

# Föreläsning 12

## Generiska programenheter Designmönstret Iterator Samlingar

### Generiska programenheter

Från och med version 5.0 är det möjligt att skriva generiska programenheter i Java:

- generiska interface
- generiska klasser
- generiska metoder

Generiska programenheter innebär återanvändning av kod.

Generiska programenheter parametreras med en eller flera typer, så kallade *typparametrar*. Man kan se den generiska programenheten som en mall från vilken kompilatorn kan generera programenheter för specifika typer.

2

### Generiska klasser

Nedan visas hur en klass `Pair`, som används för att lagra ett namn och ett konto, har skrivits om till en generisk klass. Den generiska klassen kan användas för att lagra godtyckliga par av objekt.

```
public class Pair {  
    private final String name;  
    private final BankAccount account;  
    public Pair(String name, BankAccount account) {  
        this.name = name;  
        this.account = account;  
    }  
    public String getFirst() {  
        return name;  
    }  
    public BankAccount getSecond() {  
        return account;  
    }  
}
```

```
public class Pair<T, S> {  
    private final T first;  
    private final S second;  
    public Pair(T first, S second) {  
        this.first = first;  
        this.second = second;  
    }  
    public T getFirst() {  
        return first;  
    }  
    public S getSecond() {  
        return second;  
    }  
}
```

```
Pair<String, Integer> p = new Pair<String, Integer>("Hello", 12);  
Pair<Integer, Integer> p = new Pair<Integer, Integer>(34, 56);
```

3

### Generiska klasser

Typerna som handhas i en generisk klass måste namnges och anges i klassens parameterlista.

```
public class Pair<S, T>
```

Det är praxis att ge korta namn enligt:

- E elementtyp i en samling
- K nyckeltyp i en map
- V värdetyp i en map
- T generell typinformation
- S, U fler generella typer

4

## Generiska klasser

Typparametrar i en klassdeklaration är endast information till kompilatorn.

- typparametrarna används när kompilatorn kontrollerar programmet och sätter in vissa konverteringar (autoboxing/unboxing)
- när kompilatorn översätter klassdeklarationen till bytekod tas denna information bort.

Inuti den generiska klassdefinitionen kan de formella typparametrarna användas som vanliga referenstyper, förutom att:

- de får inte användas i klassens statiska metoder
- man får inte skapa arrayer av dem (men man kan deklarerat referenser till arrayer av deras typ).

5

## Generiska metoder

I en generisk klass får en statisk metod inte använda klassens typparametrar. Men det går dock att deklarerat generiska metoder, genom att parametriskera metoden med en eller flera typparametrar.

Obs!  
Typen måste finnas  
i signaturen

```
public class ArrayUtility {
    public static void print(String[] a) {
        for (String e : a)
            System.out.print(e + " ");
            System.out.println();
        }
    ...
}
```

```
public class ArrayUtility {
    public static <E> void print(E[] a) {
        for (E e : a)
            System.out.print(e + " ");
            System.out.println();
        }
    ...
}
```

```
Rectangle[] rectangles = . . . ;
String[] strings = . . . ;
ArrayUtility.print(rectangles);
ArrayUtility.print(strings);
```

6

## Begränsade typparametrar

Det är ibland nödvändigt att definiera vilka typer som man kan använda i en generisk klass eller i en generisk metod.

Betrakta en metod `min` som bestämmer det minsta värdet i ett generiskt fält. Hur kan man finna det minsta värdet när man inte vet vilken typ elementen i fältet har?

Ett sätt är att kräva att elementen tillhör en typ som implementerar interfacet `Comparable` (som är generiskt). I detta fall behöver vi således specificera typparametern ytterligare eftersom inte alla typer kan accepteras.

7

## Begränsade typparametrar

```
public static <E extends Comparable<E>> E min(E[] a) {
    E smallest = a[0];
    for (int i = 1; i < a.length; i++) {
        if (a[i].compareTo(smallest) < 0)
            smallest = a[i];
        }
    return smallest;
}
```

Metoden `min` ovan kan anropas med fält vars element är av en typ som implementerar `Comparable<E>`.

Ibland behöver man sätta flera begränsningar på en typparameter:

`<E extends Comparable<E> & Cloneable>`

Observera att här betyder det reserverade ordet **extends** ”extends eller implements”.

8

## Wildcards

Konceptet wildcards har införts för att beteckna ”vilken typ som helst”.

Det finns tre olika slag av wildcards:

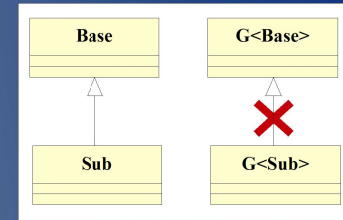
- wildcard med undre gräns    **? extends B**    B eller godtycklig subtyp till B
- wildcard med övre gräns    **? super B**    B eller godtycklig supertyp till B
- wildcard utan gräns    **?**    godtycklig typ

9

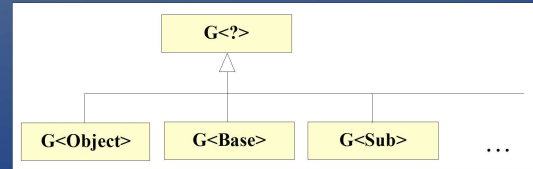
## Subtyper

Om Sub är en subklass till klassen Base och G är en generisk typ

- då är **inte** G<Sub> en subtyp till G<Base>



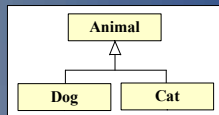
Det gäller att:



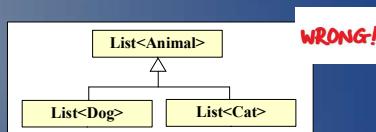
10

## Subtyper

Antag att:



Om det skulle gälla att:



Vad skulle hända?

I en lista av typen List<Animal> kan instanser både av typen Dog och Cat läggas in.

En lista av typen List<Animal> är utbytbar med en lista av typen List<Dog> eller typen List<Cat>.

