Sammansatta datatyper
(Arrayer, Collections, ...)
Generics: Parametrisk polymorfism

Extraavsnitt på svengelska

Arrayer i Java

Listor och arrayer

Enligt TDDD73 (tidigare Pythonkurs):

"**Listan** är en mycket användbar sammansatt datatyp.
I många andra språk kallar man listor för **arrayer**."

Många språk har **både** listor och arrayer – även Java och Python!

Exakta definitioner varierar…

Arrayer

Oftast lägre nivå, mer maskinnära,
mindre funktionalitet
Lagras "sekventiellt" i minnet
Vissa operationer mycket snabba

Ofta grundläggande datatyp i språket,
egen syntax

Listor

Fokus på funktionalitet / gränssnitt,
många olika sätt att lagra dem
Olika implementation →
olika egenskaper

Python har *listor*
som grundläggande datatyp

Repetition!

```
Circle[] circles;
int[] numbers;
```

För varje typ T
finns en arraytyp T[]

```
Circle[] circles = new Circle[10];
int[] numbers = new int[rnd()];
```

Arrayer skapas med **new**
Storleken sätts när de skapas, ändras aldrig

```
int second = numbers[1];
```

Element hämtas ut med index (första = 0)

```
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

Index: 0…arr.length-1

```
int val = numbers[42000];
```

Indexkontroll: Ger
ArrayIndexOutOfBoundsException

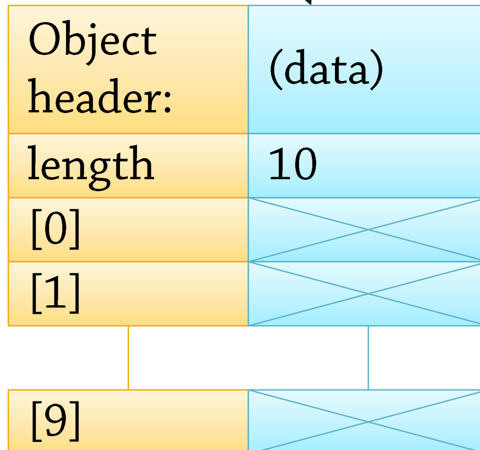
numbers	

Object header:	(data)
length	42
[0]	0
[1]	0
[41]	0

Tomma element skapas och *initialiseras*: 0, 0.0, null, false

Arrayer 2: Pekar-arrayer

- Objektarrayer innehåller **pekare**:
  - Circle[ ] circles = **new** Circle[10];



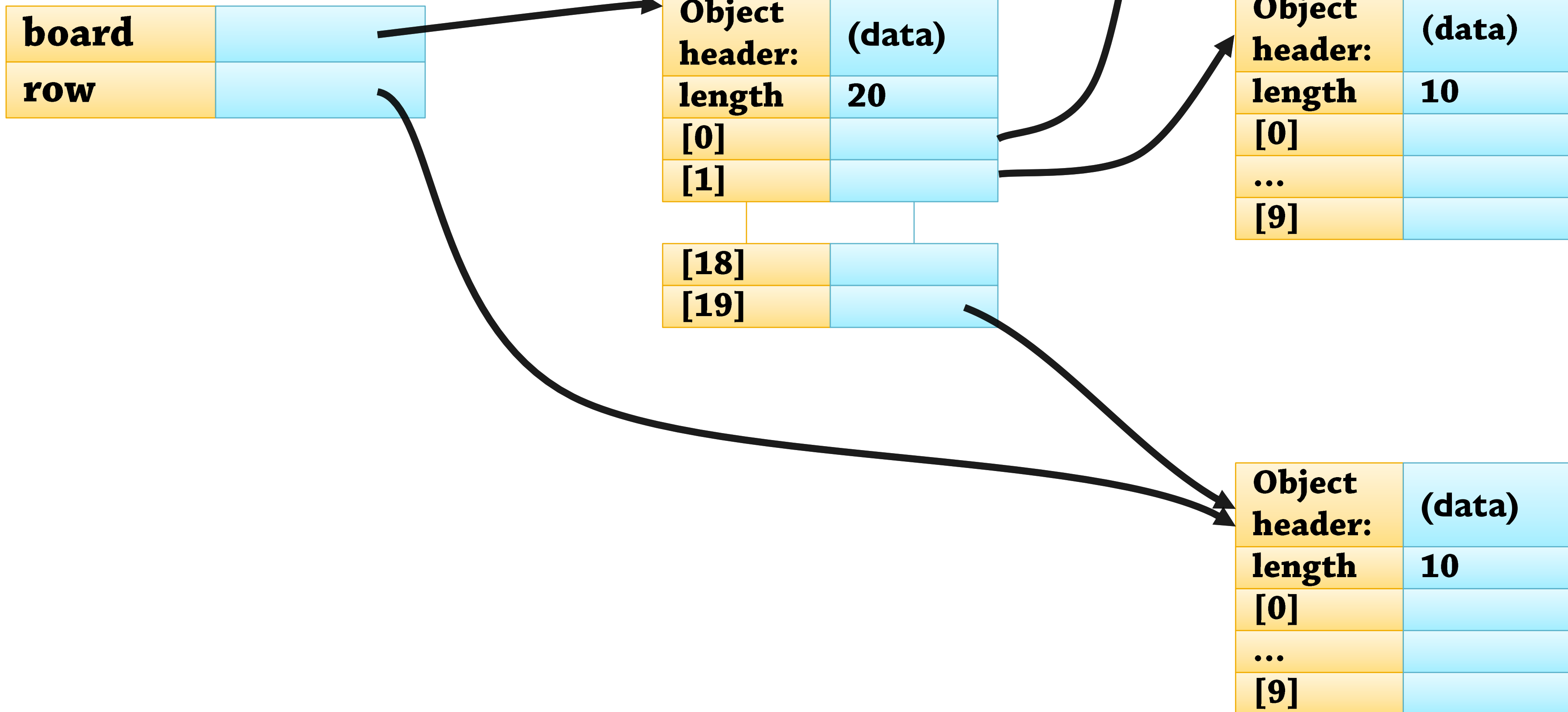
Element initialiseras till **null**

Inga nya **Circle**-objekt skapas!

Arrayer 3: Flerdimensionella arrayer

jonkv@ida

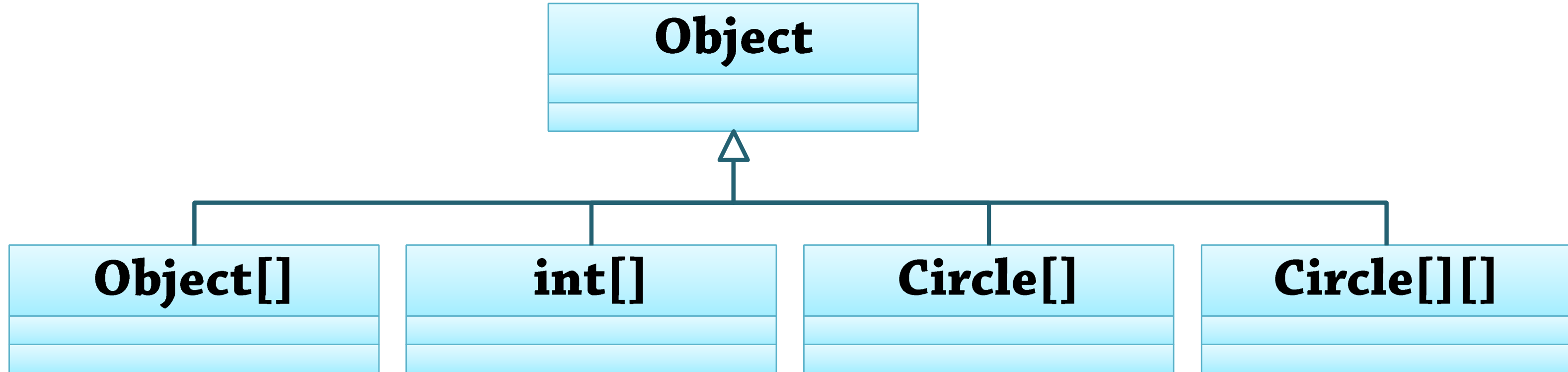
- **Arrayer av arrayer**:
  - Square[ ][ ] board = **new** Square[20][10];
  - Square[ ] row = board[19];
  - Square pos = board[1][0];



Arrayer 4: Tillgängliga metoder

jonkv@ida

- Arraytypen är **subklass** till **Object**, **ärver metoder**



Arrayer 5: Default-metoder

- Av historiska anledningar: Bara **default**-implementationerna

Default-implementationer…

equals() testar identitet

```
int[ ] ar1 = new int[3]; // 0, 0, 0
int[ ] ar2 = new int[3]; // 0, 0, 0
System.out.println(ar1.equals(ar2));
// false
```

Kan inte ändras:
Bakåtkompatibilitet!

Vad vi oftast vill

Hjälpmetoder i java.util.Arrays

