Föreläsning 4

- Repetition
  - Interface
  - Generiska klasser och metoder
  - Collection - Samling

- Containerklasser
  - Collections
    - List
    - Set
    - Queue
  - Map
    - HashMap
    - TreeMap

Repetition – interface, generik, samling

- Varför använda interface?
  - Ett interface är en nivå av abstraktion där egenskaper av en klass specificeras
  - Rättare sagt, används ett interface för att specificera VAD en klass MÅSTE göra men inte HUR
  - En klass om implementerar ett ”interface” måste skriva implementation av alla metoder som är definierade i interfacet.
  - Garant för att där finns vissa metoder implementerade.
- Generiska klasser och metoder
  - Skriv klasser och metoder som är generella för olika typer av objekt och kan användas i många olika sammanhang.
- Java Collections Framework
  - En Collection är en samling av objekt ofta av samma typ eller olika typer.
  - Java Collections Framework är en hierarki av interface, abstrakta klasser och konkreta klasser för samlingar av objekt.

Collection Framwork

Collections Framework består av:

- Ett antal interfaces: Collection, Set, List, Queue, Map, SortedList, etc.
- Implementations: ArrayList, HashSet, TreeSet, LinkedList, HashMap, etc.
- Wrapper implementions: Utöka/höja/förbättra andra samlingarnas funktionaliteter. Vector, Enumset, Hashtable och en massa andra speciella typer.
- Concurrent implementations: ConcurrentHashMap
- Array utilities: Arrays containing static methods to search, sort, compare copy, resize, etc.

Generiska klasser

- En generisk klass gör att man kan återanvända sin kod.

I denna klassen Pair kan man endast lagra en String och ett BankAccount

```
Pair p = new Pair(name, BankAccount);
```

```
class Pair {
    private  String id;
    private  Person person;

    public Pair(String id, Person person){
        this.id = id;
        this.person = person;
    }//constructor

    public String getID() {
        return id;
    }//getFirst
    public String toString() {
        return id + " " +person.toString();
    }
}//Pair
```

I den generiska klassen Pair kan man välja vad som ska lagras efter behov.

```
PairGeneric<T,S r<String, Integer>= new Pair <String, Integer>(”Douglas”,42);
PairGeneric<T,S <Boolean, Char>= new Pair<Boolean, Char>(True, ’W’)
```

```
public class PairGeneric<T,S>{
    private  T first;
    private  S second;

    public PairGeneric(T first, S second) {
        this.first = first;
        this.second = second;
    }//constructor

    public T getFirst() {
        return first;
    }//getFirst

    public S getSecond() {
        return second;
    }//getSecond
    //S måste ha en toString metod
    public String toString(){
        return first.toString() + " " + second.toString();
    }
}
```