Medför att en lista av typen List<Dog> kommer att kunna innehålla element av typen Cat, och en lista av typen List<Cat> kommer att kunna innehålla element av typen Dog!

11

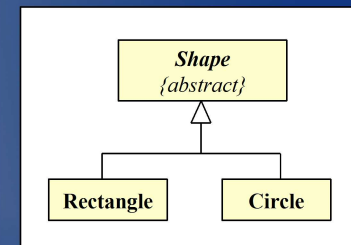
## Wildcards – godtycklig subtyp

Antag vi har följande klasshierarki:

```
public abstract class Shape {
    public abstract void draw();
}

public class Rectangle extends Shape {
    public void draw() { ... }
}

public class Circle extends Shape {
    public void draw() { ... }
}
```



12

## Wildcards – godtycklig subtyp

Metoden

```
public static void drawAll(LinkedList<Shape> shapes) {  
    for (Shape s: shapes)  
        s.draw();  
}
```

kan *inte* anropas med en parameter av typen

```
LinkedList<Circle> myList = new LinkedList<Circle>();  
ty LinkedList<Circle> är inte subklass till LinkedList<Shape>.
```

Däremot fungerar följande metod:

```
public static void drawAll(LinkedList<? extends Shape> shapes) {  
    for (Shape s: shapes)  
        s.draw();  
}
```

13

## Wildcards – godtycklig supertyp

Antag vi har följande klasshierarki:

```
public class BankAccount implements Comparable<BankAccount> {  
    ...  
}  
  
public class SavingAccount extends BankAccount {  
    ...  
}
```

Kan metoden

```
public static <E extends Comparable<E>> E min(E[] a) {  
    ...  
}
```

anropas med ett fält av typen `SavingAccount` enligt nedan?

```
SavingAccount[] sa = ...;  
...  
SavingAccount m = min(sa);
```

14

## Wildcards – godtycklig supertyp

Nej!

`SavingAccount` implementerar inte `Comparable<SavingAccount>` utan `Comparable<BankingAccount>` och `Comparable<SavingAccount>` är inte en subtyp till `Comparable<BankingAccount>`!

Metoden måste ha följande utseende:

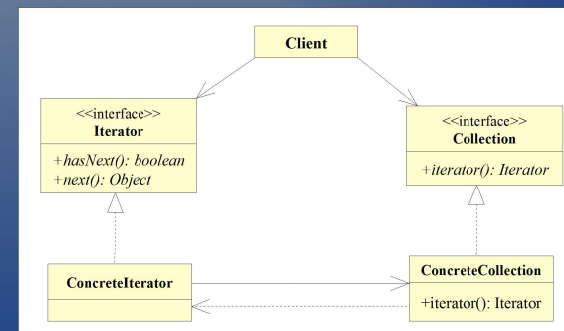
```
public static <E extends Comparable<? super E>> E min(E[] a) {  
    ...  
}
```

15

## Designmönstret Iterator

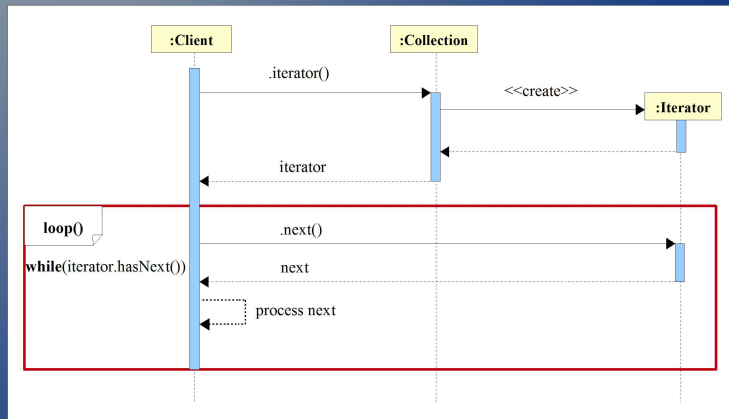
Med en iterator kan man förflytta sig igenom en samling av data, utan att behöva känna till detaljer om datans interna representation.

Iterator-mönstret är användbart eftersom det tillhandahåller ett definierat sätt att genomlöpa en uppsättning av dataelement, utan att exponera vad som händer inne i klassen.



16

## Designmönstret Iterator



17

## Gränssnitten Iterable<E> och Iterator<E>

Java tillhandahåller gränssnittet Iterable<E> och gränssnittet Iterator<E> :

```
public interface Iterable<E> {
    Iterator<E> iterator();
}
```

```
public interface Iterator<E> {
    boolean hasNext();
    <E> next();
    default void remove();
}
```

18

## Designmönstret Iterator

Antag att vi har en klass Deck, enligt nedan, för att avbilda en kortlek:

```
import java.util.ArrayList;
public class Deck {
    private ArrayList<Card> theDeck;
    public Deck () {
        theDeck = new ArrayList<Card>();
    }
    //more methods not shown here
}
```

Utöka klassen Deck med en anonym inre klass för att tillhandahålla en iterator för att genomlöpa korten som finns i kortleken. Implementera metoden remove genom att kasta UnsupportedOperationException.

19

## Designmönstret Iterator

```
import java.util.ArrayList;
import java.util.Iterator;
public class Deck implements Iterable<Card> {
    private ArrayList<Card> theDeck = new ArrayList<Card>();
    public Deck () {
        ...
    }

    public Iterator<Card> iterator() {
        return new Iterator() { // anonym inre klass
            private int pos = 0;
            public boolean hasNext() {
                return (pos < theDeck.size());
            }
            public Card next() {
                return theDeck.get(pos++);
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
    //other methods not shown here
}
```

20

## Samlingar – Java Collections Framework

När man programmerar har man ofta behov av att gruppera flera objekt av samma typ till en större enhet - en *samling*.

Java tillhandahåller ett antal samlingsklasser, *Java Collections Framework*, som finns i `java.util`.