```
int[ ] ar1 = new int[3]; // 0, 0, 0
int[ ] ar2 = new int[3]; // 0, 0, 0
System.out.println(Arrays.equals(ar1,ar2));
// true
```

Också i java.util.Arrays: sort(), binarySearch(), copyOfRange(), deepToString(), hashCode(), fill(), …

Default-implementationer…

toString(): typnamn + ID

```
System.out.println(new int[10].toString());
// [I@3effd44e
System.out.println(new Circle[4].toString());
// [LCircle;@2b78146d
```

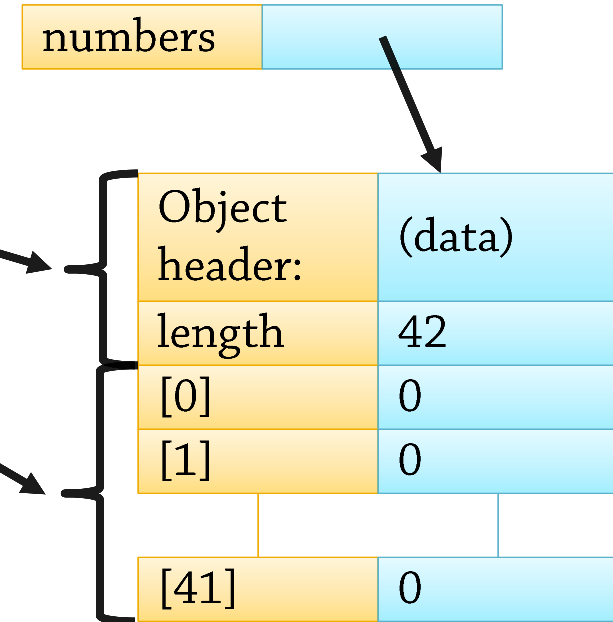
Vad vi oftast vill

Hjälpmetoder i java.util.Arrays

```
System.out.println(
   Arrays.toString(new int[10]));
// [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Arrayer 6: Styrkor och svagheter

- Styrka: Effektiv användning av **minne**
  - *Liten* konstant "startkostnad", 12–20 bytes
  - Plus elementens "*verkliga*" kostnad
    - Array av **short:** 2 bytes/element



- **Svaghet**: Saknar viss funktionalitet – kan t.ex. inte ändra längd!
  - Saknar metoder som add(), append(), remove(), …
- **Svaghet**: Bara en ”variant” med fixerade egenskaper
  - Java har *många* olika sorters listor

Listor i Java

Listor: Intro

- **Listor**:
  - **Variabel storlek** → bekvämare att använda!
  - Ingen speciell **syntax**, utan vanliga **klasser**

- För bättre förståelser för **listor** och **generics**:
  - Vi visar **möjliga implementationer**
  - Börja med ett **gränssnitt**

```
public interface StringList {
    public void add(String element);
    public String get(int index);
}
```

Arraylistor 1: Alternativ

- Ett sätt att lagra listor: **med arrayer**

```
public class StringArrayList implements StringList {
    private String[]        elements;
… }
```

Listan har men är inte en array
Metoderna använder arrayen

Alt. 1: Arraystorlek = listlängd	
Skapa tom lista →	
Addera element →	1
Addera element →	1 2
Addera element →	1 2 3
Addera element →	1 2 3 4
Addera element →	1 2 3 4 5

Alt. 2: Allokera "extra" element	
Skapa tom lista →	- - - -
Addera element →	1 - - -
Addera element →	1 2 - -
Addera element →	1 2 3 -
Addera element →	1 2 3 4
Addera element →	1 2 3 4 5 - - -

Vid varje tillägg:
Allokera array med ny längd
Kopiera över gamla innehållet
Lägg till det nya elementet

Ha en "reserv" av tomma platser
Fullt → dubblera storleken

Måste lagra *antal använda platser*!

Arraylistor 2: En listklass

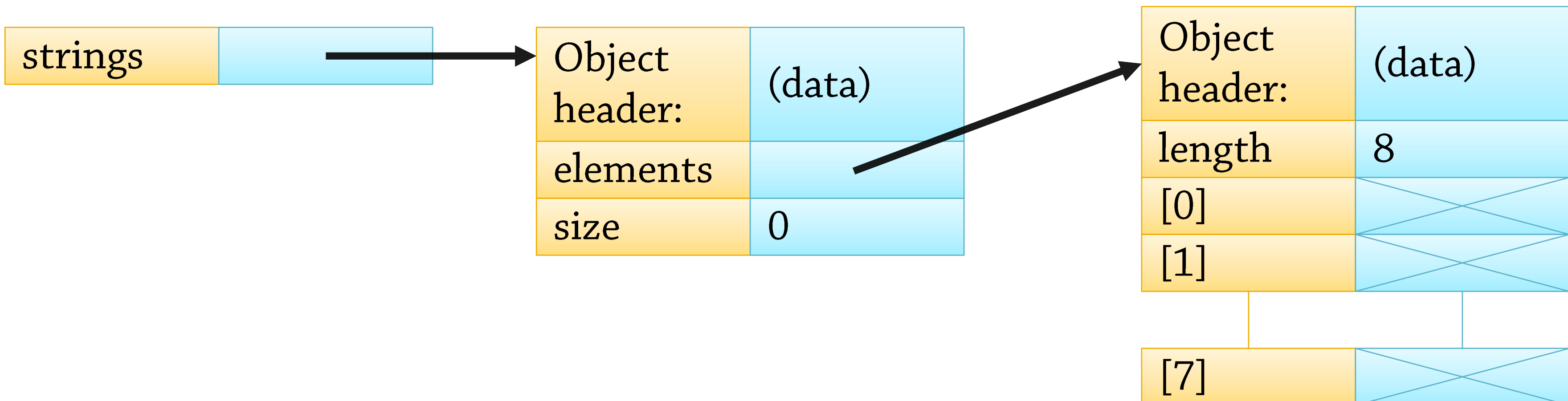

```
public class StringArrayList implements StringList {
    private String[]    elements;
    private int         size;   // Använda platser
```

```
    public StringArrayList() {
        this.elements = new String[8];
        this.size = 0;
    }
```

Allokerar lite extra plats

Hur många av platserna är **använda**?

```
StringList strings = new StringArrayList();
```

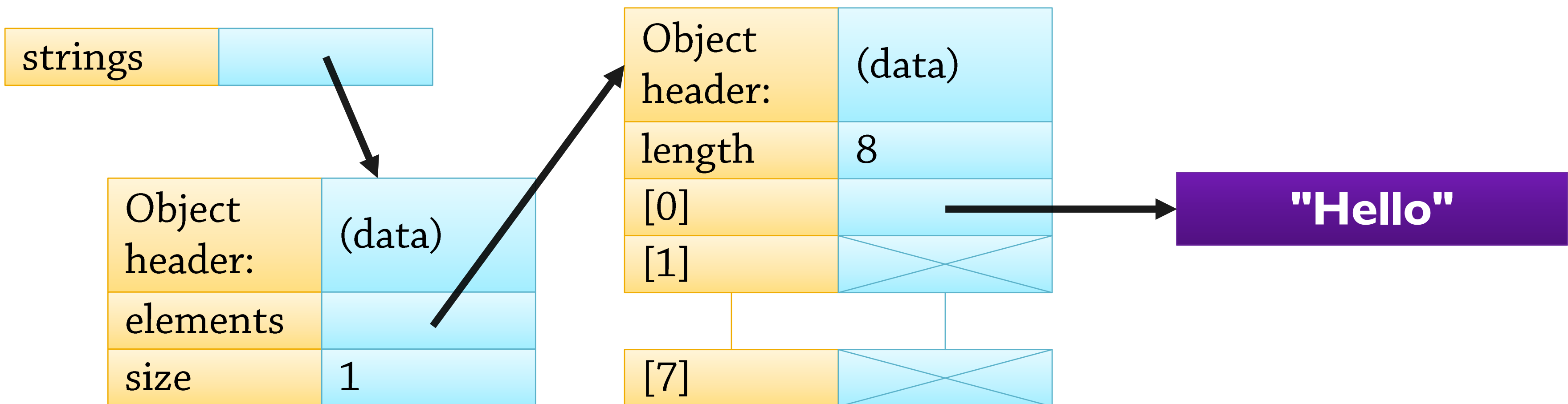


Arraylistor 3: Addera element

jonkv@ida

```
public void add(String element) {
    elements[size] = element;
    size++;
}
```

```
StringList strings = new StringArrayList();
strings.add("Hello");
```



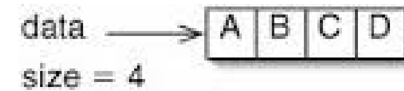
Men… vad händer när arrayen blir full?