Samling - Containerklass

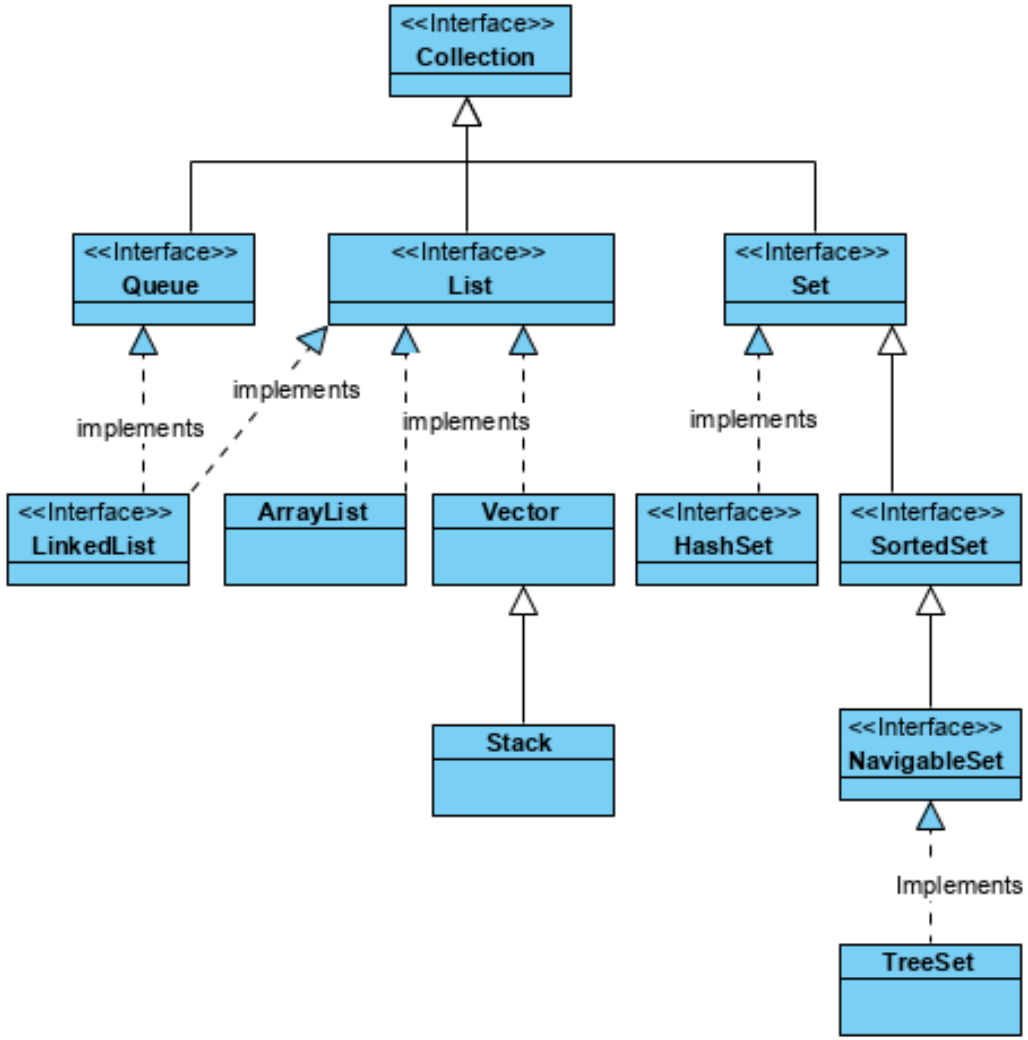
- Är en klass som på något sätt håller reda på flera objekt.

