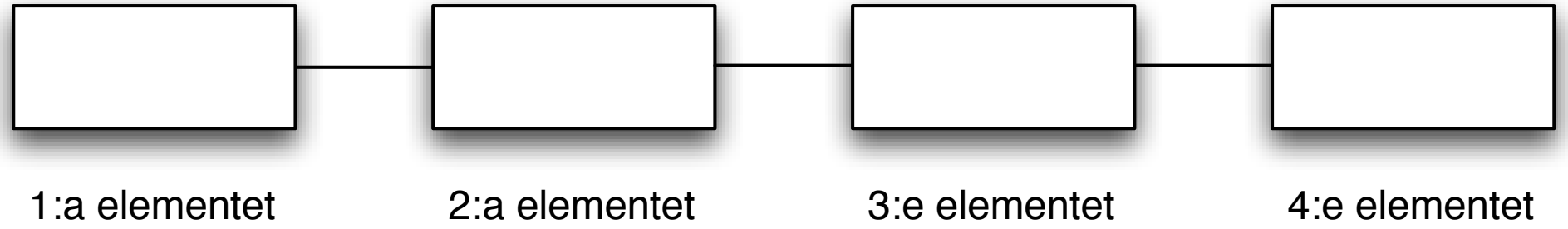
Abstrakta datatypen lista

Definition

En lista är en följd av element.

- Det finns en före-efter-relation mellan elementen.
- Begrepp som ”första elementet i listan”, ”efterföljaren till visst element i listan” är meningsfulla. Det finns alltså ett positionsbegrepp.
- Definitionen innebär *inte* att elementen är sorterade på något visst sätt t.ex. i storleksordning.



Abstrakt datatypen lista

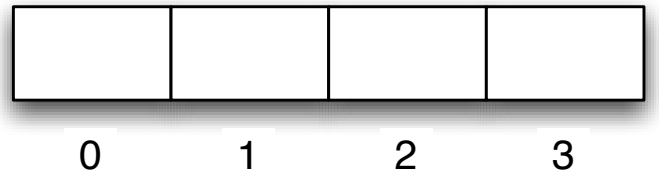
Abstrakt datatyp

En abstrakt modell tillsammans med de operationer man kan utföra på den.

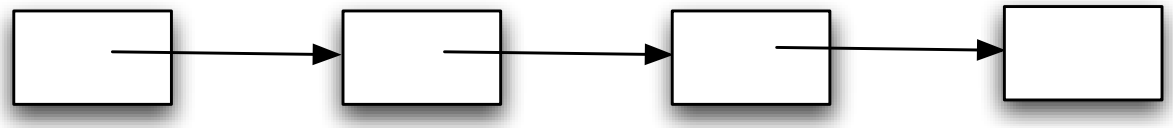
- Abstrakt modell: lista
- Operationer på modellen:
  - Lägga in element i listan (först, sist ...)
  - Ta bort ett element ur listan
  - Undersöka om ett visst element finns i listan
  - Ta reda på ett elementet i listan (första, sista ...)
  - Undersöka om listan tom
  - ...

Implementering av listor

- En vektor kan användas för att hålla reda på listans element.