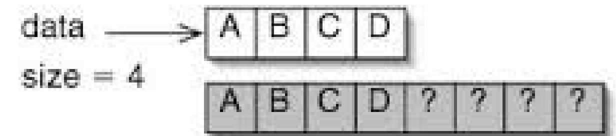
Arraylistor 4: Utöka utrymmet

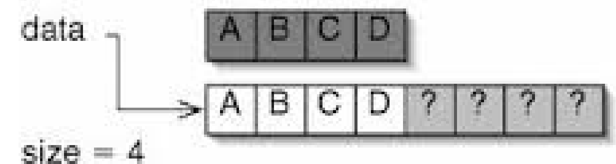

```
public void add(String element) {
   if (size == elements.length) {
      int newsize = 2*size;
      String[] arr2 = new String[newsize];
      System.arraycopy(…);
      this.elements = arr2;
   }
   elements[size] = element;
   size++;
}
```

Är interna lagringen full?

Skapa större array,
kopiera innehållet,
kasta bort gamla arrayen!







Arraylistor 5: Hämta element

```
public String get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    } else {
        return elements[index];
    }
}
```

Kolla index:
Tillåt inte get(4) när size==2!

Arraylistor 6: Stoppa in element

- Effektiv representation – för de *flesta* operationer, men…

```
public void insert(int index, String element) {
    if (size == elements.length) { … allocate more memory … }

    System.arraycopy(elements, index, elements, index + 1, size - index);
    elements[index] = element;
    size++;
}
```

A	B	C	D	E	F	G	H								

```
list.insert(2, "X");
```

A	B	C	C	D	E	F	G	H							

Kopiera – tar tid!

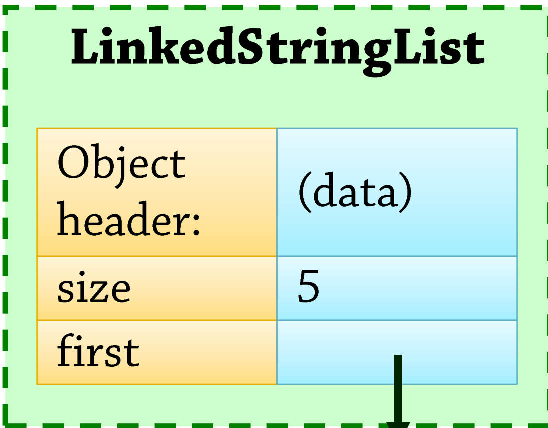
A	B	X	C	D	E	F	G	H							

Stoppa in

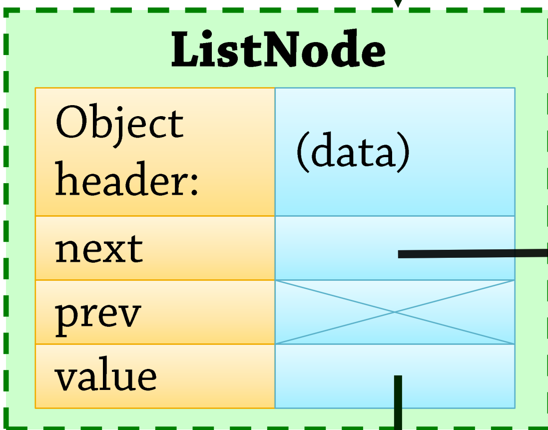
Länkade listor 1: Principen

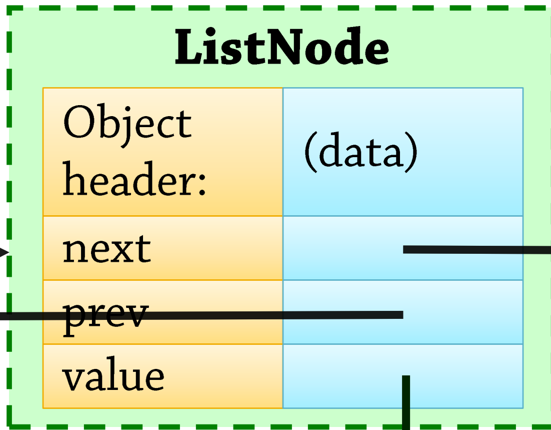
jonkv@ida

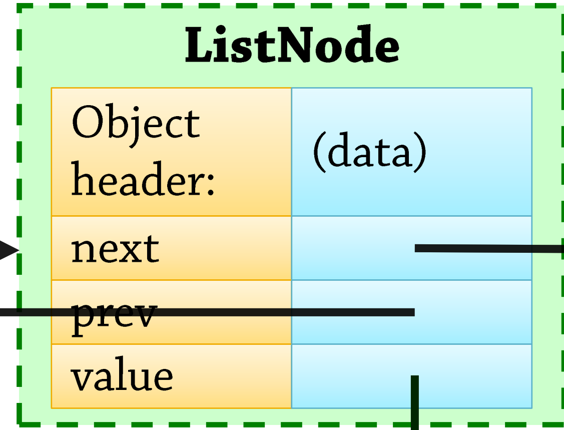
- Principen bakom **länkade** listor:

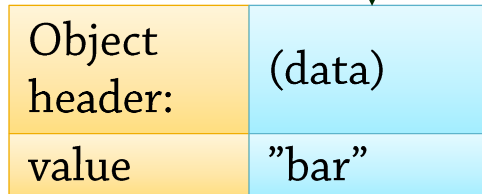


En extra listnod per element
➔ använder mer minne





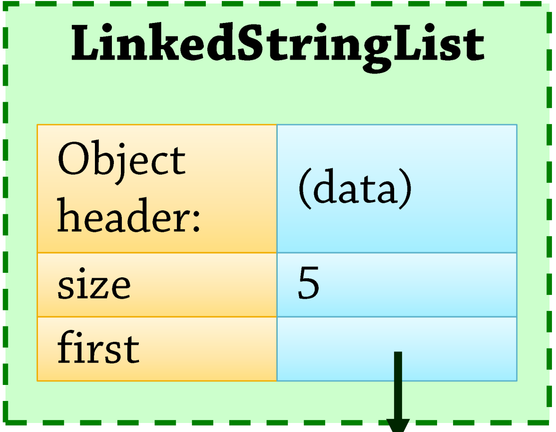


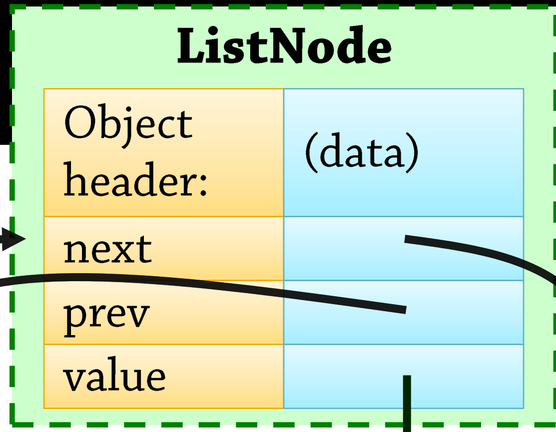


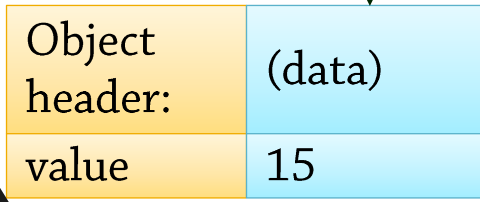
Länkade listor 2

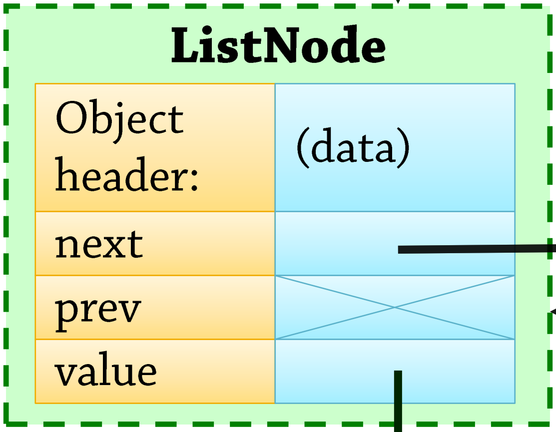
jonkv@ida

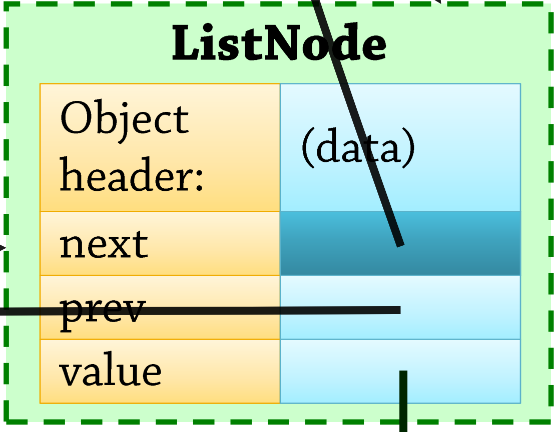
Snabbt att skjuta in / ta bort värden

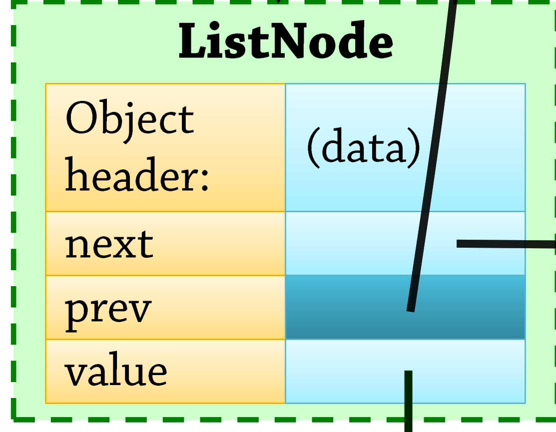


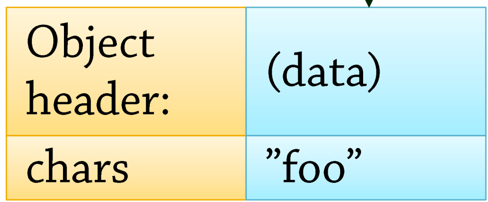


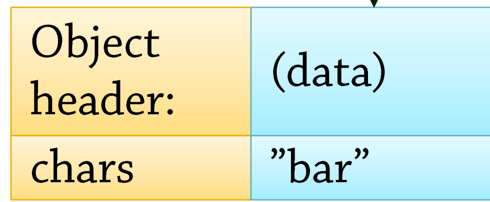


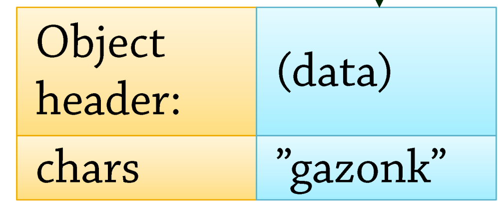












Länkade listor 3

jonky@ida

Tar lång tid att ta fram värde nummer *n* i listan

LinkedStringList

Object header:	(data)
size	5
first	

```
String get(int index) {
    ListNode x = first;
    for (int i = 0; i < index; i++)
        x = x.next;
    return x.value;
}
```

ListNode

Object header:	(data)
next	
prev	
value	

Object header:	(data)
chars	”foo”

ListNode

Object header:	(data)
next	
prev	
value	

Object header:	(data)
chars	”bar”

ListNode

Object header:	(data)
next	
prev	
value	

Object header:	(data)
chars	”gazonk”

Använda arrayer eller listor?

Arrayer används:

- För att **implementera** datatyper
  - ArrayList måste använda en "primitiv" array
- När **varje nanosekund** spelar roll
  - Bara efter CPU-profilering som visar att listhantering spelar roll
- När **varje byte** spelar roll
  - Bara efter minnesprofilering…
- Ibland om **storleken** är **fixerad**
  - Om storleken varierar:
Låt listorna hantera detta!

Listor används:

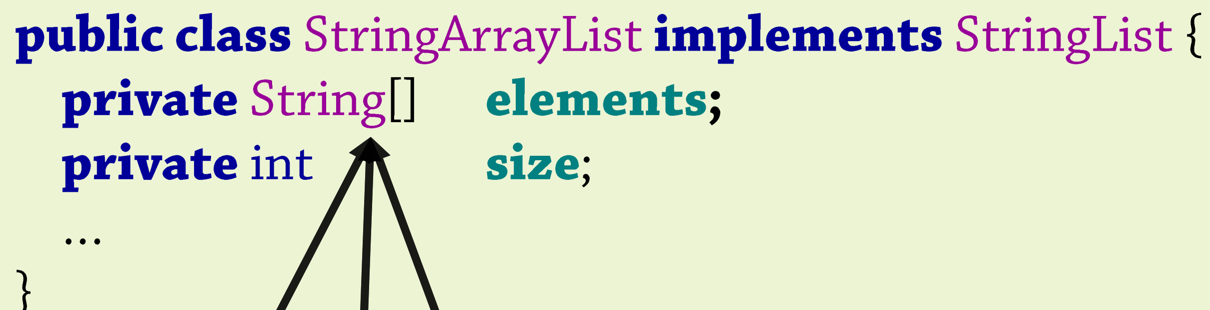
- Nästan alltid!
  - Många hjälpfunktioner tillgängliga
  - Kan lätt lägga till och ta bort element, osv
  - Används av många klasser i Javas klassbibliotek
  - …

Tetrislabben använder arrayer eftersom Board-storleken är fixerad!

Listor och elementtyper

Elementtyper 1: Typspecifika listor?

- Våra listklasser lagrar bara strängar!



Behöver vi en listklass för varje elementtyp?

Elementtyper 2: Godtyckliga objekt

```
public class ObjectArrayList {
    private ] elements;
    private int size;

    public ObjectArrayList() {
        this.elements = new 8];
        this.size = 0;
    }

    public get(int index) {
        if (index >= 0 && index < size) { return elements[index]; }
        else /* Signal error */
    }

    public void add( element) {
        if (size == elements.length) { /* Create a new array; copy old element */ }
        elements[size] = element;
        size++;
    }
}
```

Array av Object-pekare,
kan peka på objekt av godtycklig klass

Elementtyper 3: Problem

- Nu får vi **nya problem**!
  - 1: **Ingen typsäkerhet** när vi **adderar** element!
    - Vill ha en lista där vi stoppar in strängar:

```
ObjectArrayList greetings = new ObjectArrayList();
```

    - Kan lägga in strängar:

```
greetings.add("hello");
```

    - Kan råka lägga in cirklar också – inget fel vid kompilering eller körning!

```
greetings.add(new Circle(1, 1, 3));
```

```
public class ObjectArrayList {
    public ObjectArrayList() { … }
    public Object get(int index) { … }
    public void add(Object element) { … }
}
```

Elementtyper 4: Problem

- Problem 2: **Ingen typinformation** när vi **hämtar** element!

```
ObjectArrayList greetings = new ObjectArrayList();
greetings.add("hello");
greetings.add(new Circle(1, 1, 3));
...
String str = greetings.get(0);                    // Kompileringsfel
```

```
String str = (String) greetings.get(0);           // OK
```

Vi måste tala om för kompilatorn:
"Jag *vet* att jag la en sträng på position 0 – jag lovar!"
Krävs en *cast* – extra arbete för oss

```
public class ObjectArrayList {
    public ObjectArrayList() { … }
    public Object get(int index) { … }
    public void add(Object element) { … }
}
```

Elementtyper 5: Problem

- Problem 3: Tänk om vi har fel!

```
ObjectArrayList greetings = new ObjectArrayList();
greetings.add("hello");
greetings.add(new Circle(1, 1, 3));

for (int i = 0; i < greetings.size(); i++) {
    String str = (String) greetings.get(i);
    // Runtime error for i=1 – it's a Circle!
}
```

Upptäcks inte när programmet kompileras

Generics i Java:
Parametrisk Polymorfism

Vanliga parametrar till metoder


```
void printMult3() {
    for (int i = 1; i <= 13; i++) {
        System.out.println(3 * i);
    }
}
```

```
void printMult5() {
    for (int i = 1; i <= 13; i++) {
        System.out.println(5 * i);
    }
}
```

```
void printMult(int num) {
    for (int i = 1; i <= 13; i++) {
        System.out.println(num * i);
    }
}
```

num är en ***heltalsvariabel***

Kan ha värdet 3, 4, 5, 8, …

```
void printMult4() {
    for (int i = 1; i <= 13; i++) {
        System.out.println(4 * i);
    }
}
```

```
void printMult8() {
    for (int i = 1; i <= 13; i++) {
        System.out.println(8 * i);
    }
}
```

Typparametrar till klasser


```
class ListOfString {
   String[] elements;
   int      size;
   String   get(int index)         { … }
   void     add(String element) { … }
}
```

```
class ListOfShape {
   Shape[] elements;
   int     size;
   Shape   get(int index)         { … }
   void    add(Shape element) { … }
}
```

```
class MyList<E> {
   E[]      elements;
   int      size;
   E        get(int index)      { … }
   void     add(E element)      { … }
}
```

E* är en *typvariabel

Kan ha värdet String, Shape, …

```
class ListOfCircle {
   Circle[] elements;
   int      size;
   Circle   get(int index)         { … }
   void     add(Circle element) { … }
}
```

```
class ListOfButton {
   Button[] elements;
   int      size;
   Button get(int index)         { … }
   void     add(Button element) { … }
}
```

Generics 1: Intro

- **Generiska typer** med **typparametrar**

```
class              {
   E[]       elements;
   int       size;
   E         get(int index)       { return elements[index]; }
   void      add(E element)       { … } // Takes a parameter of type E
}
```

En array där elementen har typ *E*
– verkliga typen är inte angiven än

- Kan "instansieras" med olika typer som parameter

MyList<String> – *nästan* exakt som:

```
class MyList<String> {
   String[] elements;
   int      size;
   String   get(int index)        { … }
   void     add(String element) { … }
}
```

MyList<Shape> – *nästan* exakt som:

```
class MyList<Shape> {
   Shape[] elements;
   int      size;
   Shape    get(int index)        { … }
   void     add(Shape element) { … }
}
```

```
class MyList<E> {
    E[] elements;
    int size;
    E get(int index)            { return elements[index]; }
    void add(E element)         { … }
}
```

- En lista där **E=String**:

```
MyList<String> greetings = new MyList<String>();
greetings.add("hello");                    // OK
greetings.add(new Circle(1, 1, 3));        // Compile-time error: add(String)
String str = greetings.get(0);             // No cast needed: String get(int index)
```

- En lista där **E=MyList<String>** ➔ en lista av listor!

```
MyList<MyList<String>> lists = new MyList<MyList<String>>();
lists.add(greetings);
```

Generics 3: Att skriva mindre


```
class MyList<E> {
    E[] elements;
    int size;
    E get(int index)            { return elements[index]; }
    void add(E element)         { … }
}
```

- Mycket repetition:

```
MyList<MyList<String>> listor = new MyList<MyList<String>>();
```

- Syntaktiskt socker: *Diamant*-operatorn (betyder "samma som förut")

```
MyList<MyList<String>> listor = new MyList<>();
```

DRY-principen: Don't Repeat Yourself!