Sedan Java 5 är dessa klasser generiska för att få typsäkerhet.

Java Collection Framework ger en enhetlig arkitektur för att representera och hantera samlingar.

Idén är att definiera ett litet antal generella gränssnitt som beskriver listor, mängder, köer och avbildningar (mapningar).

21

## Java Collections Framework

Java Collections Framework innehåller:

### Gränssnitt

- abstrakta datatyper representerande samlingar
- för att göra hanteringen av samlingar oberoende av implementeringsdetaljer

### Implementeringar

- konkreta implementeringar av samlingsgränssnitt
- återanvändbara datastrukturer

### Algoritmer

- användbara beräkningar (t.ex. sökning, sortering) i samlingar
- 'polymorfa' metoder: återanvändbar funktionalitet

### Infrastruktur

- andra gränssnitt (t.ex. `Iterator`, `Comparable`), stöder framtagningen av olika sorters samlingar

### Omslagsklasser

- ökar funktionaliteten mot andra implementeringar, t.ex. behöver man inte själv transformera mellan samlingar

22

## Fördelar

### Förenklar programmeringsarbetet

- ger användbara datastrukturer och algoritmer, du behöver inte skriva dem själv

### Effektiviserar arbetet

- genom att tillhandahålla implementeringar av hög kvalitet
- implementeringarna är utbytbara, smidigt att välja rätt implementering
- lätt att sätta ihop olika programkomponenter som skickar och tar emot datastrukturer i form av samlingar

### Operera mellan olika paket i API:n

- fastställer gemensamma typer för att utbyta samlingar

### Mindre besvär att begripa API:n

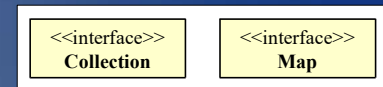
- systematiskt val av namn för olika samlingar, många metoder kan ärvas

### Mindre besvär att utforma nya API-samlingar

- mindre frestande att göra 'ad hoc'-lösningar, då det finns standardgränssnitt att tillgå

23

## Gränssnitt för samlingar



### Gränssnitt för samlingar

- definierar metoder för att hantera samlingarna
- fastställer typer för att överföra samlingar via metoder

Det finns två grundläggande gränssnitt:

`Collection<E>` handhar listor, mängder och köer.

`Map<K, V>` handhar avbildningar, d.v.s. element som är associerade med nyckelvärden.

(De råa varianterna `Collection` och `Map` finns fortfarande att tillgå men skall inte användas.)

24

## Gränssnitt Collection<E>

Gränssnittet Collection<E> handhar listor, mängder och köer.

De viktigaste subgränssnitten till Collection<E>:

- List<E> handhar en sekvens av objekt. Ett element kan förekomma flera gånger.
- Set<E> handhar en mängd av objekt. Ett element kan endast förekomma en gång i en mängd.
- SortedSet<E> handhar en ordnad (sorterad) mängd. Alla element måste vara jämförbara på 'storlek'.
- Queue<E> handhar en kö av element. Elementen i en kö har ett *first-in-first-out*-beteende.
- Deque<E> handhar en deque av element. I en deque (*double ended queue*) kan man stoppa in/ta bort element i båda ändar.

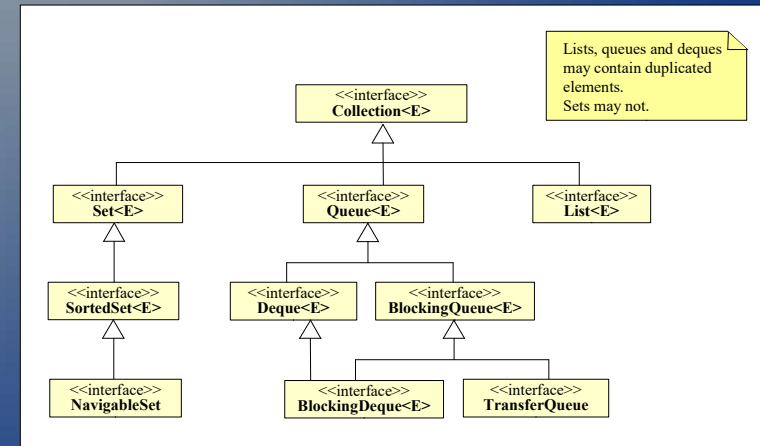
Varje klass K som implementerar Collection<E> har en konstruktor

K(Collection<? extends E> c)

därmed kan man lätt konvertera mellan olika samlingar.

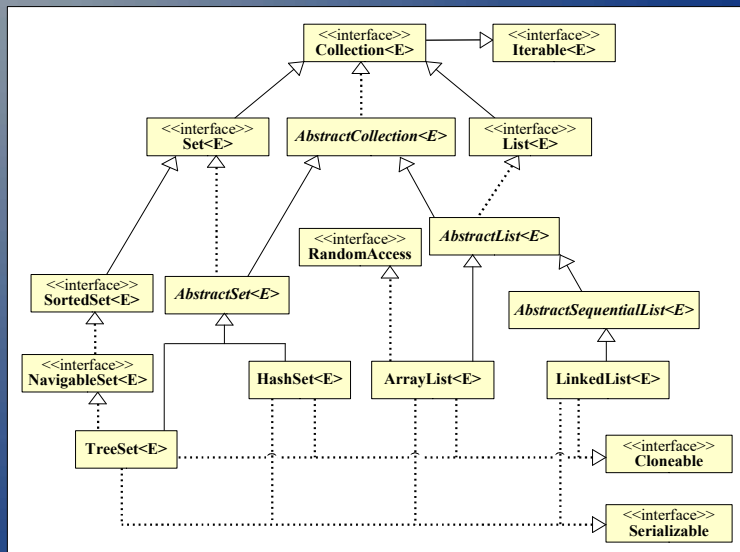
25

## Gränssnitt Collection<E>



26

## Några implementeringar av Collection<E>



27

## Gränssnittet Collection<E>

```

public interface Collection<E> implements Iterable<E> {
    // grundoperationer
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); // Valbar
    boolean remove(Object element); // Valbar
    Iterator<E> iterator();

    // mängdoperationer
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // Valbar
    boolean removeAll(Collection<?> c); // Valbar
    boolean retainAll(Collection<?> c); // Valbar
    void clear(); // Valbar

    // fältoperationer
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
    
```

28

## Valbara metoder

JDK erbjuder inte separata gränssnitt för alla varianter av samlingar, t.ex. samlingar som är icke-muterbara, är av en viss storlek eller endast tillåter att lägga till element.