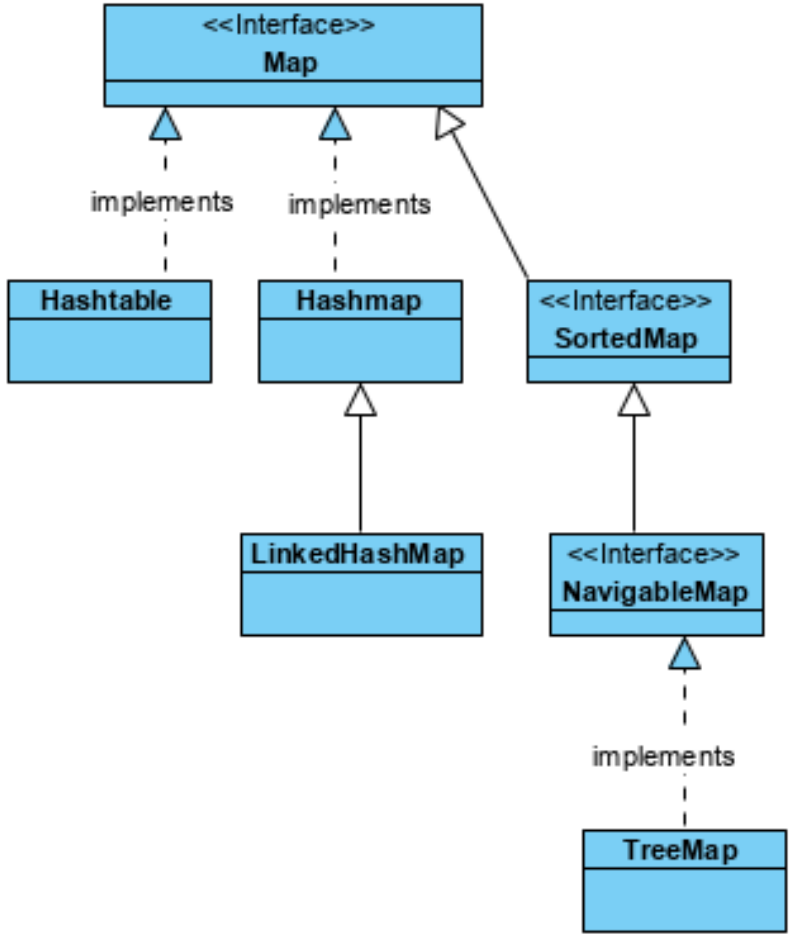
- Collection och Map

Exempel på några olika containerklasser

- Queue
- ArrayList
- HashMap
- JFrame

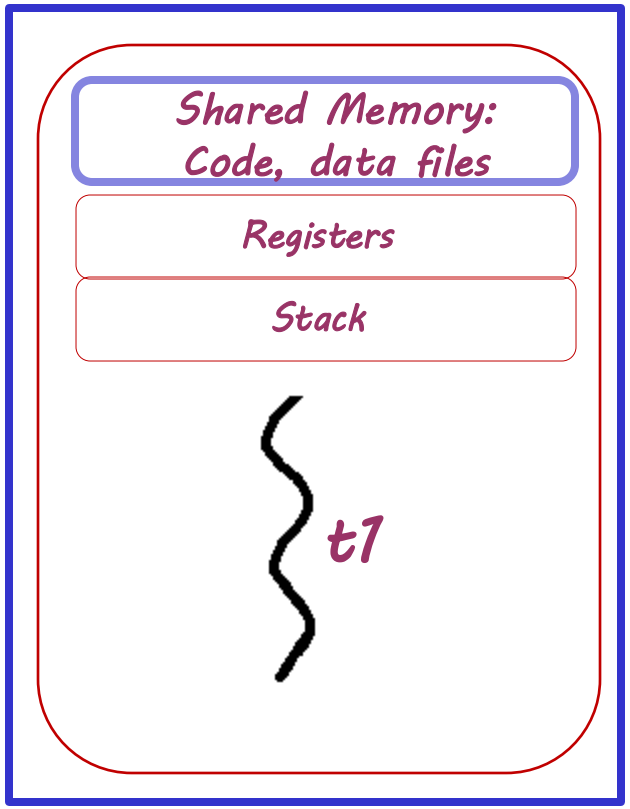
Java Collections Framework





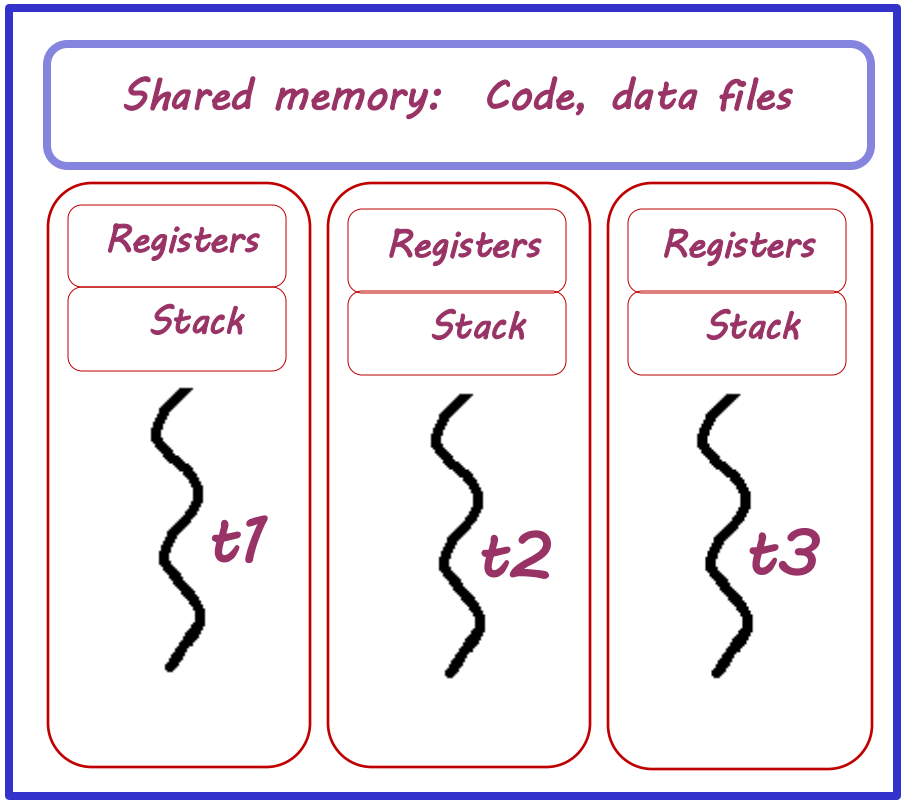
En kort introduktion till tråd och synkronisering

- När en applikation börjar köras blir den en process.
- Varje process har minst en tråd som kör processen.
  - Ett java programs main method körs av en tråd.
- Tråden brukar kallas för huvudtråden (Main Thread).
- Java har klassen Thread som har allt för att skapa nya trådar.