Regelbrott är WET: Write Everything Twice…

Generics 4: "Råa" typer

```
class MyList<E> {
    E[] elements;
    int size;
    E get(int index)              { return elements[index]; }
    void add(E element)           { … }
}
```

- Man *kan* använda en generisk typ **utan** att ange typparametrar:

```
MyList listor = new MyList();  // Defaultvärde för E blir Object
```

- **Undvik** – enbart för **kompatibilitet med gammal kod**!
  - Måste kunna kompilera kod skriven före 2004,
    när generics inte fanns

Parametrisk polymorfism

jonkv@ida

- Kallas *generisk programmering*, ger **parametrisk polymorfism**
  - Samma namn (MyList)
  - Olika beteende (beroende på typparameter)
- Jämför med **subtypspolymorfism**
  - Samma metodnamn, flera implementationer inom en typhierarki
- Jämför med **ad-hoc-polymorfism** (overloading)
  - Samma metodnamn, flera implementationer i samma klass

Generics: Wildcards

(Bonusmaterial på engelska)

Generics: Wildcards


```
static boolean contains(List<String> haystack, Object needle) {
    for (int i = 0; i < haystack.size(); i++) {
        if (haystack.get(i).equals(needle)) return true;
    }
    return false;
}
```

Works fine, but *only* for string lists!

```
public static void main(String[] args) {
    List<String> list = ...;
    if (contains(list, "hello")) { ... }
}
```

```
static boolean contains(List<Object> haystack, Object needle) {
    for (int i = 0; i < haystack.size(); i++) {
        if (haystack.get(i).equals(needle)) return true;
    }
    return false;
}
```

Seems more general…

```
public static void main(String[] args) {
    List<String> list = ...;
    if (contains(list, "hello")) { ... }
}
```

…but won't compile! A list of Strings *is not* a list of Objects!

Generics: Wildcards (2)

- If a **String** is a kind of **Object**…
  - …why isn't a **List<String>** a kind of **List<Object>** (or vice versa)?

List<Object>	List<String>
public void add(E element) **public void add(Object element)** **Promises to be able to add any Object**	**public void add(E element)** **public void add(String element)** **Can only add Strings, violating the List<Object> contract**
public E get(int index) **public Object get(int index)** **Only promises to return some Object**	**public E get(int index)** **public String get(int index)** **Promises to always return String (OK: Covariant return types)**

Generics: Wildcards (3)

```
static boolean contains(List<?> haystack, Object needle) {
    for (int i = 0; i < haystack.size(); i++) {
        if (haystack.get(i).equals(needle)) return true;
    }
    return false;
}
```

"List of whatever…" We don't care what the element type is

```
public static void main(String[] args) {
    List<String> list = …;
    if (contains(list, "hello")) { … }
}
```

Works!

…but contains() can't call haystack.add():

haystack.add("Hello") ➔ what if the list's actual element type is SquareType?

Generic Methods
Type dependencies between arguments

Generics: Type Variables for Methods

```
public class ArrayTools
{
   static void arrayToCollection1(Object[] arr, Collection<Object> coll) {
      for (Object o : arr) {
         coll.add(o);
      }
   }
}
```

Only works for object arrays and object collections

How do we generalize?

```
public class ArrayTools
{
   static <T> void arrayToCollection2(T[] arr, Collection<T> coll) {
      for (T o : arr) {
         coll.add(o);
      }
   }
}
```

The class isn't *necessarily* generic

The array and collection must have the *same* element type T, but we don't care what T is

Generics:
Bounded Wildcards

Bounded Wildcards 1

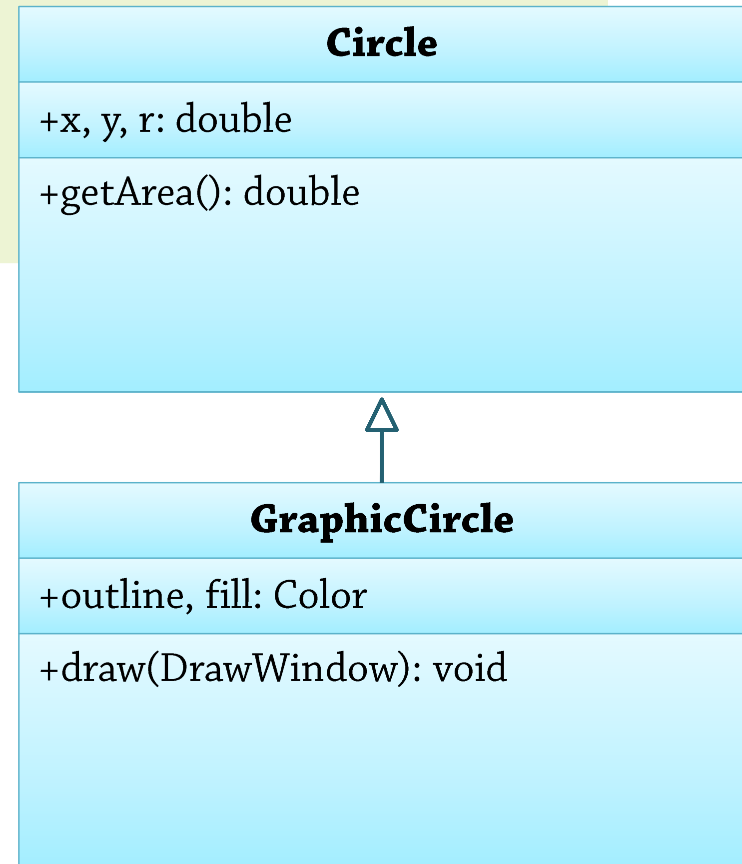
jonkv@ida

```
public class ArrayTools
{
    static <T> void arrayToCollection2(T[] arr, Collection<T> coll) {
        for (T o : arr) {
            coll.add(o);
        }
    }
}
```

Types have to be matched exactly!

How do we add an array of GraphicCircles into a collection of Circles?

Should be possible:
If it can contain *any Circle*,
it can contain GraphicCircle objects



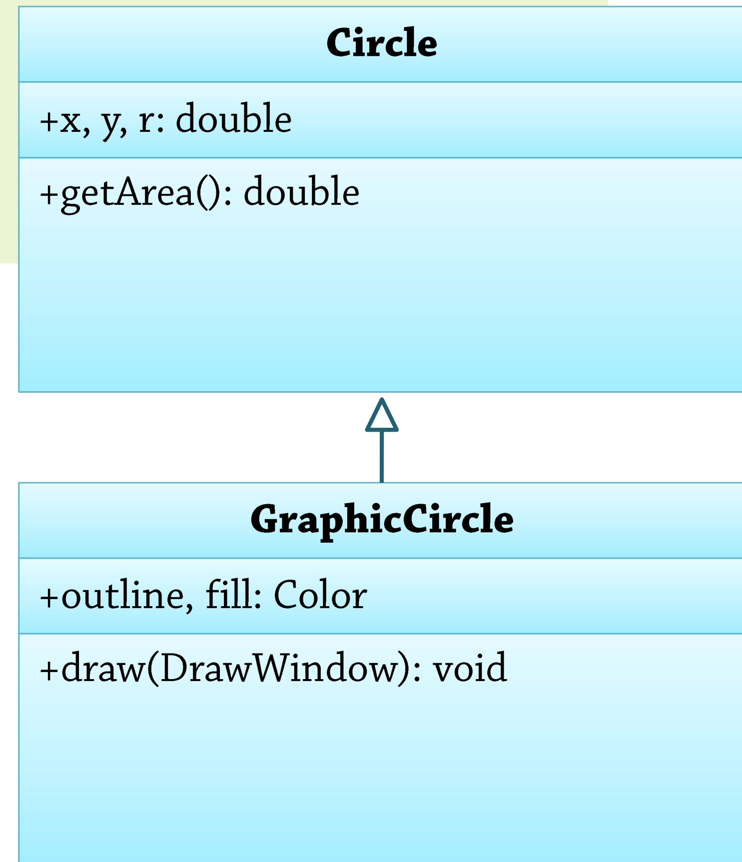
Bounded Wildcards 2

jonkv@ida

```
public class ArrayTools
{
   static <T> void arrayToCollection2(T[] arr, Collection<? super T> coll) {
      for (T o : arr) {
         coll.add(o);
      }
   }
}
```