I stället har varje gränssnitt modifierande operationer (mutator metoder) vilka anges som valbara (*optional*)

- implementeringar *kan* men behöver inte 'utföra' dessa operationer
- men fortfarande måste dessa metoder implementeras (som alltid)
- metoder som ej utför operationer implementeras med:  
{ **throw new** UnsupportedOperationException();}
- implementeringen måste dokumentera vilka detta gäller

29

## Collection<E>: basoperationer

Frågor om antal element i samlingen:

- int** size()                   returnerar antalet element i samlingen.
- boolean** isEmpty()        returnerar **true** om samlingen är tom, annars returneras **false**.

Lägga till eller ta bort ett element i samlingen:

- boolean** add(E e)       lägger in **e** i samlingen. Garanterar att **e** återfinns i samlingen efter operationen. Returnerar **true** om samlingen har förändrats av operationen, annars returneras **false**.
- boolean** remove(E e)   tar bort **e** ur samlingen. Returnerar **true** om samlingen har förändrats av operationen, annars returneras **false**.

30

## Collection<E>: basoperationer

Fråga om ett givet objekt finns med som element i samlingen:

- boolean** contains(Object o)   returnerar **true** om **O** finns i samlingen, annars returneras **false**.

Tillhandahålla en s.k. iterator för att kunna genomlöpa hela samlingen:

- Iterator<E> iterator()       returnerar en iterator för att genomlöpa samlingen.

31

## Collection<E>: mängdoperationer

Mängdoperationer utför operationer på hela samlingen:

- boolean** containsAll(Collection<?> c) – returnerar **true** om denna samling innehåller alla element i **C**, annars returneras **false**.
- boolean** addAll(Collection<? extends E> c) – lägger till alla elementen i **C** till denna samling. Returnerar **true** om samlingen har förändrats av operationen, annars returneras **false**.
- boolean** removeAll(Collection<?> c) – tar bort alla element i **C** från denna samling. Returnerar **true** om samlingen har förändrats av operationen, annars returneras **false**.
- boolean** retainAll(Collection<?> c) – tar bort alla element i denna samling utom de som finns i **C**. Returnerar **true** om samlingen har förändrats av operationen, annars returneras **false**.
- void** clear() – tar bort alla element ur denna samling.

32



## Collection<E>: fältoperationer

Används för att lägga alla elementen i en samling i ett fält

`Object[] toArray()` kopierar (referenserna till) elementen till ett fält med referenstypen `Object`.

### Exempel:

```
Collection<String> coll = new ArrayList<String>();  
...  
Object[] a = coll.toArray();
```

`<T> T[] toArray(T[] arr)` kopierar (referenserna till) elementen till ett nytt fält med referenstypen `T`.

### Exempel:

```
Collection<String> coll = new ArrayList<String>();  
...  
String[] a = coll.toArray(new String[0]);
```

33

## Gränssnittet Set<E>

- modellerar matematiska mängder
- en mängd är en samling element som *inte kan innehålla dubletter och där elementen inte har någon inbördes ordning*.

Definition av `Set<E>`:

```
public interface Set<E> extends Collection<E> {}
```

`Set<E>` innehåller *inga ytterligare metoder* förutom de som ärvt från `Collection<E>`!

- specifikationen av vissa operationer (dvs deras semantik) är annorlunda
- skillnaden ligger alltså i kommentarerna på API-sidan!

`add(E e)` lägger till `e` till mängden om `e` inte redan finns i mängden.

`addAll(Collection<? extends E> c)` lägger till alla element i samlingen `c` som inte redan finns i mängden.

34

## Klassen HashSet<E>

Klassen `HashSet<E>` implementerar `Set<E>` med hjälp av en underliggande hashtabell.

Klassen använder de lagrade objektens `hashCode`-metod för att lokalisera objekt i hashtabellen och objektens `equals`-metod för att jämföra om två objekt är lika.

Konstruktörer:

`public HashSet()` – skapar en tom mängd.

`public HashSet(Collection<? extends E> coll)` – skapar en mängd vars initiala innehåll är elementen i samlingen `coll`.

`public HashSet(int initialCapacity)` – skapar en tom mängd, där hashtabellen har specificerad initial storlek med en fyllnadsgrad av 0.75.

`public HashSet(int initialCapacity, float loadFactor)` – skapar en tom mängd, där hashtabellen har specificerad initial storlek och fyllnadsgrad.

35

## Exempel:

Skriv ett program som läser in ord via kommandoraden och skriver ut alla dubletter som påträffas, samt en lista av alla olika ord.

```
import java.util.*;  
public class FindDup {  
    public static void main(String[] args) {  
        Set<String> ss = new HashSet<String>();  
        for (String s : args)  
            if (!ss.add(s))  
                System.out.println("Detecting duplicate: " + s);  
        System.out.println(ss.size() + " different words: " + ss);  
    }  
}
```

Körningsexempel:

```
java FindDup jag kom jag såg jag segrade  
Detecting duplicate: jag  
Detecting duplicate: jag  
4 different words: [såg, kom, jag, segrade]
```

36

## for-each-satsen

För samlingar har vi den bekväma for-each-satsen, när vi skall gå igenom hela samlingen

```
for (elementType x : samling) //för alla element x som tillhör samling
    'block med x'
```

for-each-satsen kan användas på fält och alla klasser som implementerar gränssnittet `Iterable<E>`, vilket `Collection<E>` gör.