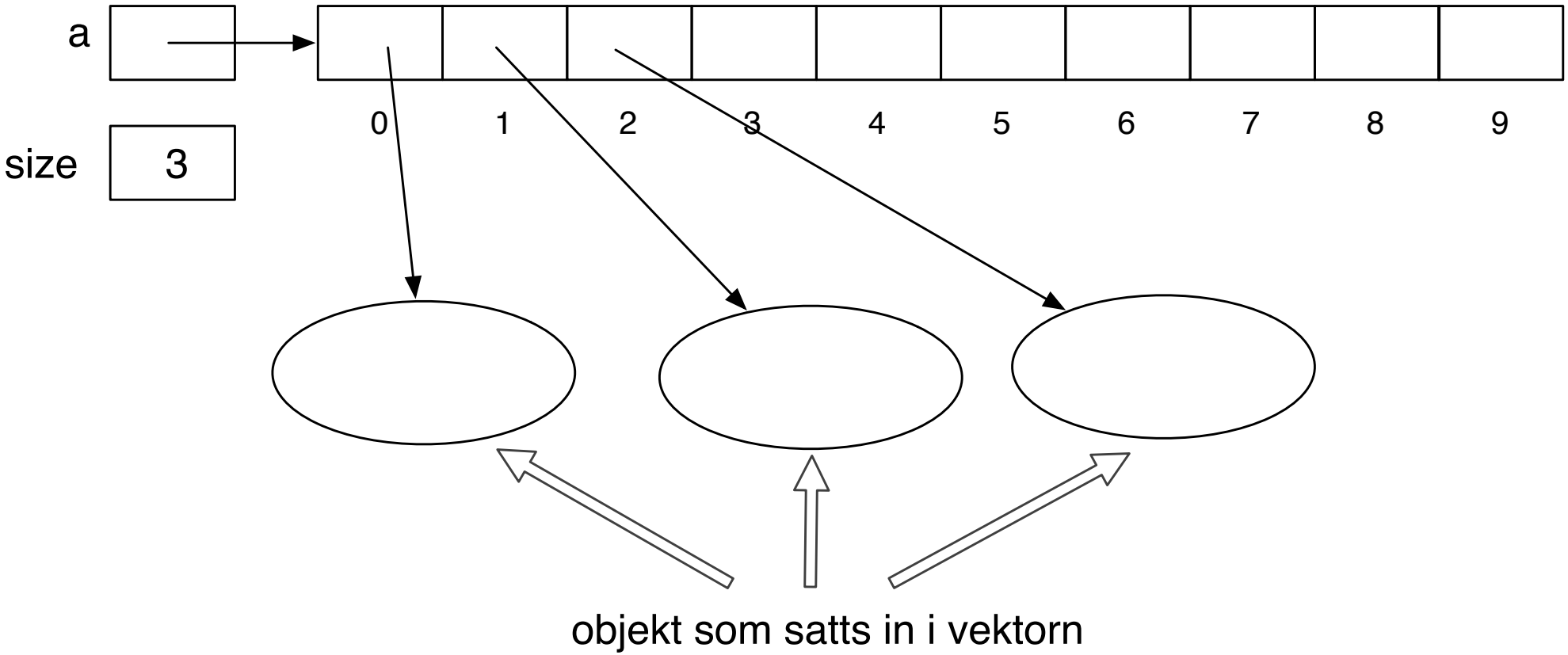


- Ett annat sätt är att utnyttja länkad datastruktur.
  - I en länkad struktur består listan av noder som har en referens till efterföljaren (och ev. till föregångaren).



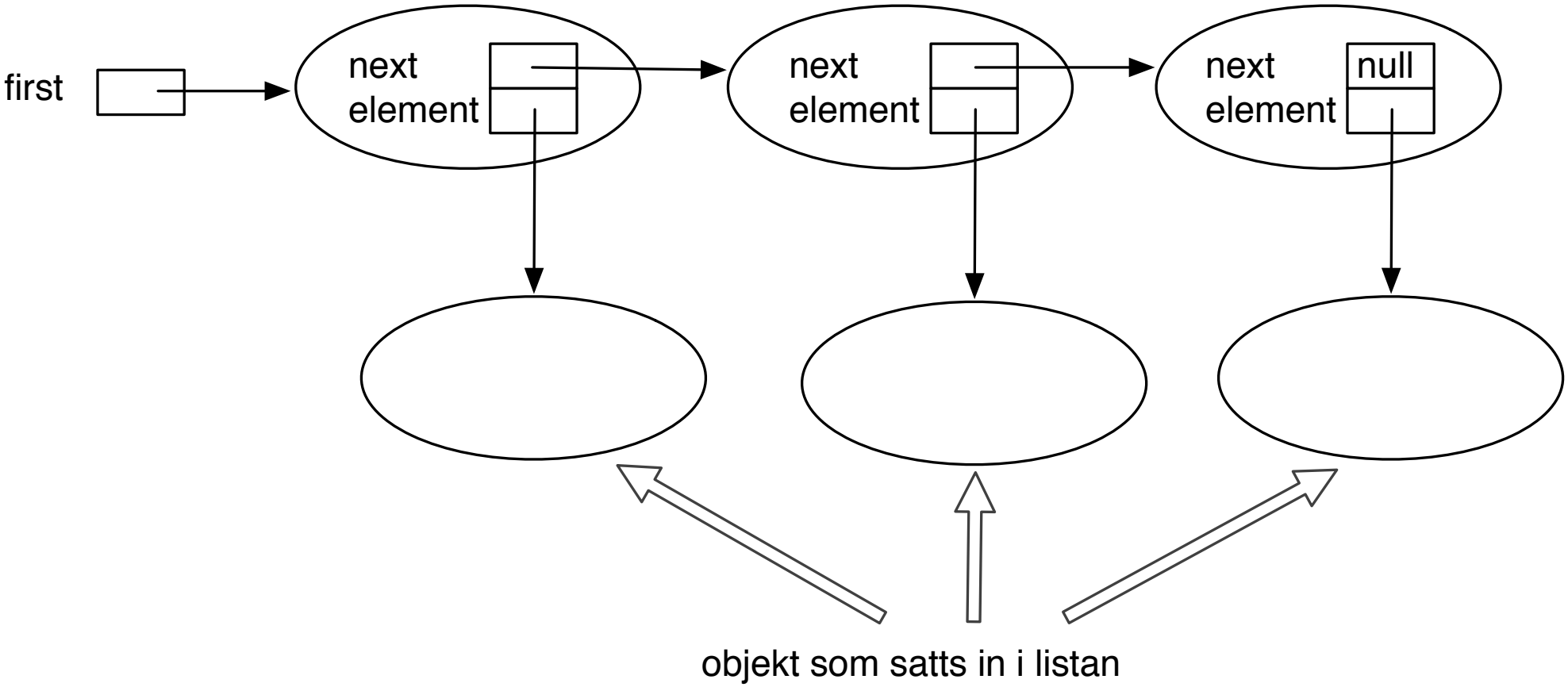
Egen implementering av listor från grunden