"We need an array of *some* type T, and a collection whose element type is *T or something above (super) it.*

T = GraphicCircle, ? = Circle
T = Player, ? = GameObject
T = String, ? = Object



Generics:
Bounded Wildcards – another example

Generics: Element Types

```
class MyList<E> implements List<E> {
   E[] elements;
   int length;

   void add(E element) {
      if (element.equals(...)) { ... }

      String str = element.toString();



      ... element.getBounds() ...
   }
}
```

What can we do with **element**?
Depends on its static (apparent) type,
but E is a type *variable / placeholder*…

E could be String, Component, Shape,…
but *all* classes inherit from
java.lang.Object!

equals() and toString() can be used:
Defined in Object

getBounds() can't be used:
Exists in java.awt.**Shape** (for example),
but this code is also used for
MyList<**String**>…

Generics: Type Bounds / extends

- **Bounds on types**
  - A ShapeList that can **only** be applied to Shape subclasses:

```
class ShapeList<                      implements List<E>{
  // Any E is at least a Shape:
  // Might be Area, Rectangle2D, Arc2D, ...
  E[] elements;
  int length;

  void getLargest() {
    ...
    E element = ...;
    if (element.getBounds() ... ) ...
    ...
  }
}
```

"Must specify *some* class E that is a *subtype* of Shape"

getBounds() can be used: E is some kind of Shape, where this method exists

```
new ShapeList<Area>()             // OK, can only store Areas
new ShapeList<Rectangle2D>()      // OK, can only store Rectangle2Ds
new ShapeList<String>()           // Wrong: String is not a Shape
```

Generics: Type Bounds / extends (2)

- How about another example?
  - ```
    public abstract class GameObject {
        public abstract void tick();
    }
    ```
  - ```
    public class GameObjectList<E extends GameObject> {
        // We know that any E is a GameObject or a subtype

        private List<E> objects;

        public void tickAllObjects() {
            for (E object : objects) {
                // If GameObject has tick(), we can do object.tick()
                // Even through E might be Player
                object.tick();
            }
        }
    }
    ```
  - `new GameObjectList<Player>()  // Can only contain Players (subclass of GameObj)`
  - `new GameObjectList<Enemy>()`

Generics: Objekttyper och primitiva typer
Wrappers

Typparametrar och objekttyper

En typparameter måste ange en *objekttyp!*

```
class List<E> {
    E[]     elements;
    int     size;
    E       get(int index)       { … }
    void    add(E element)       { … }
}

List<String> strings; // OK
List<Button> buttons; // OK

List<int> numbers; // Does not work!
```

…för egentligen skapas bara *en* klass!

```
class List {
    Object[]  elements;
    int       size;
    Object    get(int index)           { … }
    void      add(Object element)  { … }
}
```

Kompilatorn tar hand om resten

Sparar minne… men List<int> då?

Wrappers 1: Intro

jonkv@ida

- Hur skapar vi en listklass som accepterar heltal?

Implementera om strukturerna?

```
public class IntArrayList {
    private int[] elements;
    private int size;
    public int get(int index) {
        return elements[index];
    }
    void add(int element) { … }
}
```

Repetera för 8 primitiva typer
och för ***varje datastruktur***

Kan göras – finns färdiga bibliotek
på nätet (GNU Trove, …)

Använd wrappers!

```
public class Integer {
    private final int value;
    public Integer(int value) {
        this.value = value;
    }
    public int intValue() {
        return value;
    }
}
```

Repetera för 8 primitiva typer
➔ klart

Wrappers 2: Jämförelse

jonkv@ida

IntArrayList

Object header:	(data)
size	5
elements	

Object header:	(data)
length	8
[0]	12
[1]	14
[4]	20

ObjectArrayList

Object header:	(data)
size	5
elements	

Object header:	(data)
length	8
[0]	
[1]	
[4]	

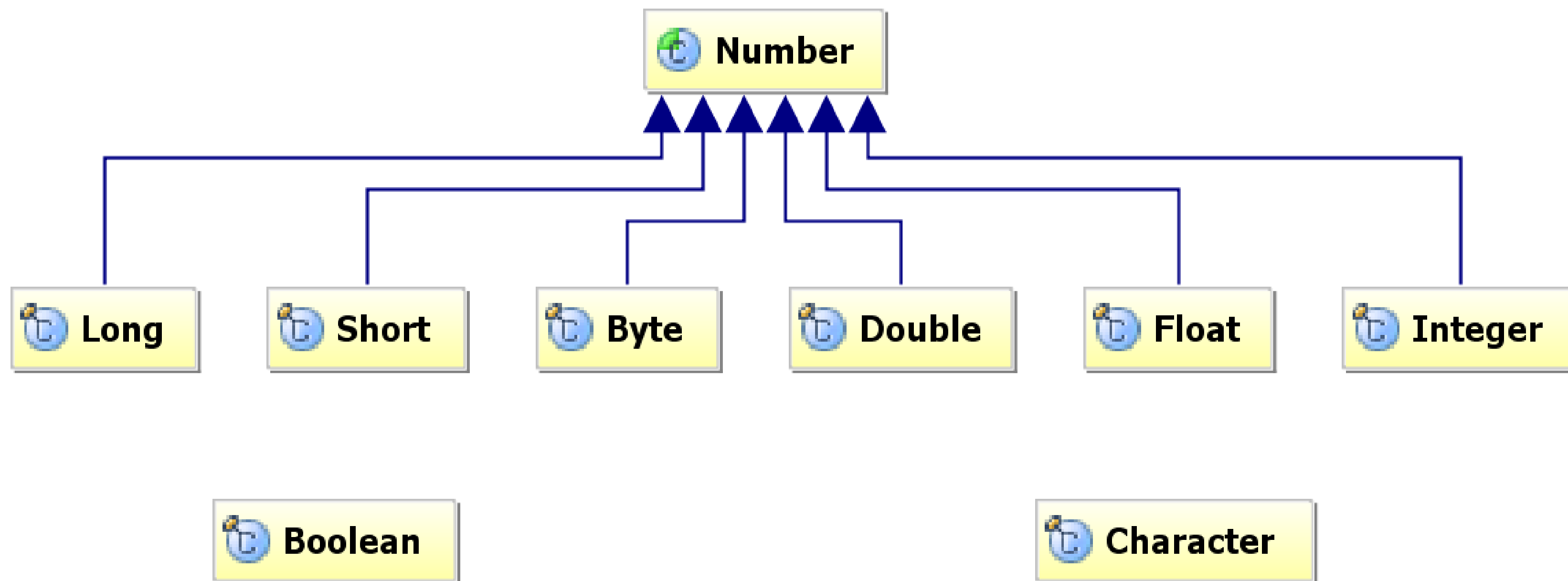
Object header:	(data)
value	12

Object header:	(data)
value	14

Wrappers ger mindre *andel* overhead om strukturerna är mer komplexa…

Wrappers 3: Wrapper-objekt

- Det finns en wrapper-klass för varje primitiv datatyp



Används bara om vi ska lagra i listor, objektpekare, …!

Använd int, double, boolean, … i normalfallet: Returvärden, parametrar

Wrappers 4: Skapa

jonkv@ida

- **Kan** skapas med vanlig konstruktor

```
List<Number> numbers = new ArrayList<>();
numbers.add(new Integer(10));
numbers.add(new Double(3.14));
```

- **Bättre**: Statiska **fabriksmetoden valueOf**()

```
numbers.add(Integer.valueOf(10));
numbers.add(Double.valueOf(3.14));
```

Sparar tid och minne: **Återanvänder** objekt för vanliga värden!

Integer-klassen har redan skapat objekt för heltalen -128 till +127

- Alternativ: **Autoboxing** (”paketinslagning”)

```
numbers.add(10);
numbers.add(3.14);
```

Kompilatorn:
”Vi behöver ett objekt men får ett heltal.
Stoppa in Integer.valueOf()!”

Wrappers 5: Hämta värdet

- För att hämta ut värdet: Anropa *type*Value()

```
Number intobj   = numbers.get(0);
int val0        = intobj.intValue();
```

```
double val1     = numbers.get(1).doubleValue();
```

- Alternativ: **Automatisk unboxing** ("öppna paket")

```
double val1 = numbers.get(1);
```

Kompilatorn:
"Vi behöver ett flyttal, så packa upp det."

Wrappers 6: Python?