`Iterable<E>` kräver att metoden

```
public Iterator<E> iterator()
```

är definierad.

Notera: Det går endast att läsa av elementen med for-each-satsen. Det går inte att ändra i samlingen. En tilldelning till variabeln `x` ändrar endast den lokala variabeln och påverkar inte samlingen över huvud taget.

37

## Ett exempel till

Skriv ett program som läser in ord via kommandoraden och som dels skriver ut alla olika ord som påträffas, dels alla ord som förekommer som dubletter.

```
import java.util.*;
public class FindDups {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups = new HashSet<String>();
        for (String s : args)
            if (!uniques.add(s))
                dups.add(s);
        System.out.println("Unique words: ");
        Iterator<String> iu = uniques.iterator();
        while (iu.hasNext())
            System.out.println(iu.next());
        System.out.println("Duplicated words: ");
        for (String s : dups)
            System.out.println(s);
    }
}
```

### Körningsexempel:

```
java FindDups jag kom jag såg jag segrade
Unique words:
såg
kom
jag
segrade
Duplicated words:
jag
```

38

## Mängdoperationer i Set<E>

Mängdoperationerna är speciellt anpassade till `Set<E>`, då de utför de 'vanliga' mängdoperationerna: (delmängd, union, snitt och differens ( $\subseteq$ ,  $\cup$ ,  $\cap$  och  $\setminus$ )).

```
s1.containsAll(s2)    returnerar true om s2 ⊆ s1 (delmängd)
s1.addAll(s2)         gör om s1 till s1 ∪ s2 (union)
s1.retainAll(s2)     gör om s1 till s1 ∩ s2 (snitt)
s1.removeAll(s2)     gör om s1 till s1 \ s2 (differens)
```

39

## Mängdoperationer i Set<E>

### Exempel:

Skriv ett program som läser in ord via kommandoraden och som dels skriver ut alla olika ord som förekommer endast en gång, dels alla ord som förekommer som dubletter.

```
import java.util.*;
public class FindDups2 {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups = new HashSet<String>();
        for (String s : args)
            if (!uniques.add(s))
                dups.add(s);
        uniques.removeAll(dups);
        System.out.println("Single occurrence: " + uniques);
        System.out.println("Duplicated occurrence: " + dups);
    }
}
```

### Körningsexempel:

```
java FindDups2 jag kom jag såg jag segrade
Single occurrence: [såg, kom, segrade]
Duplicated occurrence: [jag]
```

40

## Ta bort element via iteratorn

### public void remove() throws IllegalStateException

- Tar bort från den underliggande samlingen det element som senast returnerades från iteratorn via ett anrop av `next()`.
- Metod `remove()` kan endast anropas en gång per anrop av `next()`.
- Beteendet för en iterator är odefinierat om den underliggande samlingen modifieras, medan iteratorn är aktiv, på något annat sätt än via anrop av `iterator`s `remove()`.
- Kastar `IllegalStateException` – om metoden `next()` ännu inte anropats eller om metoden `remove()` redan anropats, efter det senaste anropet av metoden `next()`.

41

## Ta bort element under iteration

Korrekt tillvägagångssätt:

```
//remove all strings longer than maxLen
public void removeLongStrings(Collection<String> coll, int maxLen) {
    Iterator<String> it = coll.iterator();
    while (it.hasNext()) {
        String str = it.next();
        if (str.length() > maxLen)
            it.remove();
    }
}
```

Felaktigt tillvägagångssätt:

```
//remove all strings longer than maxLen
public void removeLongStrings(Collection<String> coll, int maxLen) {
    Iterator<String> it = coll.iterator();
    while (it.hasNext()) {
        String str = it.next();
        if (str.length() > maxLen)
            coll.remove(str);
    }
}
```

Misstag!  
Beteendet hos iterator  
blir odefinierat!

42

## Gränssnittet SortedSet<E> extends Set<E>

Sorterade mängder *garanterar att objekten hela tiden är sorterade.*

Objekten som lagras i en sorterad mängd måste implementera gränssnittet `Comparable` (dvs. metoden `compareTo`).

Detta leder bland annat till att objekten löps igenom med iteratorn i en given ordning. Detta gäller inte för osorterade mängder.

`SortedSet` definierar också några extra metoder, bl.a.:

`E first()` – returnerar det 'minsta' elementet i mängden.

`E last()` – returnerar det 'största' elementet i mängden.

`SortedSet<E> subSet(E min, E max)` – returnerar en delmängd bestående av alla element i intervallet `[min ... max]`.

`Comparator<? super E> comparator()` - returnerar det comparator-objekt som används för att sortera mängden, null returneras om den naturliga ordningen används (dvs. `compareTo`-metoden)

43

## Gränssnittet NavigableSet<E>

Gränssnittet `NavigableSet` utökar gränssnittet `SortedSet` med ett antal metoder.

De viktigaste metoderna är:

`Iterator<E> descendingIterator()` – returnerar en iterator som genomlöper mängden i omvänd ordning

`NavigableSet<E> descendingSet()` – returnerar en mängd i vilken elementen är sorterade i avtagande ordning.

44

## Klassen TreeSet<E>

"Tree" refererar till den datastruktur som används i implementeringen (en typ av balanserat binärt sökträd).

Två viktiga konstruktörer:

TreeSet() - skapar en ny, tom mängd

TreeSet(Collection<? extends E> coll) - skapar en ny mängd vars initiala innehåll är elementen i samlingen coll.

Notera: Mängden som skapas blir sorterad även om samlingen coll inte är det. Vi kan alltså sortera via en konstruktor !

45

## Klassen TreeSet<E>

Exempel:

```
import java.util.*;
public class FindDup {
    public static void main(String[] args) {
        Set<String> ss = new TreeSet<String>();
        for (String s : args )
            if (!ss.add(s))
                System.out.println("detecting duplicate: " + s);
        System.out.println(ss.size() + "different words: " + ss);
    }
}
```

Körningsexempel:

```
java FindDups jag kom jag såg jag segrade
detecting duplicate: jag
detecting duplicate: jag
different words: [jag, kom, segrade, såg]
```

Samlingen  
är sorterad!

