

Java Quiz

```
public abstract class Enum<E extends Enum<E>>
{
    // ...
}
```

```
public class Collections
{
    // ...

    public static <T> int binarySearch
        (List<? extends Comparable<? super T>> list, T key);

    // ...

    public static <T extends Object & Comparable<? super T>>
        T min(Collection<? extends T> coll);

    // ...
}
```

Bigger Than Long

```
BigInteger x = BigInteger.valueOf(2);  
for (int i = 0; i <= 64; ++i)  
{  
    x.add(x);  
}  
  
System.out.println(x);
```

Bigger Than Long

```
BigInteger x = BigInteger.valueOf(2);
```

```
for (int i = 0; i <= 64; ++i)
{
    x.add(x);
}
```

```
System.out.println(x);
```

```
/**
 * Immutable arbitrary-precision integers.
 */
public class BigInteger

/**
 * Returns a BigInteger whose value is (this + val).
 */
public BigInteger add(BigInteger val);
```

Bigger Than Long

```
BigInteger x = BigInteger.valueOf(2);
```

```
for (int i = 0; i <= 64; ++i)
{
    x = x.add(x);
}
```

```
System.out.println(x);
```

```
/**
 * Immutable arbitrary-precision integers.
 */
public class BigInteger

/**
 * Returns a BigInteger whose value is (this + val).
 */
public BigInteger add(BigInteger val);
```

Frühstücksbrühe

```
String s = "Frühstücksbrühe";  
int b = s.indexOf('b');  
int zähler = 0;  
  
for (int i = 0; i < 5; ++i)  
{  
    Character x = s.charAt(i);  
    Character y = s.charAt(b + i);  
  
    if (x == y)  
    {  
        ++zähler;  
    }  
}  
  
System.out.println(zähler);
```

Frühstücksbrühe

```
String s = "Frühstücksbrühe";  
int b = s.indexOf('b');  
int zähler = 0;  
  
for (int i = 0; i < 5; ++i)  
{  
    Character x = s.charAt(i);  
    Character y = s.charAt(b + i);  
  
    if (x == y)  
    {  
        ++zähler;  
    }  
}  
  
System.out.println(zähler);
```

Früh
s
brü
he

Boxing

Ideally, boxing a given primitive value *p* would always yield an identical reference.

In practice, this may not be feasible using existing implementation techniques.

```
public final class Character
{
    // ...

    public static Character valueOf(char c)
    {
        if (c <= 127)
        {
            return CharacterCache.cache[(int) c];
        }
        return new Character(c);
    }
}
```

Mensch Meier!

```
class Lebewesen
{
    protected Lebewesen()
    {
        System.out.println(this);
    }
}

class Mensch extends Lebewesen
{
    private String _name = "Kain";

    public Mensch(String name) { _name = name; }

    public String toString() { return _name; }
}

new Mensch("Meier");
```

Mensch Meier!

```
class Lebewesen
{
    protected Lebewesen()
    {
1      System.out.println(this);
    }
}

class Mensch extends Lebewesen
{
3   private String _name = "Kain";

4   public Mensch(String name) { _name = name; }

2   public String toString() { return _name; }
}

new Mensch("Meier");
```

Objektinitialisierung

Die `this`-Referenz hat im Superkonstruktor bereits den dynamischen Typ der Subklasse, obwohl das entsprechende Objekt noch gar nicht initialisiert wurde.

Konstrukturen sollten daher niemals (direkt oder indirekt) redefinierbare Operationen der eigenen Klasse aufrufen!