Vektor



Egen implementering av listor från grunden

Enkellänkad lista



Enkellänkad lista

Egen implementering från grunden

```
public class SingleLinkedList<E> {
   private ListNode<E> first; // referens till första noden
                              // null om listan är tom

   ..metoder..

   /* Statisk nästlad klass. Representerar en nod som
      innehåller ett element av typ E. */
   private static class ListNode<E> {
      private E element;         // data som lagras
      private ListNode<E> next; // refererar till nästa nod

      private ListNode(E e) {
         element = e;
         next = null;
      }
   }
}
```

Nästlade klasser i Java

Klasser kan deklareras *inuti* andra klasser (**nästlade klasser**).

- Används oftast när den nästlade klassen bara är meningsfull för den omgivande klassen.
- Användare behöver oftast inte känna till existensen av den nästlade klassen.
- En nästlad klass kan deklareras `private` om den bara ska användas i den omgivande klassen. Även konstruktorn kan då vara `private`.
- I den omgivande klassen har man tillgång till allt i den nästlade klassen (även det som är `private`).

- Det finns två typer av nästlade klasser:
  - **statiska nästlade klasser**
  - **inre klasser** (eng: inner classes).

Statiska nästlade klasser

```
public class OuterClass {
   ...

   public void p() {
      NestedClass x = new NestedClass();
      ...
   }

   private static class NestedClass {
      private NestedClass() {...}
      ...
   }
}
```

En statisk nästlad klass kan bara komma åt statiska attribut och statiska metoder i den omgivande klassen.

Inre klasser

```
public class OuterClass {
   private int i;

   public void p() {
      InnerClass x = new InnerClass();
      ...
   }

   private class InnerClass {
      private InnerClass() {...}

      private void q() {
         int b = i; ...;   // Här används i från OuterClass!
      }
   }
}
```

Ett objekt av en inre klass kan komma åt allt i det objekt av den omgivande klassen som skapade objektet av den inre klassen.

Att skapa objekt av nästlade klasser

- Görs oftast bara i den omgivande klassen.
  - Då blir det samma syntax som vanligt.
  - Exempel finns på föregående bilder.

- Man kan skapa objekt av nästlade klasser även utanför den omgivande klassen.
  - Kräver dock att den nästlade klassen och dess konstruktor är `public`.
  - Detaljer på nästa bild.

Att skapa objekt av nästlade klasser

Statiska nästlade klasser

Om den nästlade klassen är *statisk*:

```
public class OuterClass {
    ...

    public static class NestedClass {
        public NestedClass() {...}
        ...
    }
}
```

så skapas en instans av den nästlade klassen med följande syntax:

```
OuterClass.NestedClass x = new OuterClass.NestedClass(...);
```

Att skapa objekt av nästlade klasser

Inre klasser

Om den nästlade klassen är en *inre* klass:

```
public class OuterClass {
   ...

   public class InnerClass {
      public InnerClass(...) {...}
      ...
   }
}
```

så kan instanser av den inre klassen bara skapas genom ett objekt av den yttre klassen:

```
OuterClass a = new OuterClass();
OuterClass.InnerClass b = a.new InnerClass();
```

Inre klass eller statisk nästlad klass i vår listklass?

- Det fungerar alltid med en inre klass.

```
public class SingleLinkedList<E> {
    ...
    private class ListNode { ...
```

  - Men varje objekt av den inre klassen har en referens till ett objekt av den omgivande klassen.
  - Dessa referenser tar upp minne.