46

## Gränssnittet List<E>

En lista är en samling där elementen sparas vid en speciell *position*, i form av ett heltalsindex. *Elementen har en inbördes ordning i listan.* I likhet med fältindex är första positionen 0.

Åtkomst via positionen – hantering av elementen sker via den numeriska positionen i listan.

Sökning – kan söka efter ett specificerat objekt och returnera vid vilken position objektet påträffas.

En lista kan innehålla dubletter.

Har en mer avancerad iterator (*ListIterator*) där man kan utnyttja listans sekventiella karaktär.

47

## Gränssnittet List<E>

```
public interface List<E> extends Collection<E> {
    // Åtkomst via position
    E get(int index);
    E set(int index, E element); // Valbar
    void add(int index, E element); // Valbar
    E remove(int index); // Valbar
    boolean addAll(int index, Collection<? extends E> c); // Valbar

    // Sökning
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iterarorer
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Dellistor (kallas också vy av listan)
    List<E> subList(int fromIndex, int toIndex);
}
```

48

## Operationer i List<E>

**E** `get(int index)` – returnerar det element som ligger i position `index`

**E** `set(int index, E elem)` – ersätter elementet i position `index` med `elem`. Det ersatta elementet returneras.

**void** `add(int index, E elem)` – skjuter in `elem` på position `index`. Alla element efter position `index` flyttas ett steg åt höger.

**E** `remove(int index)` – tar bort och returnerar elementet på position `index`. Alla övriga element efter position `index` flyttas ett steg åt vänster.

**boolean** `addAll(int index, Collection<? extends E> c)` – skjuter in alla elementen i samlingen `c` vid position `index`. Alla element efter `index` flyttas åt höger.

49

## Operationer i List<E>

**int** `indexOf(Object obj)` – returnerar första positionen för objektet `obj`. Returnerar -1 om `obj` inte finns i samlingen.

**int** `lastIndexOf(Object obj)` – returnerar sista positionen för objektet `obj`. Returnerar -1 om `obj` inte finns i samlingen.

`ListIterator<E>` `listIterator()` – returnerar en `ListIterator` som börjar i position 0.

`ListIterator<E>` `listIterator(int index)` – returnerar en `ListIterator` som börjar i position `index`.

`List<E>` `subList(int from, int to)` – returnerar en `List` som omfattar elementen i position `from` till `to-1`.

**void** `add(E e)` – lägger till elementet `e` sist i listan

**void** `addAll(Collection<? extends E> c)` – lägger till `c` sist i listan

50

## Dellistor - vyer: Proxy-mönstret

Metoden

`List<E>` `subList(int fromIndex, int toIndex)`

returnerar en dellista av elementen som finns i positionerna `fromIndex` till `toIndex-1`.

- dellistan presenterar en vy av listan
- dellistor kan användas som vanliga listor
- förändringar i dellistan syns i originallistan
- eliminerar behovet av explicita intervalloperationer
- beteendet för en dellista blir odefinierat om element sätts in eller tas bort från originallistan på annat sätt än via dellistan

Radera ett intervall av element i listan `list`

`list.subList(from, to).clear();`

51

## Gränssnittet ListIterator<E>

Utökar gränssnittet `Iterator` med följande metoder:

**boolean** `hasPrevious()` – returnerar true om det finns ett föregående element, annars returneras false

**int** `nextIndex()` – returnerar positionen för nästa element, ger antalet element om det inte finns något nästa element

**E** `previous()` – returnerar föregående element och flyttar iteratoren ett steg bakåt.

**int** `previousIndex()` – returnerar positionen för föregående element, finns inget föregående element returneras -1

**void** `add(E e)` – lägger in `e` omedelbart före det element som skulle returneras med `next` och omedelbart efter det element som skulle returneras med `previous` (valbar metod)

**void** `set(E e)` – ersätter det element som senast returnerades med `next` eller `previous` med `e` (valbar metod)

52

## Klassen ArrayList<E>

- sparar internt elementen i ett fält
- skapar ett nytt fält (50 % större) om antalet element blir för många
- snabb åtkomst av elementen
- långsamma insättningar (skiftande och ev göra nytt fält).

53

## Klassen ArrayList<E>

Utökar List med konstruktörer och fler operationer:

**public** ArrayList() – skapar en tom ArrayList<E>, med initial storlek 10 på det interna fältet.

**public** ArrayList(Collection<? **extends** E> c) – skapar ArrayList<E> innehållande elementen i c, elementen ligger i den ordning som ges av c.iterator(). Storleken av det interna fältet blir 110% av c.size().

**public** ArrayList(int initialCapacity) – skapar en tom ArrayList<E>, med initial storlek initialCapacity på det interna fältet.

**void** ensureCapacity(int minCapacity) – sätter storleken på det interna fältet till att åtminstone rymma minCapacity element.

**void** trimToSize() – anpassar storleken på det interna fältet till aktuellt antal element i listan.

54

## Klassen LinkedList<E>

- sparar elementen internt i en dubbellänkad struktur
- långsamma åtkomster
- snabba insättningar

Utökar List med konstruktörer och fler operationer:

**public** LinkedList() – skapar en tom LinkedList<E>.

**public** LinkedList(Collection<? **extends** E> c) – skapar en LinkedList<E> innehållande elementen i c. Elementen ligger i den ordning som ges av c.iterator().

**void** addFirst(E e) – lägger in e först i listan.

**void** addLast(E e) – lägger in e sist i listan.

E gerFirst() – returnerar första elementet i listan.

E getLast() – returnerar sista elementet i listan.

E removeFirst() – tar bort och returnerar första elementet i listan.

E removeLast() – tar bort och returnerar sista elementet i listan.

