = box.getValue();  
box.setValue("hallo"); --> Compile-Error
```



# Generische Klassen: Mehrere Typen

Mehrere Platzhaltertypen werden einfach durch «,» abgetrennt

```
public interface Map <K, V> {  
    public void put(K key, V value);  
    public V get(K key);  
    ...  
}
```

K ist im JDK Object wegen  
Abwärtskompatibilität:  
V get(Object key)

# Generische Interfaces

```
public interface List<E> {  
    public void add(E e);  
    public E get(int i);  
    ...  
}
```

```
public class LinkedList <E> implements List<E> {  
    private E first= null;  
  
    public void add(E e){  
        ...  
    }  
  
    ...  
}
```

```
List<Integer> list = new LinkedList<>();
```



## Entwicklung von generische Methoden

# Generische Methoden

<T> vor dem Rückgabebetyp.

```
static <T> void foo(T arg) {  
    ...  
}
```

```
foo(4);
```

- Der konkrete Typ muss nicht angegeben werden. Er wird anhand der Parameter hergeleitet (Type Inference / Typ Ableitung).

```
static <T> T foo(T t) {  
    return t;  
}
```

```
int i = foo(4);  
int i = foo(4.3); --> Compile-Error
```

- Generische Methoden können auch in nicht-generischen Klassen verwendet werden.



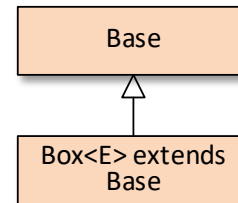
# Vererbung und Subtyping von generischen Klassen

# Vererbung & Subtyping: Generischen Klassen

Es kann auf drei Arten von generischen Klassen geerbt werden:

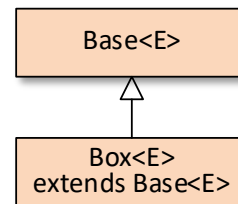
1. Die erbende Klasse ist generisch, die Beerbte nicht.

```
class Box<E> extends Base
```



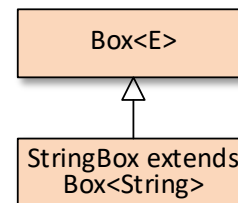
2. Die erbende Klasse bleibt weiterhin generisch.

```
class Box<E> extends Base<E>
```



3. Die erbende Klasse konkretisiert den Typ.

```
class StringBox extends Box<String>
```



# Vererbung & Subtyping: Klassenhierarchie

```
List<String> ls = new ArrayList<>();  
List<Object> lo = ls;
```

Zeigen auf dieselbe Liste!

Problem:

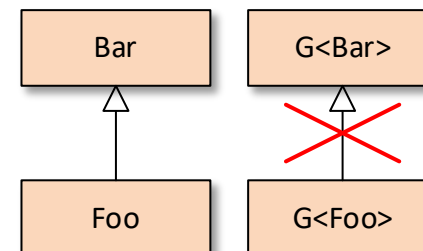
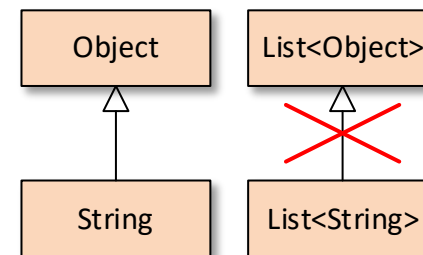
```
lo.add(new Integer(4)); // unsafe  
String s = ls.get(0);  // run time error
```

→ daher «per Definition» verboten.

Wenn **Bar** Oberklasse von **Foo** ist und **G** eine generische Typendeklaration (z.B. **List<Bar>**) dann ist **G<Bar>** keine Oberklasse von **G<Foo>**.

Compile time error:  
Incompatible Types

Ist der Code ok?



# Übung: Generischer ADT – Stack-Interface

Definieren Sie das Stack-Interface generisch (später mittels Array oder Liste zu implementiert).

Der konkrete Typ kann später mittels Typparameter festgelegt werden.