Vad är en tråd och synkronisering?

- En programmerare kan skapa flera trådar i Java för att göra flera saker samtidigt. Huvudsyfte är:
  - Bättre prestanda (snabbare program)
  - Bättre mottaglihet (bättre responsiveness).
- Trådar kan köra metoder i en klass och det kan hända att en metod körs av flera trådar samtidigt. Då kan en tråd ändra t.ex. värdet av en instansvariabel utan att andra trådar vet om det.
- Därför måste trådarna ändra resurserna (t ex instansvaribler) på ett kontrolllerat sätt. Detta kallas **synchronization**.




```
private int count = 0;
public synchronized void incrementValue()
{
    count = count++;
}
```

Synchronization exempel

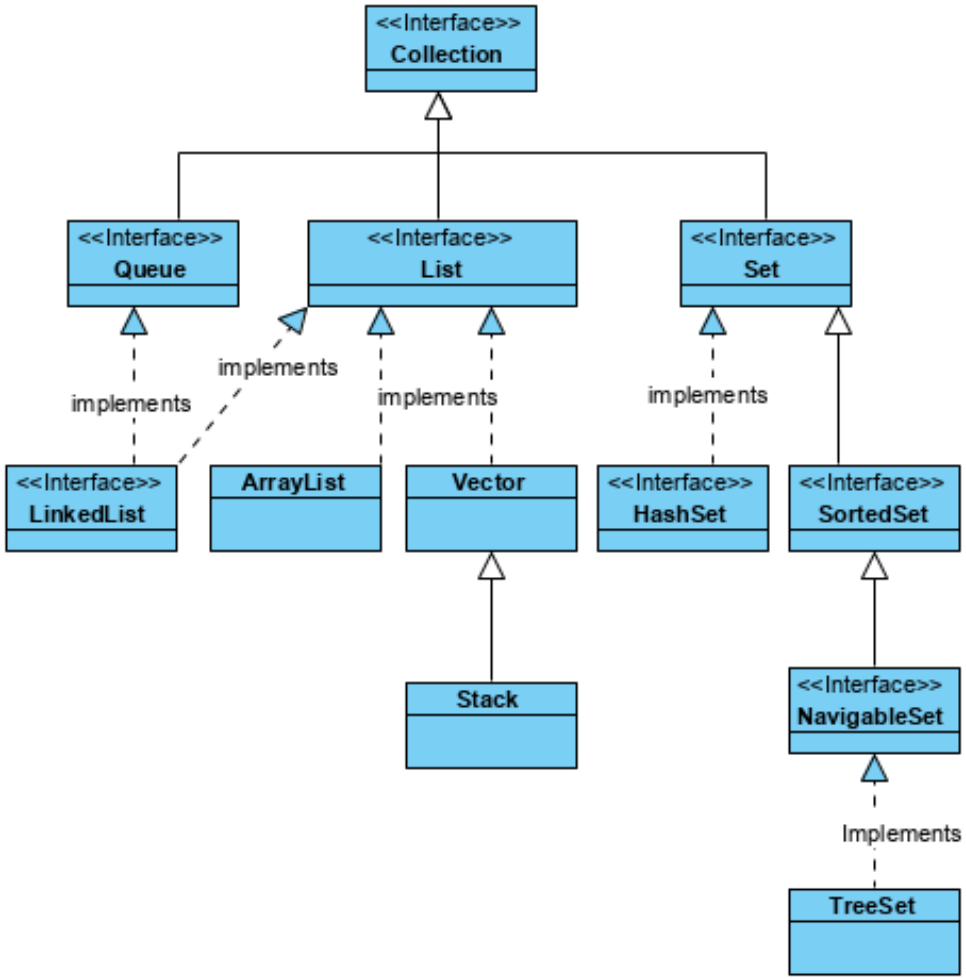
```
private int count = 0;
public void incrementValue()
{
    count = count++;
}
```

```
//Vad händer när metoden körs.
   // 1. Hämta count från minnet (Ladda)
   // 2. Beräkna count +1
   // 3. Spara count till minnet

//Ett scenario:  count = 0
   // T1 hinner med 1 och 2  --> count = 1
   // T2 kommer in och gör 1 --> count = 0
   // T1 kommer in och gör 3 --> count = 1
   // T2 kommer in och gör 2 och 3 count = 1
        //fel resultat! Rätt: count = 2
```

```
// Syncronisering - låt en tråd köra metoden
// helt och hållet innan nästa släpps!
// Metoden blir då Trådsäker
public synchronized void incrementValue1()
{
    count = count++;
}
```