Sternstunden

```
int[] array = {1492, 1789, -65000000, -753, 0};  
Arrays.sort(array);
```

```
int i = Arrays.asList(array).indexOf(1492);  
System.out.println(i * i);
```

Sternstunden

```
int[] array = {1492, 1789, -65000000, -753, 0};  
Arrays.sort(array);
```

```
int i = Arrays.asList(array).indexOf(1492);  
System.out.println(i * i);
```

Vor der Sortierung:

[1492, 1789, -65000000, -753, 0]

Nach der Sortierung:

[-65000000, -753, 0, 1492, 1789]

Sternstunden

```
int[] array = {1492, 1789, -65000000, -753, 0};  
Arrays.sort(array);
```

```
int i = Arrays.asList(array).indexOf(1492);  
System.out.println(i * i);
```

Vor der Sortierung:

[1492, 1789, -65000000, -753, 0]

Nach der Sortierung:

[-65000000, -753, 0, 1492, 1789]

```
// Returns a fixed-size list backed by the specified array.  
public static <T> List<T> asList(T... a);
```

Sternstunden

```
int[] array = {1492, 1789, -65000000, -753, 0};  
Arrays.sort(array);
```

```
int i = Arrays.asList(array).indexOf(1492);  
System.out.println(i * i);
```

Vor der Sortierung:

[1492, 1789, -65000000, -753, 0]

Nach der Sortierung:

[-65000000, -753, 0, 1492, 1789]

```
// Returns a fixed-size list backed by the specified array.  
public static <T> List<T> asList(T... a);
```

```
public static      List<int[]> asList(int[]... a);
```

Sternstunden

```
Integer[] array = {1492, 1789, -65000000, -753, 0};  
Arrays.sort(array);
```

```
int i = Arrays.asList(array).indexOf(1492);  
System.out.println(i * i);
```

Vor der Sortierung:

[1492, 1789, -65000000, -753, 0]

Nach der Sortierung:

[-65000000, -753, 0, 1492, 1789]

```
// Returns a fixed-size list backed by the specified array.  
public static <T> List<T> asList(T... a);
```

```
public static      List<Integer> asList(Integer... a);
```

Generische, variadische Funktionen

Primitive Typen sind keine Referenztypen und können daher nicht als generische Typparameter verwendet werden.

Daher existiert zu jedem primitiven Typ eine entsprechende Wrapperklasse mit impliziten Konvertierungsregeln (Auto-Boxing).

Arrays von primitiven Typen sind Referenztypen.

Es gibt keine Konvertierungsregeln von `int[]` nach `Integer[]`.

Generische, variadische Funktionen behandeln `int[]` als ein einziges Argument, `Integer[]` dagegen als Argumentliste.

Kleine Primzahlen

```
List<Integer> primes = new ArrayList<Integer>();  
  
primes.add(1);           // whoops, not actually a prime...  
primes.add(2);  
primes.add(3);  
primes.add(5);  
primes.add(7);  
  
primes.remove(1);        // ...so let's remove it again!
```

Kleine Primzahlen

```
List<Integer> primes = new ArrayList<Integer>();  
  
primes.add(1);           // whoops, not actually a prime...  
primes.add(2);  
primes.add(3);  
primes.add(5);  
primes.add(7);  
  
primes.remove(1);       // ...so let's remove it again!  
  
System.out.println(primes);  
  
[1, 3, 5, 7]
```

Kleine Primzahlen

```
primes.add(1);          // whoops, not actually a prime...  
primes.remove(1);      // ...so let's remove it again!
```

```
public interface List<E> extends Collection<E> {  
    /**  
     * Removes the first occurrence of the specified element  
     */  
    boolean remove(Object o);  
  
    /**  
     * Removes the element at the specified position  
     */  
    E remove(int index);  
}
```

Kleine Primzahlen

```
collection<Integer> primes = new ArrayList<Integer>();
```

```
primes.add(1);           // whoops, not actually a prime...  
primes.add(2);  
primes.add(3);  
primes.add(5);  
primes.add(7);
```

```
primes.remove(1);       // ...so let's remove it again!
```

```
System.out.println(primes);
```

```
[2, 3, 5, 7]
```

Hello, Hello, Hello

```
static void createStrings(String x, String y, String z)
{
    x = "hello";
    y = "hello" + "";
    z = x + "";
}

public static void main(String[] args)
{
    String a[] = new String[3];
    createStrings(a[0], a[1], a[2]);

    Set<String> set = new HashSet<String>(Arrays.asList(a));
    System.out.println(set.size());
}
```