- **Python** har inga primitiva typer, ***bara*** objekttyper
  - Trevligare på ytan – slipper bry sig om skillnaden

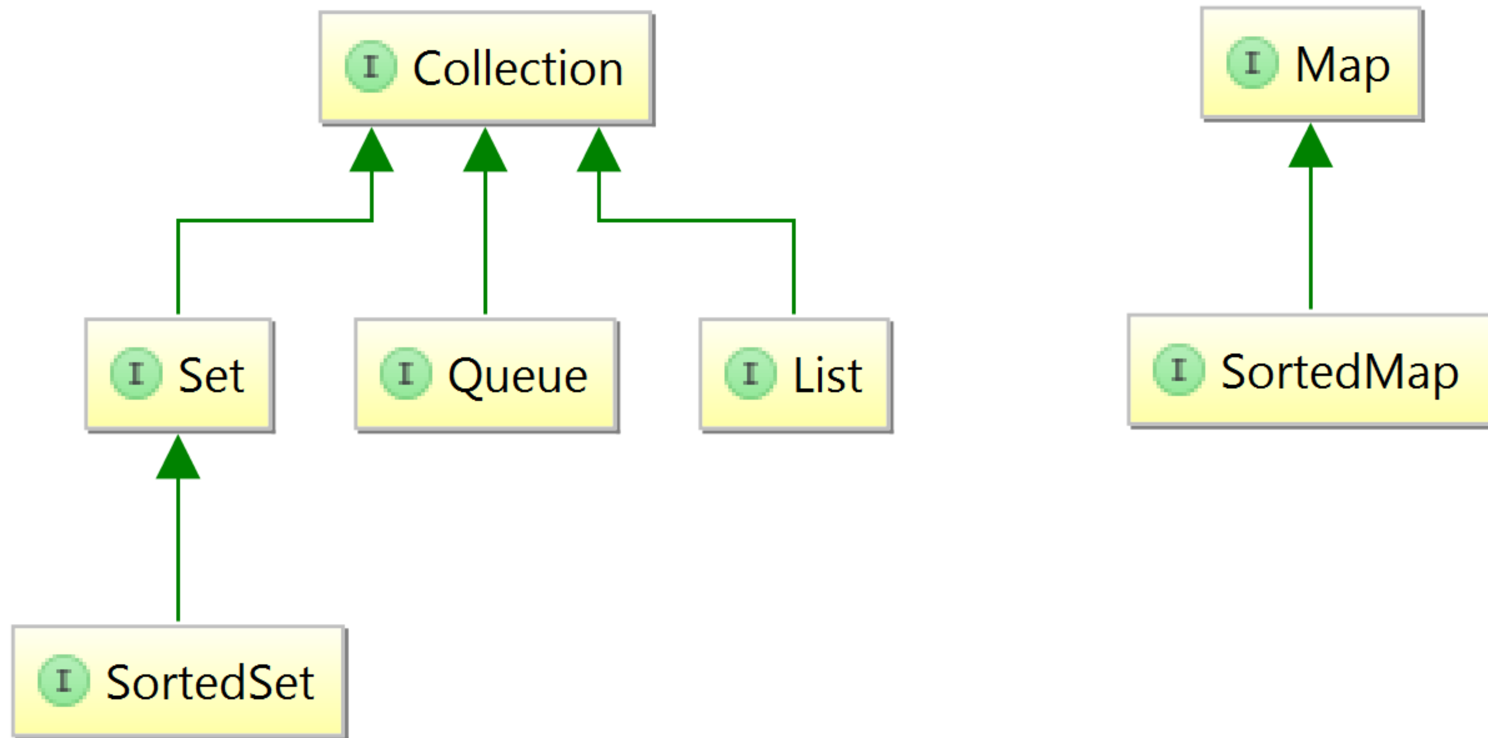
  - Vanliga heltal **är i princip wrapperobjekt**, har metoder!
    - >>> **x = 10**
    - >>> **x.bit_length()**
      4
    - >>> **x.__add__(10)**
      20

  - I Java: Kan ***välja*** mellan
    - Primitiva typer – inte objekt, vissa begränsningar
    - Wrapperklasser – är objekt, men långsammare, kräver mer minne

Java Collections Framework

Collections Framework 1: Intro

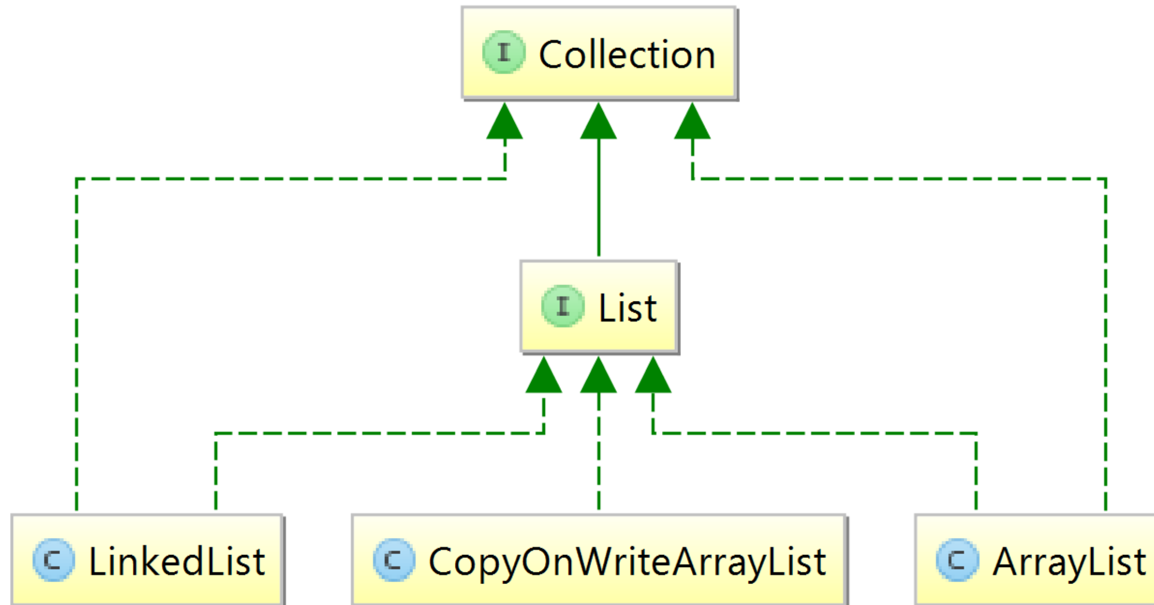
- **Collections Framework**: Många sammansatta datastrukturer
  - 1) Uppsättning **gränssnitt** för *listor*, *köer*, *mängder*, *mappningar*
  - De viktigaste:



Kom ihåg: Gränssnitt ger oss frihet att välja implementation!

Collections Framework 2: Intro

- 2) En uppsättning **implementationer**



- 3) Några **hjälpklasser**

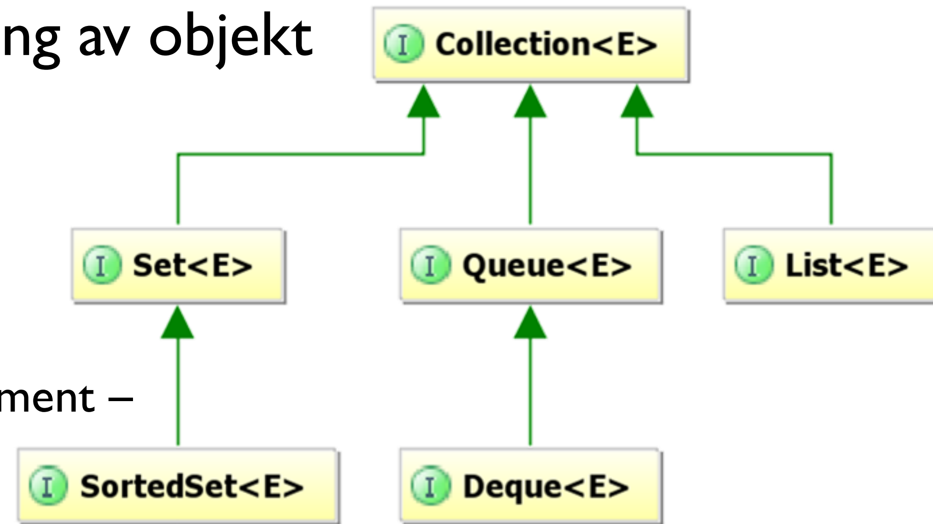
Objects

Arrays

Collections

Collection, List

- **Collection<E>**: Generell samling av objekt
  - *Vissa* samlingar är *ordnade*
    - Listor har ett *första* element – inte mängder
  - *Vissa* tillåter *dubblerade element*
    - Listor kan innehålla två identiska element – inte mängder



- List<E> **extends** Collection<E>
  - Alltid **ordnad**: Element har index
  - **Implementeras av**
    - ArrayList – Mycket vanlig, implementation liknar våra tidigare exempel
    - LinkedList – Ganska vanlig, andra egenskaper
    - Stack, Vector – Använd inte, ***deprecated***

List-funktionalitet

- Se websidorna:

För att **skapa en lista** används dessa klasser.

```
ArrayList       - klart vanligast, implementationen liknar föreläsningsexempel
LinkedList      - om man har specifika krav på snabb insättning/borttagning mitt i en lista
Stack           - gammal klass, används normalt inte
Vector          - gammal klass, används normalt inte
```