Collections

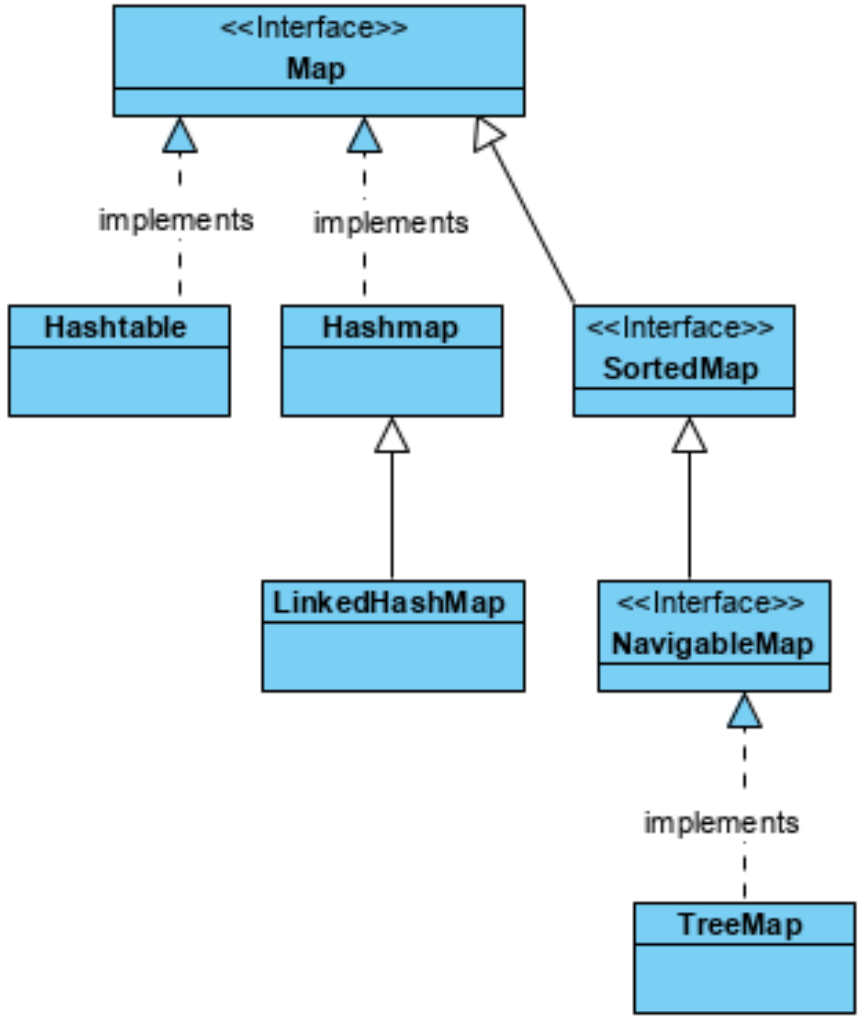
- ***List***
  - en ordnad sekvens av element. Ett element kan förekomma flera gånger i listan. Varje element kan nås med ett index.
- ***Set***
  - en mängd där inga element förekommer mer än en gång.
- ***Queue***
  - en kö innehåller en sekvens av element. I kön lagras och hämtas element normalt enligt FIFO-principen.



Map

Allt som sparas i en Map-klass har två delar,
en unik nyckel, **`key`**, och värde, **`value`**

- ***Hashtable***
  - Är synkroniserad, används när flera trådar ska nå samma objekt, tillåter inga `null`-nyckel eller värde.
- ***HashMap***
  - Är inte synkroniserad. Ler mer avancerad än Hashtable.
- ***SortedMap***
  - Håller nycklarna sorterade.



Collection – ett interface

Collection är ett interface vilket definierar grundläggande funktionalitet i ett antal klasser i vilka man kan lagra godtyckliga objekt, sk *containerklasser / objektsamlingar*.