Hello, Hello, Hello

```
static void createStrings(String x, String y, String z)
{
    x = "hello";
    y = "hello" + "";
    z = x + "";
}

public static void main(String[] args)
{
    String a[] = new String[3];
    createStrings(a[0], a[1], a[2]);

    Set<String> set = new HashSet<String>(Arrays.asList(a));
    System.out.println(set.size());
}

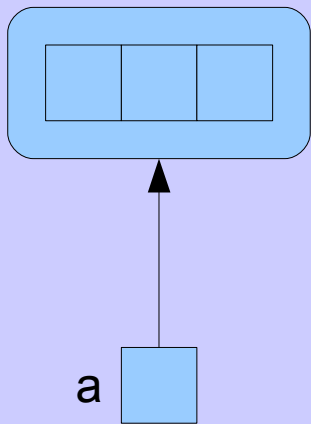
System.out.println(set);

[null]
```

Hello, Hello, Hello

```
static void createStrings(String x, String y, String z)
{
    x = "hello";
    y = "hello" + "";
    z = x + "";
}
```

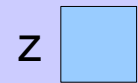
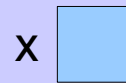
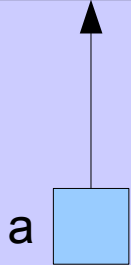
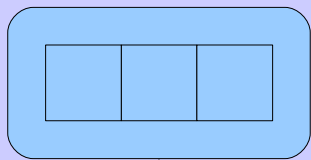
```
public static void main(String[] args)
{
    String a[] = new String[3];
    createStrings(a[0], a[1], a[2]);
}
```



Hello, Hello, Hello

```
static void createStrings(String x, String y, String z)
{
    x = "hello";
    y = "hello" + "";
    z = x + "";
}
```

```
public static void main(String[] args)
{
    String a[] = new String[3];
    createStrings(a[0], a[1], a[2]);
}
```



☐ 0

☒ 1

☐ 2

☐ 3

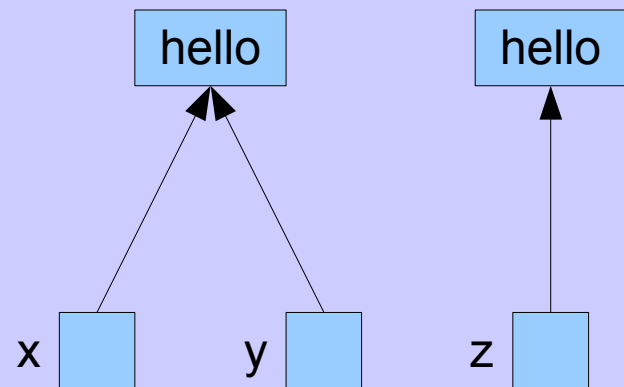
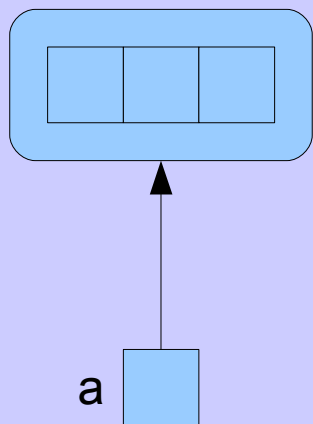
☐ Compiler ☹

☐ Laufzeit ☹

Hello, Hello, Hello

```
static void createStrings(String x, String y, String z)
{
    x = "hello";
    y = "hello" + "";
    z = x + "";
}
```

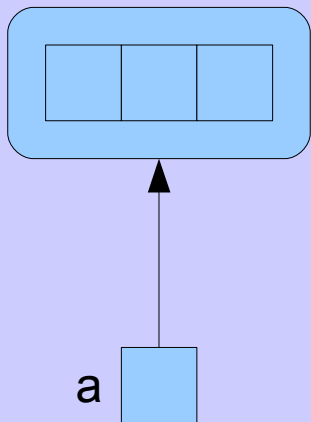
```
public static void main(String[] args)
{
    String a[] = new String[3];
    createStrings(a[0], a[1], a[2]);
}
```