- Om man i den nästlade klassen bara behöver använda sådant som är deklarerat static i den omgivande klassen kan man istället ha en statisk nästlad klass.

```
public class SingleLinkedList<E> {
    ...
    private static class ListNode<E> { ...
```

  - I den statisk nästlade klassen `ListNode` når vi inte typparametern i den omgivande klassen. `ListNode` måste därför också ha en typparameter.

Exempel på metoder i en enkellänkad lista

Insättning och borttagning först i listan

Länka in en ny nod innehållande elementet x *först* i listan:

```
public void addFirst(E e) {
   ListNode<E> n = new ListNode<E>(e);
   n.next = first;
   first = n;
}
```

Tag bort första noden i listan, returnera dess innehåll:

```
public E removeFirst() {
   if (first == null) {
      throw new NoSuchElementException();
   }
   ListNode<E> temp = first;
   first = first.next;
   return temp.element;
}
```

Traversering av elementen i listan

Exempel: metoden `toString`

Returnera en sträng som representerar listan:

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append('[');
    ListNode<E> p = first;
    while (p != null) {
        sb.append(p.element.toString());
        if (p.next != null) {
            sb.append(", ");
        }
        p = p.next;
    }
    sb.append(']');
    return sb.toString();
}
```

Traversering av elementen i listan

Mönster

```
ListNode<E> p = first;
while (p != null) {
    ...
    p = p.next;
}
```

Exempel på metoder i en enkellänkad lista

Insättning sist i listan

Länka in en ny nod innehållande elementet `x` `sist` i listan:

```
public void addLast(E e) {
   ListNode<E> n = new ListNode<E>(e);
   if (first == null) {
      first = n;
   } else {
      ListNode<E> p = first;
      while(p.next != null) {
         p = p.next;
      }
      p.next = n;
   }
}
```

Söka upp sista noden i listan

Mönster

```
if (first == null) {
    ...
} else {
    ListNode<E> p = first;
    while (p.next != null) {
        p = p.next;
    }
    // Här refererar p till sista noden
}
```

Exempel på metoder i en enkellänkad lista

Borttagning sist i listan

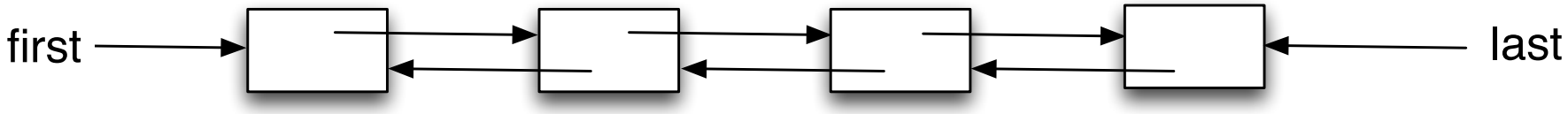
```
public E removeLast() {
    if (first == null) {          // tom lista
        throw new NoSuchElementException();
    }
    if (first.next == null) {     // ett element
        ListNode<E> temp = first;
        first = null;
        return temp.element;
    }
    ListNode<E> p = first;        // minst två element
    ListNode<E> pre = null;
    while (p.next != null) {
        pre = p;
        p = p.next;
    }
    pre.next = null;
    return p.element;
}
```

Fundera på

- Två av metoderna vi implementerat, `addlast` och `removeLast`, är långsammare än motsvarande metoder för att sätta in och ta bort i början av listan. Både `addlast` och `removeLast` innehåller en loop.
- Hur skulle man kunna implementera listklassen så att dessa loopar kan tas bort.

Dubbellänkad lista

- Vissa operationer blir krångliga och ineffektiva i den enkellänkade implementeringen.
  - Dessa kan förenklas om man i varje nod också har en referens till föregångaren. Detta kallas dubbellänkade listor.



  - Man kan förenkla implementeringar av vissa operationer ytterligare genom att ha ett speciellt element ("huvud") i början av listan.

Använda klassen SingleLinkedList

- När vi använder vår egen generiska listklass ska vi ange typargument som vanligt:

```
SingleLinkedList<Integer> list = new SingleLinkedList<>();
    list.addFirst(10);
    ...
```

- Däremot kan vi inte iterera över listan med en for-each-loop.
  - För att det ska vara möjligt måste vi låta klassen implementera `Iterable` och skriva en egen iterator-klass.

```
public class SingleLinkedList<E> implements Iterable<E> {
    ...
    public Iterator<E> iterator() {
        return new MyListIterator();
    }
    ...
    // Inre klass som implementerar interfacet Iterator
    private class MyListIterator implements Iterator<E> {...}
```