55

## Genomlöpningar av List<E>

Genomlöpning av en lista framifrån:

```
List<String> listA = new LinkedList<SomeType>();
...
ListIterator<SomeType> itForward = listA.listIterator();
while(itForward.hasNext()) {
    SomeType elem = itForward.next();
    ...
}
```

Genomlöpning av en lista bakifrån:

```
List<SomeType> listB = new ArrayList<SomeType>();
...
ListIterator<SomeType> itBack = listB.listIterator(listB.size());
while(itBack.hasPrevious()) {
    SomeType elem = itBack.previous();
    ...
}
```

Startposition

56

## Avbildningar: Map<K,V>

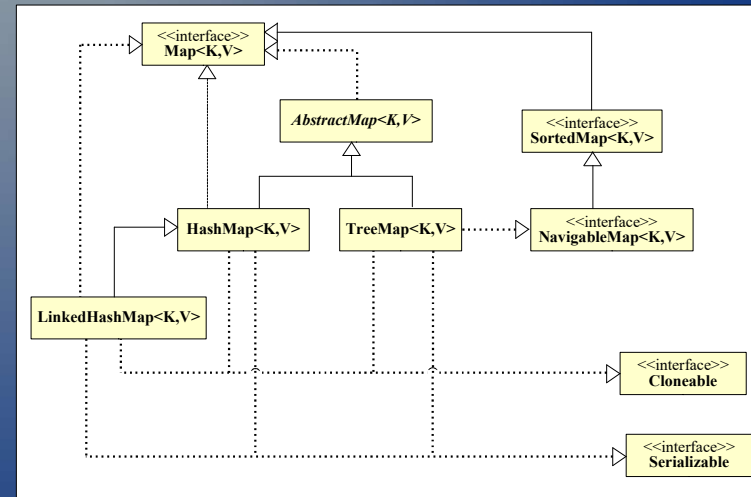
När man har en söknnyckel för att komma åt information använder man sig av avbildningstabeller eller *maps*.

- i en map lagras par av nycklar och värden, key/value pairs
- i en map kan man slå upp värdet som är bundet till en viss nyckel
- en map kan inte ha dubletter av nyckelvärden
- varje nyckel är bundet till ett och endast ett värde

Avbildningar är subtyper till gränssnittet Map<K,V> (inte till Collection<E>, semantiken är alldeles för olik)

57

## Några implementeringar av Map<K,V>



58

## Gränssnittet Map<K,V>

```
public interface Map<K,V> {
    // basoperationer
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // samlingsvyer
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K,V>> entrySet();

    // mängdoperationer
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
}

// Gränssnitt för elementen i entrySet
static interface Entry<K,V> {
    K getKey();
    V getValue();
    V setValue(V value);
}
```



59

## Gränssnittet Map<K,V>

Några metoder i Map<K,V>:

**V put(K key, V value)** – lägger in avbildningen i avbildningstabellen. Returnerar det föregående värdet som associerades med *key* eller **null** om inget sådant värde finns.

**void putAll(Map<? extends K, ? extends V> m)** – lägger in alla avbildningar som finns i avbildningstabellen *m* i aktuell avbildningstabell.

**V remove(Object key)** – tar bort avbildningen för nyckeln *key*. Returnerar det värde som associerades med *key* eller **null** om inget sådant värde finns.

**V get(Object key)** – returnerar värdet som har nyckeln *key*, eller **null** om *key* inte finns i avbildningstabellen.

**Set<K> keySet()** – returnerar en **Set** med alla nycklar.

**Collection<V> values()** – returnerar en **Collection** med alla värden.

**Set<Map.Entry<K,V>> entrySet()** – returnerar en **Set** med alla avbildningar, där avbildningarna är av klassen **Map.Entry**.

60

## Klassen HashMap<K,V>

Klassen HashMap<K,V> implementerar gränssnittet Map<K,V> med hjälp av en underliggande hashtabell.

Används då snabba sökningar är det högst prioriterade.

Konstruktörer:

**public** HashMap() – skapar en tom HashMap<K,V>.

**public** HashMap(Map<? extends K,? extends V> m) – skapar en tom HashMap<K,V> med samma innehåll som m.

**public** HashMap(int initialCapacity) – skapar en tom HashMap<K,V>, där hashtabellen har specificerad initial storlek med en fyllnadsgrad av 0.75.

**public** HashMap(int initialCapacity, float loadFactor) – skapar en tom HashMap<K,V>, där hashtabellen har specificerad initial storlek och fyllnadsgrad.

61

## Gränssnittet SortedMap<K,V>

Används för att skapa sorterade avbildningar.

Lägger till bl.a. följande metoder:

**K** firstKey() – returnerar den minsta söknyckeln

**K** lastKey() – returnerar den största söknyckeln

SortedMap<K,V> headMap(toKey) – returnerar en SortedMap<K,V> med alla avbildningar där söknyckeln är mindre än toKey.

SortedMap<K,V> tailMap(fromKey) – returnerar en SortedMap<K,V> med alla avbildningar där söknyckeln är större än fromKey.

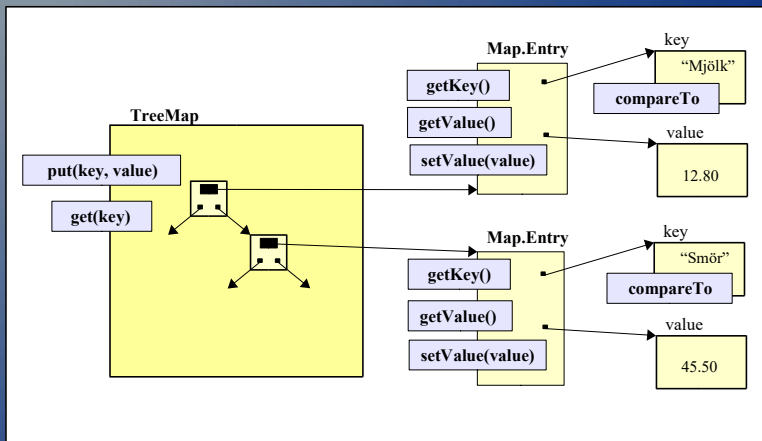
SortedMap<K,V> subMap(fromKey, toKey) – returnerar en SortedMap<K,V> med alla avbildningar där söknyckeln är större än fromKey och mindre än toKey.