Hello, Hello, Hello

```
static void createStrings(String x, String y, String z)
{
    x = "hello";
    y = "hello" + "";
    z = x + "";
}
```

```
public static void main(String[] args)
{
    String a[] = new String[3];
    createStrings(a[0], a[1], a[2]);
}
```



Parameterübergabe

Java hat nur einen Parameterübergabemechanismus:
Übergabe per Wert (*call by value, pass by value*).

Bei der Übergabe per Wert wirken sich Zuweisungen auf formalen Parametern niemals auf die aktuellen Parameter aus. Das wäre nur mittels Übergabe per Referenz möglich, die es in Java nicht gibt.

In Java werden niemals Variablen oder Objekte übergeben, sondern ausschließlich primitive Werte oder Referenzen.

Falsch: „Objekte werden per Referenz übergeben.“

Korrekt: „Referenzen werden per Wert übergeben.“

Eine swap-Funktion, die die Belegung zweier Variablen vertauscht, ist in Java (mangels Übergabe per Referenz) nicht realisierbar.

Vor lauter Bäumen...

```
Set<StringBuilder> set = new TreeSet<StringBuilder>();  
StringBuilder x = new StringBuilder("hello");  
StringBuilder y = new StringBuilder("world");  
set.add(x);  
set.add(y);  
  
System.out.println(set.contains(y));  
y.append("!");  
System.out.println(set.contains(y));  
y.delete(0, 5);  
System.out.println(set.contains(y));
```

Vor lauter Bäumen...

```
Set<StringBuilder> set = new TreeSet<StringBuilder>();
StringBuilder x = new StringBuilder("hello");
StringBuilder y = new StringBuilder("world");
set.add(x);
set.add(y);    // stringBuilder cannot be cast to Comparable

System.out.println(set.contains(y));
y.append("!");
System.out.println(set.contains(y));
y.delete(0, 5);
System.out.println(set.contains(y));
```

Vor lauter Bäumen...

```
Set<StringBuilder> set = new TreeSet<StringBuilder>(new C());  
StringBuilder x = new StringBuilder("hello");  
StringBuilder y = new StringBuilder("world");  
set.add(x);  
set.add(y);
```

```
System.out.println(set.contains(y));  
y.append("!");  
System.out.println(set.contains(y));  
y.delete(0, 5);  
System.out.println(set.contains(y));
```

```
class C implements Comparator<StringBuilder>  
{  
    public int compare(StringBuilder x, StringBuilder y)  
    {  
        return x.toString().compareTo(y.toString());  
    }  
}
```

Vor lauter Bäumen...

```
Set<StringBuilder> set = new TreeSet<StringBuilder>(new C());
StringBuilder x = new StringBuilder("hello");
StringBuilder y = new StringBuilder("world");
set.add(x);
set.add(y);           // hello < world
```

```
System.out.println(set.contains(y));
y.append("!");        // hello < world!
System.out.println(set.contains(y));
y.delete(0, 5);       // hello < !
System.out.println(set.contains(y));
```

```
class C implements Comparator<StringBuilder>
{
    public int compare(StringBuilder x, StringBuilder y)
    {
        return x.toString().compareTo(y.toString());
    }
}
```

Vor lauter Bäumen...

```
Set<StringBuilder> set = new TreeSet<StringBuilder>(new C());  
StringBuilder x = new StringBuilder("hello");  
StringBuilder y = new StringBuilder("world");  
set.add(x);  
set.add(y);
```

```
System.out.println(set.contains(y));  
y.append("!");  
System.out.println(set.contains(y));  
y.delete(0, 5);  
System.out.println(set.contains(y));
```

```
class C implements Comparator<StringBuilder>  
{  
    public int compare(StringBuilder x, StringBuilder y)  
    {  
        return ???  
    }  
}
```