```
public interface Stack {  
    public void push(Object obj) throws StackOverflowError;  
    public Object pop() throws EmptyStackException;  
    public Object peek() throws EmptyStackException;  
    public void removeAll();  
    public boolean isEmpty();  
    public boolean isFull();  
}
```

Stack nicht generisch.



## Übung: Generischer ADT - ListStack

Implementieren Sie die generische Klasse ListStack (implementiert Interface Stack) mittels LinkedList (public class LinkedList<E>). Erstellen Sie die generische Methodensignatur für push, noch ohne konkrete Implementation.

```
public interface Stack<T> {  
    public void push(T obj) throws StackOverflowError;  
    ...  
}
```

## Übung: Generischer ADT - ListStack

Implementieren Sie jetzt die push-Methode Ihrer ListStack-Klasse mittels einer generischen Liste.

```
public class ListStack<T> implements Stack <T> {  
  
    // Implementation mit einer LinkedList  
    LinkedList<T> data;  
  
    @Override  
    public void push(T obj) throws StackOverflowError {  
        // Element auf den Stack legen  
    }  
  
    ...  
}
```

Wie wird das implementiert?

## Übung: Generischer ADT - ListStack

Instanziiieren Sie eine ListStack-Klasse vom Typ Integer.

```
public class ListStack<T> implements Stack <T> {  
  
    // Implementation mit einer LinkedList  
    LinkedList<T> data;  
  
    @Override  
    public void push(T obj) throws StackOverflowError {  
        // Element auf den Stack legen  
    }  
  
    ...  
}
```



«?» Wildcards für Typparameter

# Klassen mit Wildcards

```
class A { /* ... */ }  
class B extends A { /* ... */ }
```

```
B b = new B();  
A a = b; // ok
```

Das Beispiel zeigt, dass die Vererbung von regulären Klassen der Regel der Subtypisierung folgen: Klasse B ist ein Subtyp von Klasse A, wenn B A erweitert.

```
List<B> lb = new ArrayList<>();  
List<A> la = lb; --> Compile-Error
```

Diese Regel gilt **nicht** für generische Typen.

```
Box<Object> b;  
Box<Integer> bI = new Box<>();  
Box<Double> bD = new Box<>();
```

Weiteres Beispiel mit generischen Typen.

```
b = bI; --> Compile-Error  
b = bD; --> Compile-Error
```

Ein bewusstes «Vergessen» der Typinformationen lässt sich durch das Wildcard Zeichen «?» erreichen.

# Klassen mit Wildcards

- Wildcards dienen dazu, unterschiedliche parametrisierte Typen zuweisbar zu machen und damit in einer Methode benutzbar zu machen.
- Es erlaubt, verschiedene Unterklassen zusammenzuführen.
- **Nur lokale Variablen und Methoden-Parameter** können mit Wildcards benutzt werden!

```
Box<Object> b;  
Box<Integer> bI = new Box<>();  
Box<Double> bD = new Box<>();  
Box<?> bw;
```

```
bw = bD; // ok  
bw = bI; // ok  
bw = b;  // ok
```

# Methoden mit Wildcards

Eine Methode die alle enthaltenen Elemente einer Collection ausgibt.

«Früher» (ohne for-each-Schleife):

Ohne Typparameter.

```
public void printCollection(Collection c) {  
    iterator iter = c.iterator();  
    while (iter.hasNext()) {  
        System.out.println(iter.next());  
    }  
}
```

# Methoden mit Wildcards

Eine Methode die alle enthaltenen Elemente einer Collection ausgibt.

Naiver Ansatz (mit for-each-Schleife):

Mit Typparameter.

```
public void printCollection(Collection<Object> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

- Funktioniert aber nur mit Collections mit dem Parametertyp <Object>.
- List<String> kann ja nicht List<Object> zugewiesen werden (siehe Folie 31).



# Methoden mit Wildcards

Eine Methode die alle enthaltenen Elemente einer Collection ausgibt.