Method Summary

boolean	**add**(E o) Ensures that this collection contains the specified element.
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection.
void	**clear**() Removes all of the elements from this collection.
boolean	**contains**(Object o) Returns true if this collection contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this collection contains all of the elements in the specified collection.
boolean	**equals**(Object o) Compares the specified object with this collection for equality.
int	**hashCode**() Returns the hash code value for this collection.
boolean	**isEmpty**() Returns true if this collection contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present.
boolean	**removeAll**(Collection<?> c) Removes all this collection's elements that are also contained in the specified collection.
boolean	**retainAll**(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection.
int	**size**() Returns the number of elements in this collection.
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Interfacet List<E>

Gränssnittet **List<E>** innehåller,
förutom metoderna i Collection, bl.a. metoderna:

Method Summary	
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	**remove**(int index) Removes the element at the specified position in this list
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

Klasser som implementerar **List**
(och därmed Collection) är bl.a.

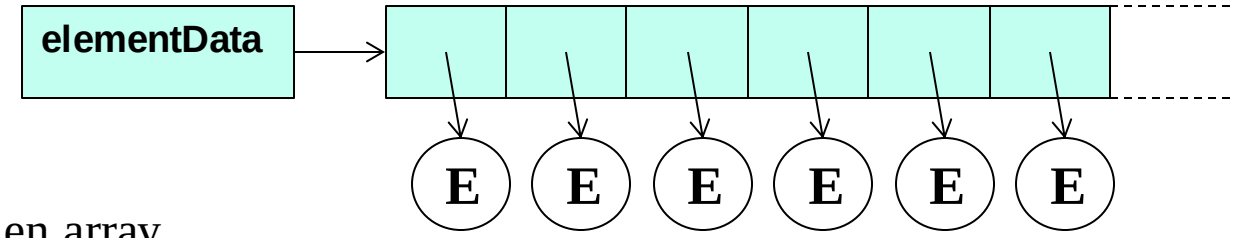
```
public class ArrayList<E> implements List<E> {…}

public class LinkedList<E> implements List <E> {…}

public class Vector<E> implements List<E> {…}.
```

Den underliggande strukturen, *datastrukturen*, för att lagra elementen är dold för användaren. Klassernas namn ger dock information om datastrukturen som används.

List implementation: `ArrayList<E>`

- I klassen ***ArrayList<E>*** finner man instansvariabeln:

```
private transient E[] elementData;
```



- Det innebär att listan implementeras med en array, vilket klassens namn antyder. Implementeringen i array innebär bl.a. att det går väldigt snabbt att komma åt element med hjälp av index.

- Klassen ***Vector<E>*** har en liknande implementation som *ArrayList* och likvärdiga egenskaper. Numera används *ArrayList* men *Vector* kan dyka upp i gammal kod.
    - Huvudskillnaden är att Vector är synchronized men ArrayList är inte det.

Constructor Summary

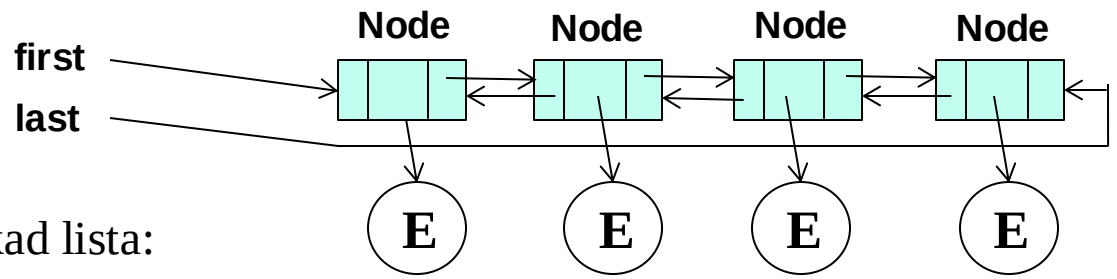
`ArrayList()`	Constructs an empty list with an initial capacity of ten.
`ArrayList(Collection<? extends E> c)`	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
`ArrayList(int initialCapacity)`	Constructs an empty list with the specified initial capacity.

ArrayListEx.java

List implementation: `LinkedList<E>`

- I klassen ***LinkedList*** finner man instansvariablerna:

```
transient Node<E> first;
transient Node<E> last;
```



- Listan är implementerad som en dubbellänkad lista:
- Första och sista elementet i listan hanteras snabbt. Resterande element måste man stega sig fram till.
- *Linkedlist* implementerar dessutom metoder för att hantera Stack respektive Queue.