Klassen TreeMap<K,V> implementerar SortedMap<K,V>.

TreeMap använder sig internt av ett balanserat binärt sökträd.

62

## Strukturen i TreeMap<K,V>



63

## Iterera över en Map<K,V>

Map-klasser har ingen iterator. Istället finns det tre metoder som returnerar nya datasamlingar som man kan iterera över (designmönstret Proxy).

Set<K> keySet() – som returnerar en Set med alla nyckel-objekt

Collection<V> values() – som returnerar en Collection med alla värde-objekt

Set<Map.Entry<K,V>> entrySet() – som returnerar en Set med alla Map.Entry-objekt

```
TreeMap<String, Integer> map = new TreeMap<String, Integer>();
...
Set<Map.Entry<String, Integer>> entries = map.entrySet();
Iterator<Map.Entry<String, Integer>> iter = entries.iterator();
while (iter.hasNext()) {
    Map.Entry<String, Integer> entry = iter.next();
    String key = entry.getKey();
    String value = entry.getValue();
    System.out.println(key + " is the number of " + value);
}
```

64



## Avslutande exempel

### Uppgift:

Skriv ett program som läser en textfil och skriver ut alla ord som ingår i texten i alfabetisk ordning samt på vilka rader i filen varje ord förekommer.

Radnumren på raderna där ett ord förekommer skall skrivas ut i växande ordning. Ingen skillnad skall göras mellan stora och små bokstäver.

Namnet på filen läses från kommandoraden.

65

## Avslutande exempel

### Lösning:

Vi har ett antal ord där varje ord associeras med en uppsättning radnummer. Således kan vi använda en `Map<K,V>`. Vi vet att orden skall skrivas ut i alfabetisk ordning varför valet faller på `TreeMap<K,V>`.

Våra nycklar är orden, d.v.s. `K` är `String`. Men hur sparar vi radnumren?

Med hjälp av en `Container`! Eftersom radnumren skall skrivas ut i växande ordning faller valet på `SortedSet<E>`.

Ett radnummer är en `int` men primitiva datatyper kan inte lagras i en generisk klass varför vi får `SortedSet<Integer>`.

Vår `Map` får alltså utseendet

```
TreeMap<String, SortedSet<Integer>>
```

För att inte göra någon skillnad mellan stora och små bokstäver använder vi oss av ett `Collator`-objekt.

66

## Det kompletta programmet

```
import java.util.*;
import java.io.*;
import java.text.*;
public class TextAnalys {
    public static void main(String[] arg) throws FileNotFoundException {
        Locale.setDefault(new Locale("sv", "sw"));
        Collator co = Collator.getInstance();
        co.setStrength(Collator.PRIMARY);
        TreeMap<String, SortedSet<Integer>> map =
            new TreeMap<String, SortedSet<Integer>>(co);
        Scanner lineScanner = new Scanner(new File(arg[0]));
        int lineNr = 0;
        while (lineScanner.hasNextLine()) {
            lineNr++;
            String buffer = lineScanner.nextLine();
            Scanner wordScanner = new Scanner(buffer);
```

67

## Det kompletta programmet

```
        while(wordScanner.hasNext()) {
            String word = wordScanner.next();
            if (map.containsKey(word)){
                SortedSet<Integer> li = map.get(word);
                li.add(lineNr);
                map.remove(word);
                map.put(word, li);
            }
            else {
                SortedSet<Integer> li = new TreeSet<Integer>();
                li.add(lineNr);
                map.put(word, li);
            }
        }
        for (Map.Entry<String, SortedSet<Integer>> e : map.entrySet())
            System.out.println(e.getKey() + ": " + e.getValue());
    }
}
```

68

## Klassen Collections

Klassen Collections innehåller en uppsättning statiska metoder som opererar på eller returnerar samlingar:

- 'polymorfa' algoritmer, som opererar på samlingar
- trådsäkra wrappers
- icke-muterbara wrappers
- lite annat smått och gott

<<utility>> Collections
+binarySearch(List<? extends Comparable<? super T>>, T): T +copy(List<? super T>, List<? super T>): void +max(Collection<? extends T>): T +min(Collection<? extends T>): T +reverse(List): void +shuffle(List): void +singletonList(T): List<T> +singletonMap(T): Map<T> +sort(List): void +sort(List, Comparator): void +unmodifiableSet(Set<? extends T> s): Set<T> +unmodifiableList(List<? extends T> l): List<T> +synchronizedSet(Set<T> s): Set<T> +synchronizedList(List<T> l): List<T> ...

69

## Icke-muterbara wrappers

Exempel på designmönstret Decorator.

Förhindra alla operationer som förändrar en samling (genom att kasta UnsupportedOperationException).

Har två användningsområden

- göra en existerande samling icke-muterbar

```
List<String> list = new ArrayList<String>();  
... //lägg in element i listan  
list = Collections.unmodifiableList(list);
```

- endast tillåta read-only access för klienter, men själv ha full kontroll (designmönstret Proxy)

```
List<String> list = new ArrayList<String>();  
List<String> clientList = Collections.unmodifiableList(list);
```

70

## Klassen Arrays

Det finns också en klass Arrays som innehåller en uppsättning statiska metoder för bl.a. sökning och sortering på fält.

<<utility>> Arrays
+asList(E[]): List<E> +sort(int[]): void +sort(int[], int, int): void +sort(double[]): void +sort(double[], int, int) +sort(Object[]): void +sort(Object[], Comparator): void +sort(Object[], int, int): void +sort(Object[], int, int, Comparator): void +binarySearch(int[], int): void +binarySearch(int[], int, int, int): void +copyOf(int[]): int[] +copyOf(int[], int, int): void +equals(int[], int[]): boolean +fill(int[], int): void ...

71