Mit Wildcars:

```
public void printCollection(Collection<?> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

Die Typkompatibilität wird  
nicht mehr geprüft.

`Collection<?>` ist die «Oberklasse» aller Collections

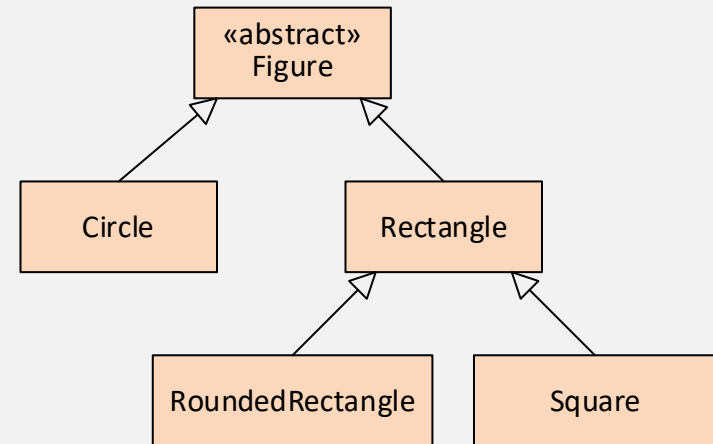


# Bounded Wildcards

# Bounded Wildcards

```
abstract class Figure {  
    public abstract void draw();  
}  
  
class Circle extends Figure {  
    public void draw() {};  
}  
  
class Rectangle extends Figure {  
    public void draw() {};  
}  
  
class RoundedRectangle extends Rectangle {  
    public void draw() {};  
}  
  
class Square extends Rectangle {  
    public void draw() {};  
}
```

```
public void drawAll(List<Figure> figures) {  
    for (Figure f : figures) f.draw();  
}  
  
List<Circle> lc = new LinkedList<>();  
drawAll(lc); // ok ?
```



Compile time error: Incompatible Types.  
Regel der Subtypisierung gelten nicht  
für generische Typen (wie bereits  
aufgezeigt).

# Upper Bounded Wildcards

```
public void drawAll(List<Figure> figures) {  
    for (Figure f : figures) f.draw();  
}
```

```
List<Circle> lc = new LinkedList<>();  
drawAll(lc); --> Compile-Error
```

- Problem: `List<Figure>` ist zu restriktiv `List<?>` zu offen (keinerlei Information über Typ).
- Wir wollen eigentlich ausdrücken: «irgend ein Typ der von Figure erbt»  
⇒ Upper Bound Wildcard

Soooo geht das!

```
public void drawAll(List<? extends Figure> figures) {  
    for (Figure f : figures) f.draw();  
}
```

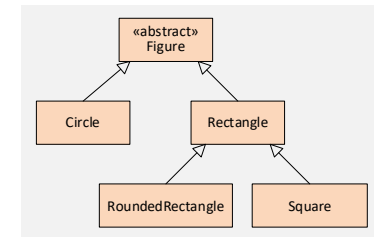
```
List<Circle> lc = new ArrayList<>();  
drawAll(lc);
```

# Lower Bounded Wildcards

```
public void addRectangle(List<? extends Figure> rects) {  
    rects.add(new RoundedRectangle()); // OK ?  
}
```

Upper Bound Wildcard

- Methode die ein neues **RoundedRectangle** zu einer generischen Liste hinzufügt.
- Problem: **List<? extends Figure>** könnte eine Liste von Circles oder Squares sein und wir fügen ein gerundetes Rechteck hinzu, das möchten wir verhindern.
- Wir wollen ausdrücken, dass der Typ der Liste **RoundedRectangle**, oder deren Oberklassen **Rectangle** oder **Figure** ist  $\Rightarrow$  Lower Bound Wildcard



```
public void addRectangle(List<? super RoundedRectangle> rects) {  
    rects.add(new RoundedRectangle); // OK?  
}
```

Lower Bound Wildcard

Ja: Liste von Rectangle und RoundedRectangle sind erlaubt.

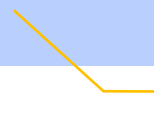
# Bounded Wildcards, Abhängigkeiten

- Abhängigkeit zwischen zwei Typen in Kombination mit Wildcards:

```
class Collections {  
    public static <T> void copy(List<T> dest, List<? extends T> src) { ... }  
}
```

- auch möglich:

```
class Collections {  
    public static <T, S extends T> void copy(List<T> dest, List<S> src) {...}  
}
```



Upper Bound Wildcard

# Bounded Wildcards bei Rückgabotyp

- Upper bound wildcards (extends) erlauben lesende Funktionen, also Funktionen, die einen parametrisierten Typ zurückgeben.