☐ fff

☐ tff

☐ ttf

☒ ttt

☐ Compiler ☹

☐ Laufzeit ☹

Vor lauter Bäumen...

```
Set<StringBuilder> set = new TreeSet<StringBuilder>(new C());
StringBuilder x = new StringBuilder("hello");
StringBuilder y = new StringBuilder("world");
set.add(x);
set.add(y);
```

```
System.out.println(set.contains(y));
y.append("!");
System.out.println(set.contains(y));
y.delete(0, 5);
System.out.println(set.contains(y));
```

```
class C implements Comparator<StringBuilder>
{
    public int compare(StringBuilder x, StringBuilder y)
    {
        return System.identityHashCode(x)
            - System.identityHashCode(y);
    }
}
```

Gleichheit

Veränderliche Objekte eignen sich nicht als Elemente in sortierten und Hash-basierten Sammlungen, weil die Invarianten der Sammlung durch Änderungen von außen verletzt werden können.

Das Redefinieren von `equals` und `hashCode` ist in Objektklassen selten sinnvoll. Nach einer Veränderung können Objekte möglicherweise nicht mehr gefunden werden, obwohl sie noch enthalten sind.

Gleiche Objekte können nicht in dasselbe `HashSet` eingefügt werden:

```
for (int i = 0; i < 10; ++i)
{
    konten.add(new Konto(100));
}
```

Gedeckte Konten

```
List<Konto> konten = new ArrayList<Konto>();  
  
konten.add(new Konto( 1));  
konten.add(new Konto(-2));  
konten.add(new Konto(-3));  
konten.add(new Konto( 4));  
  
for (Konto konto: konten)  
{  
    if (konto.gibSaldo() < 0) konten.remove(konto);  
}  
  
System.out.println(konten.size());
```

Gedeckte Konten

```
List<Konto> konten = new ArrayList<Konto>();  
  
konten.add(new Konto( 1));  
konten.add(new Konto(-2));  
konten.add(new Konto(-3));  
konten.add(new Konto( 4));  
  
for (Konto konto: konten)  
{  
    if (konto.gibSaldo() < 0) konten.remove(konto);  
}  
  
System.out.println(konten.size());
```

java.util.ConcurrentModificationException

Gedeckte Konten

```
List<Konto> konten = new ArrayList<Konto>();  
  
konten.add(new Konto( 1));  
konten.add(new Konto(-2));  
konten.add(new Konto(-3));  
konten.add(new Konto( 4));  
  
for (int i = 0; i < 4; ++i)  
{  
    Konto konto = konten.get(i);  
    if (konto.gibSaldo() < 0) konten.remove(i);  
}  
  
System.out.println(konten.size());
```

Gedeckte Konten

```
List<Konto> konten = new ArrayList<Konto>();  
  
konten.add(new Konto( 1));  
konten.add(new Konto(-2));  
konten.add(new Konto(-3));  
konten.add(new Konto( 4));  
  
for (int i = 0; i < 4; ++i)  
{  
    Konto konto = konten.get(i);  
    if (konto.gibSaldo() < 0) konten.remove(i);  
}  
  
System.out.println(konten.size());
```

java.lang.IndexOutOfBoundsException: Index: 3, Size: 3

Gedeckte Konten

```
List<Konto> konten = new ArrayList<Konto>();  
  
konten.add(new Konto( 1));  
konten.add(new Konto(-2));  
konten.add(new Konto(-3));  
konten.add(new Konto( 4));  
  
for (int i = 0; i < konten.size(); ++i)  
{  
    Konto konto = konten.get(i);  
    if (konto.gibSaldo() < 0) konten.remove(i);  
}  
  
System.out.println(konten.size());
```

Gedeckte Konten

```
List<Konto> konten = new ArrayList<Konto>();  
  
konten.add(new Konto( 1));  
konten.add(new Konto(-2));  
konten.add(new Konto(-3));  
konten.add(new Konto( 4));  
  
for (int i = 0; i < konten.size(); ++i)  
{  
    Konto konto = konten.get(i);  
    if (konto.gibSaldo() < 0) konten.remove(i);  
}  
  
System.out.println(konten.size());
```