LinkedListEx.java **LinkedStack.java** **UseLinkedStack.java**

Collections – metoder för listor

I klassen *Collections* finns ett antal klassmetoder att använda tillsammans med List-implementationer, t.ex:

`public static void sort(List<E> l)`
Listan ordnas. Elementen i listan måste implementera Comparable.

`public static void sort(List<E> l, Comparator<E> c)`
Elementen i listan ordnas med Comparator-implementeringen.

`public static int binarySearch(List<E> l, E element)`
Effektiv sökning i lista ordnad
med elementens Comparable-implementering.

`public static int binarySearch(List<E> i, Comparator<E> c, E element)`
Effektiv sökning i lista ordnad med Comparator-implementeringen

`public static void reverse(List<E> l)`
Elementens ordning blir omvänd

`public static void rotate(List<E> l, int n)`
Elementen flyttas n steg åt höger (n<0 åt vänster)

`public static void shuffle(List<E> l)`
Elementen i listan blandas slumpmässigt

`public static void swap(List<E> l, int i, int j)`
Byter plats på elementen i positionerna i och j

Collections – att sortera

Implementera `Comparable<Person>`

Men innan en del av metoderna kan användas (sort, binarySearch) måste klassen *Person* implementera ***`Comparable<Person>`***.

Hur ser en implementering ut,
vilken ordnar *Person*-objekt växande efter id ?

```
public class Person implements Comparable<Person> {
    :
    public int compareTo( Person p ) {

        return id.compareTo(p.id);

    //  p.id.compareTo(id) ordnar avtagande
    }
}
```

Skriv en Comparator

Ett alternativ är att skriva en klass vilken implementerar *Comparator<Person>*.

Hur ser en klass ut, vilken implementerar *Comparator* på så sätt att *Person*-objekt ordnas avtagande efter id?

```
public class IdDesc implements Comparator<Person> {

    public int compare( Person p1, Person p2 ) {

        return p2.getId().compareTo( p1.getId() );
    }
}
```

List1.java **ListSpeed.java**

Queue

Interfacet `Queue<E>` innehåller 2 uppsättningar av tre metoder:

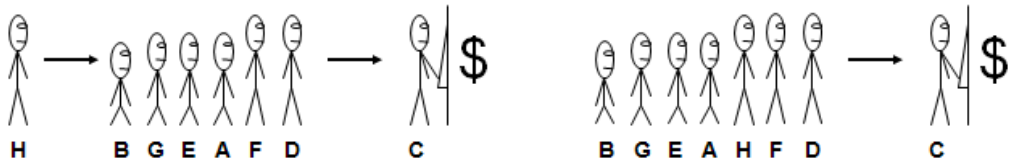
Queue Interface Structure

Type of Operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Uppsättningen till vänster kastar ett undantag om ett fel uppstår medan uppsättningen till höger returnerar speciella värden.

Klassen `LinkedList` implementerar `Queue<E>`

Klassen ***PriorityQueue*** implementerar **Queue<E>** på så sätt att element placeras i kön efter *prioritet*.



Element med samma prioritet bildar en normal kö i prioritetskön.

Prioriteten ges av elementens *Comparable*-implementeringen eller av en separat klass som implementerar *Comparator*.

QueueEx.java

Map-avbildningar

I en Map bildar en nyckel (key), och ett värde (value) ett par `<key,value>`.

Med hjälp av nyckeln placeras värdet i objektsamlingen och med hjälp av nyckeln söker man efter värdet.

Klasserna `TreeMap<K,V>` och `HashMap<K,V>` implementerar gränssnittet `Map<K,V>`.

Method Summary

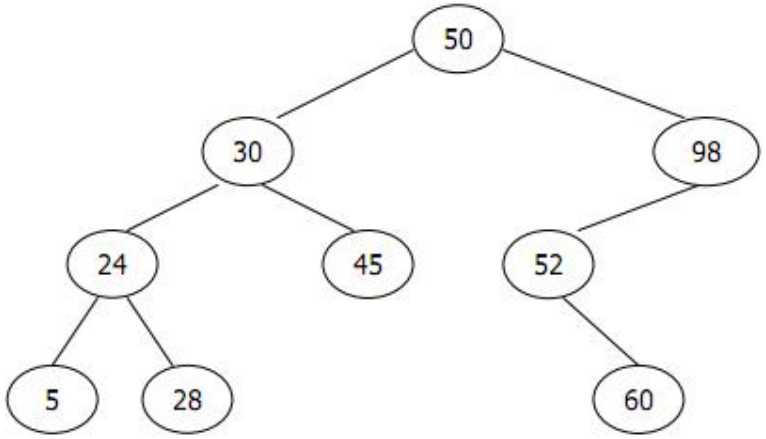
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
V	**get**(Object key) Returns the value to which this map maps the specified key.
Set<K>	**keySet**() Returns a set view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map.
void	**putAll**(Map<? extends K,? extends V> t) Copies all of the mappings from the specified map to this map.
V	**remove**(Object key) Removes the mapping for this key from this map if it is present.
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a collection view of the values contained in this map.