```
public List<? extends Figure> getFigures () { }
```

## Übung: Generischer ADT

Definieren Sie eine generische PriorityQueueGen-Klasse, so dass der Wertetyp generisch und die Priorität ein von Comparable abgeleiteter Typ ist.

```
public class PriorityQueue {  
    public push(Object o, Priority p) {...}  
}
```

Soll generisch sein.

Soll generisch sein und von Comparable abgeleitet.



## Übung: Generischer ADT

Aufgabe hat nichts mehr  
mit Wildcards zu tun!

Definieren Sie eine generische max-Methode, welche aus einer Collection den höchsten Wert liefert. Die Werte der Collection müssen von identischem Typ und vergleichbar (Comparable) sein.

# Zusammenfassung

- Sets
- Wert und Referenztypen
- Generics
  - Generische Typen und Methoden erhöhen die Sicherheit, da Typprüfungen zur Laufzeit reduziert werden.
  - Viele falsche Typzuweisungen werden vom Compiler abgelehnt.
  - Die Aussagekraft der Quelle wird erhöht.
  - Es gibt aber Einschränkungen und Ausnahmen, die das Arbeiten mit Java Generics erschweren.



Kontrollfragen Lektion 3  
nicht vergessen – heute mit  
Schneewittchen





Nerd-Zone



## Raw Types & Type-Erasure



## Raw Type

- Wenn man eine Variable ohne «<>» deklariert, dann spricht man von einem Raw Type.
- Raw Types und Generic Types sind Zuweisungskompatibel; die statische Typensicherheit geht aber verloren; es werden deshalb vom Compiler Warnungen generiert.
- Mittels `SuppressWarnings("unchecked")` lassen sich diese ausschalten.

```
Box<String> bs = new Box<>;  
Box raw;    //Raw Type
```

```
@SuppressWarnings("unchecked")  
  
raw = bs;           // ok  
bs = raw;           // Unchecked Assignment Warning  
bs = (Box<String>)raw; // Unchecked Cast Warning
```



# Type Erasures (Typlöschung)

- Java entfernt die Typeninformation vollständig zur Laufzeit
- Entscheid des Java Design Teams zur Implementierung der Aufwärtskompatibilität
- Aus `Box<T>` wird zur Laufzeit `Box<Object>`
- Störend: Daraus ergeben sich Einschränkungen
  - keine Typenprüfung möglich: `if (e instanceof List<Integer>) ...`
  - Cast sind nicht überprüfbar -> Warnung `E e = (E)o;`
  - kein Instanzierung möglich `E e = new E();`
  - keine Arrays von `E[] a = new E[10];`



# Type Erasures

- Beim Ablauf des Programms kann z.B. nicht mehr von `LinkedList<String>` und `LinkedList<Integer>` unterschieden werden. Beide haben zur Laufzeit den Typ `LinkedList<Object>`, die Ausgabe in der Console ist «true».

```
List<String> l1 = new ArrayList<>();  
List<Integer> l2 = new ArrayList<>();  
System.out.println(l1.getClass() == l2.getClass());
```

- Auch Casts funktionieren nicht wirklich

```
<T> T badCast(Object o) {  
    return (T) o; // unchecked warning  
}
```

```
String x = badCast(myObject); --> Runtime-Error
```



# Type Erasures und Erzeugung von Instanzen

- Erzeugung von Instanzen funktioniert nicht

```
public class badCast<E> {  
    public E create() {  
        return new E(); // Type-Erasure in Action  
    }  
}
```

- Lösung: man gebe beim Aufruf noch die Class<T> mit, dann kann ich mit `getDeclaredConstructor().newInstance()` eine Instanz erzeugen:

```
<T> T foo(Class<T> clazz) {  
    return (T)clazz.getDeclaredConstructor().newInstance();  
}
```

```
String s = foo(String.class);
```