[1, -3, 4]

Gedeckte Konten

```
List<Konto> konten = new ArrayList<Konto>();

konten.add(new Konto( 1));
konten.add(new Konto(-2));
konten.add(new Konto(-3));
konten.add(new Konto( 4));

for (int i = 0; i < konten.size();)
{
    Konto konto = konten.get(i);
    if (konto.gibSaldo() < 0) konten.remove(i); else ++i;
}

System.out.println(konten.size());
```

Gedeckte Konten

```
List<Konto> konten = new ArrayList<Konto>();  
  
konten.add(new Konto( 1));  
konten.add(new Konto(-2));  
konten.add(new Konto(-3));  
konten.add(new Konto( 4));  
  
for (Iterator<Konto> it = konten.iterator(); it.hasNext();) {  
    Konto konto = it.next();  
    if (konto.gibSaldo() < 0) it.remove();  
}  
  
System.out.println(konten.size());
```

Elemente entfernen

Manuell über Sammlungen zu iterieren, um darin enthaltene Elemente zu entfernen, ist extrem fehleranfällig.

Die „fehlerfreien“ Lösungen besitzen quadratische Komplexität.

Eigentlich wollen wir sagen: „Entferne alle Elemente aus der Sammlung, die ein bestimmtes Prädikat erfüllen“.

Dieser Gedanke ist in Java nicht idiomatisch ausdrückbar.

Elemente filtern

Alternativ kann man die Liste unverändert lassen und die gedeckten Konten in eine neue Liste übernehmen:

```
List<Konto> gedeckte = new ArrayList<Konto>();  
for (Konto konto: konten)  
{  
    if (konto.gibSaldo() >= 0) gedeckte.add(konto);  
}  
System.out.println(gedeckte.size());
```

Diese Lösung ist simpel und besitzt lineare Komplexität.

Das Abstraktionsniveau dieser imperativen Lösung ist recht niedrig.

Google Collections API

```
List<Konto> gedeckte = Lists.newArrayList(  
    Iterables.filter(konten, new Predicate<Konto>()  
    {  
        public boolean apply(Konto k)  
        {  
            return k.gibSaldo() >= 0;  
        }  
    })  
);
```

Anonyme Funktionen in Java 7

```
List<Konto> gedeckte = Lists.newArrayList(  
    Iterables.filter(konten, new Predicate<Konto>()  
    {  
        public boolean apply(Konto k)  
        {  
            return k.gibSaldo() >= 0;  
        }  
    })  
);
```

```
List<Konto> gedeckte = Lists.newArrayList(  
    Iterables.filter(konten,  
        (Konto k) => k.gibSaldo() >= 0  
    )  
);
```

Blick über den Tellerrand

Scala

```
var gedeckte = konten.filter ( k => k.gibSaldo() >= 0 );
```

C#

```
var gedeckte = konten.FindAll ( k => k.gibSaldo() >= 0 );
```

F#

```
let gedeckte = List.filter (fun k -> k.gibSaldo >= 0) konten
```

Ruby

```
gedeckte = konten.reject { |k| k.gibSaldo < 0 }
```

Python

```
gedeckte = filter ( lambda k: k.gibSaldo() >= 0, konten )
```

Haskell

```
gedeckte = filter ( \k -> gibSaldo k >= 0 ) konten
```

Abstraktionen

Das Filtern von Elementen sollte nicht mit Schleifen programmiert werden, sondern als Algorithmus zur Verfügung stehen.

Die Parametrisierung von Algorithmen mit Funktionen ist deutlich einfacher, wenn die Sprache anonyme Funktionen unterstützt. Ansonsten wird der Quelltext durch Boilerplate-Code verunstaltet.

Quelltext auf einem höheren Abstraktionsniveau drückt die Absichten deutlicher aus und birgt weniger Fehlerpotential.