Map - klasser

`TreeMap<K,V>`

I en ***TreeMap*** lagras <key,value>-paret i en trädstruktur. Nyckeln avgör var i trädet paret lagras.

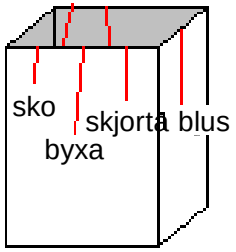
TreeMap medger snabb sökning.



`HashMap<K,V>`

I en ***HashMap*** är nycklarna inte ordnade

HashMap medger mycket snabb sökning då man räknar var elementet finns.



Maps.java **Search.java**

Ett par råd

1. ArrayList: använd det i stället för Vector

```
ArrayList<Integer> listName = new ArrayList<Integer>();
ArrayList<String> names= new ArrayList<String>();
```

Använd den generiska typen och ange explicit type av object den ska lagra.

```
ArrayList names2 = new ArrayList();  //Inte bra!!
List names3 = new ArrayList();  //Inte bra!!
```

2. Använd HashMap i stället för HashTable. Använd alltid den genriska typen och ange explicit typen av key och value.

```
HashMap<String, Person> info = new HashMap<String,
Person>();
HashMap info2 = new HashMap();  //inte bra!!
```

Iterator

Gränssnittet Iterator innebär att en klass måste implementera tre metoder.

Method Summary	
boolean	**hasNext**() Returns true if the iteration has more elements.
E	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator. Throws UnsupportedOperationException if the remove operation is not supported by this Iterator.

Med en Iterator går man i genom objekten i en containerklass:

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
Shape shape;
:
Iterator<Shape> iter = shapes.iterator();
while( iter.hasNext() ) {
   shape = iter.next();
    :
}

for( Iterator<Shape> iter = shapes.iterator(); iter.hasNext(); ) {
   shape = iter.next();
    :
}
```

Iterable

Fr.o.m. Java 1.5 kan man iterera genom en containerklass med en förenklad for-loop. Men man kan inte ändra värde på enkla variabler eller byta referens i referensvariabler.

```
int[] numbers = { 11, 42, -13, 8, 14 };
int sum = 0;
for( int a : numbers )
   sum += a;

List<Double> list = new ArrayList<Double>();
for( int i = 0; i < 5; i++ )
   list.add( new Double( Math.random() ) );
for( Double nbr : list )
    System.out.println( nbr );
```

För att man ska kunna iterera genom en struktur måste strukturen implementera gränssnittet *Iterable*.

Method Summary - Iterable	
Iterator<T>	**iterator**() Returns an iterator over a set of elements of type T.

Förenklad for-loop och objektsamlingar

För att den förenklade for-loopen ska kunna användas med en containerklass så måste klassen implementera gränssnittet Iterable. Då klassen implementerar Iterable lägger kompilatorn ut koden så här:

```
ObjectArray<Point> oa = new ObjectArray<Point>(); // ObjectArray implementerar Iterable
oa.add(new Point( 10, 2 ));
oa.add(new Point( 14, 9 ));
for(Point p : oa) {
    System.out.println( p );
    p.setLocation( 5, 9 );
}
```

Läggs ut som:

```
ObjectArray oa = new ObjectArray();
oa.add(new Point( 10, 2 ));
oa.add(new Point( 14, 9 ));
Point p;
for(Iterator i$ = oa.iterator(); i$.hasNext(); ) {
    p = (Point)i$.next();
    System.out.println( p );
    p.setLocation( 5, 9 )
}
```

Som du ser används först Iterable-implementeringen (oa.iterator()) och därefter Iterator-implementeringen (i$.hasNext() resp i$.next()). I for-loopen används en lokal variabel (p) med värde från aktuellt element.