För att **lägga till eller ändra element**:

```
coll.add(element)
list.add(int index, E value)    - lägg till på given position; flytta resten av elementen
list.set(int index, E value)    - ersätt elementet på given position; storlek ändras ej
coll.addAll(coll2)              - lägg till element från en annan Collection, t.ex. en lista
list.addAll(int index, Collection c)
Collections.fill(list, element) - ersätter existerande objekt
Collections.replaceAll(list, element1, element2)
```

För att **plocka ut specifika element eller delar**:

```
list.get(index)
list.sublist(fromInclusive, toExclusive) - en del av listan
coll.toArray()                  - en Object[ ] med samma element
```

För att **testa innehållet / leta efter element**:

```
coll.isEmpty()
coll.size()
coll.contains(Object obj)
coll1.equals(Collection c2)             - exakt samma element?
coll1.containsAll(Collection c2)        - alla element i coll2 finns i coll1?
list.indexOf(Object needle)
list.lastIndexOf(Object needle)
Collections.binarySearch(list, needle, ...) - binärsökning i sorterad lista
```

För att **ta bort element**:

```
coll.clear()                            - tar bort alla element
coll.remove(Object)
list.remove(index)
```

För att **ändra listan på andra sätt**:

```
list.sort()                     - sortera, om elementen är Comparable (se föreläsning)
list.sort(comparator)           - sortera, annars
Collections.shuffle(list)       - flytta om alla elementen i listan
Collections.rotate(list, distance) - rotera listan (flytta element d positioner)
```

Queue

- Queue<E> **extends** Collection<E>
  - Kan lägga till i *slutet* av kön
  - Kan ta bort i *början* av kön
  - **Implementeras av**
    - ArrayDeque, LinkedList – grundläggande köstrukturer
  - **Varianter**
    - Deque – Även "på andra sidan" (Double-Ended QUEue)
    - PriorityQueue – har element i sorterad ordning

	Vad händer om det är omöjligt?	
Operation	*Signalerar fel – exception*	*Returnerar speciellt värde*
Stoppa in element	add(e)	offer(e)
Ta bort element	remove()	poll()
Se på första elementet	element()	peek()
	När man "vet" att det ska gå bra – annars är ngt fel	**När man vill *prova* om det går bra**

Sorting, Comparable, and Comparator

Sorting 1: Element Class

Element class:

```
public final class Person {
    public final String firstname, lastname, phone;
    public Person(String firstname, String lastname, String phone) {
        this.firstname      = firstname;
        this.lastname       = lastname;
        this.phone          = phone;
    }
    public String toString() {
        return  firstname + " " + lastname + " (" + phone + ")";
    }
    public boolean equals(Object other) {
        // ... type checking, casting, ...
        ...         firstname.equals(other.firstname) &&
                    lastname.equals(other.lastname) && phone.equals(other.phone);
    }
}
```

Sorting 2: Comparing Elements

- We can sort people:
  - By name
  - By phone number
  - By age
  - By …
- To tell the sort() method which order to use:
  - Give it a **comparison function**
  - Represented as an **object** with a **comparison method**

```
public interface Comparator<T> {
    int compare(T obj1, T obj2);
    boolean equals(Object obj);
}
```

Returns -1, 0 or 1: obj1 should be before, equal to, or after obj2

Sorting 3: Comparing Elements

jonkv@ida

- Now we need some **person comparators**:

```
public interface Comparator<T> {
    int compare(T obj1, T obj2);
    boolean equals(Object obj);
}
```

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


    public int compare(Person p1, Person p2) {
        int len1 = p1.lastname.length();
        int len2 = p2.lastname.length();
        if (len1 > len2) return 1;
        else if (len1 < len2) return -1;
        else return 0;
    }
}
```

Compares **Person** objects…

compare() needs two **Person** parameters!

Sorting 4: Comparing Elements

- Another **person comparator**:

```
public interface Comparator<T> {
    int compare(T obj1, T obj2);
    boolean equals(Object obj);
}
```

```
public class PhoneComparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        return p1.phone.compareTo(p2.phone);
    }
}
```

Phone number is a string
String has a compareTo() method

Sorting 5: Sorting with a Comparator

- Sorting a list:

```
List<Person> students = course.getParticipants();
Collections.sort(students, new NameLengthComparator());
```

- Technicalities:
  - `public static <T> void sort(List<T> list, Comparator<? super T> c);`

If T is any type…	…then we can sort a list of T elements…	…given a comparator that can compare T objects, and possibly superclasses as well

Sorting 6: Natural Order

- Some types have a "natural" ordering
  - To support this: Let the type implement **Comparable**<T>

Any **Person** can compare itself to other **Person** objects

```
class Person implements Comparable<Person> {
    public final String firstname, lastname, phone;
    public Person(String firstname, String lastname, String phone) {
        …
    }

    public int compareTo(Person other) {
        if (!lastname.equals(other.lastname)) {          // Different last names?
            return lastname.compareTo(other.lastname);   // … then compare them.
        } else {
            return firstname.compareTo(other.firstname); // Else use first names.
        }
    }
}
```

Calls String.compareTo(…), which returns –1 or 0 or 1

Sorting 7: Sorting using Natural Order

- Sorting:

```
class Person extends Comparable<Person>;
List<Person> students = course.getParticipants();
Collections.sort(students);
```

This variation of sort()
requires elements that can compare *themselves*

- Technicalities:
  - ```
    public static <T extends Comparable<? super T>> void sort(List<T> list)
    ```

If T is something that can be compared to itself, and possibly to supertypes…

…then we can sort a list of T elements
    ```

Att iterera över objektsamlingar

Iteration 1: Med index

- Hur kan man **iterera över alla element** i en samling?
  - Med en array eller ArrayList fungerar **index** utmärkt:

```
static boolean contains(List<?> haystack, Object needle) {
    for (int i = 0; i < haystack.size(); i++) {
        if (haystack.get(i).equals(needle)) return true;
    }
    return false;
}
```

Iteration 2: Med index

- Hur kan man **iterera över alla element** i en samling?
  - Med en LinkedList fungerar index, men *långsamt*:

```
static boolean contains(List<?> haystack, Object needle) {
    for (int i = 0; i < haystack.size(); i++) {
        if (haystack.get(i).equals(needle)) return true;
    }
    return false;
}
```

```
Node<E> get(int index) {
    Node<E> x = first;
    for (int j = 0; j < index; j++)
        x = x.next;
    return x;
}
```

get(0): Gå genom element j=0
get(1): Gå genom element j=0, 1
get(2): Gå genom element j=0, 1, 2
get(3): Gå genom element j=0, 1, 2, 3
get(4): Gå genom element j=0, 1, 2, 3, 4
…

Iteration 3: Utan index

- Hur kan man **iterera över alla element** i en samling?
  - Med en generell Collection (eller Set) **finns inga index**!

```
static boolean contains(Collection<?> haystack, Object needle) {
    for (int i = 0; i < haystack.size(); i++) {
        if (haystack.get(i).equals(needle)) return true;
    }
    return false;
}
```

(In the code, `get(i)` is struck through.)

Vi är egentligen inte intresserade av index!

Vi vill veta:

Finns det fler element?
Vad är nästa element?

Iteration 4: Iterator

- Lösning: Gränssnittet **Iterator**

```
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    …
}
```

Finns det fler element?

Vad **är** nästa element?

```
static boolean contains(Iterator<?> haystack, Object needle) {
    while (haystack.hasNext()) {
        Object element = haystack.next();
        if (element.equals(needle))
            return true;
    }
    return false;
}
```

Specialsyntax:
Iterator för *vilken elementtyp som helst*

Vet inte vilken elementtyp… men:
Det är i alla fall någon sorts Object!

**Har en egen implementation för varje datastruktur,
eget sätt att hålla reda på "var vi är just nu"**

Iteration 5: Iterable och for(each)-loopar

- Gränssnittet **Iterable** säger att objektet **kan ge oss** en iterator

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

- Implementeras av Collections och arrayer, kan användas i **for**

```
static boolean contains(Collection<?> haystack, Object needle) {
    for (Object element : haystack) {
        if (element.equals(needle)) return true;
    }
    return false;
}
```

```
int sum(int[] numbers)
    for (int i : numbers) { … }
}
void printAll(List<String> greetings) {
    for (String str : greetings) { … }
}
```

Loopen anropar haystack.iterator()

Anta haystack är LinkedList

LinkedList.iterator() returnerar new LinkedListIterator(), som vet hur man effektivt itererar över länkade listor

Iteration 6: ConcurrentModificationException

- Om man **ändrar** en samling medan man **itererar** över den:
  - **ConcurrentModificationException**
  - Har inget att göra med *multitrådning*!

```
static boolean contains(Iterable<?> haystack, Object needle) {
    for (Object element : haystack) {
        if (element.equals(needle)) return true;
        if (...) {
            ändra i haystack ...
        }
    }
    return false;
}
```

Iteration 7: Ta bort element

jonkv@ida

- Iterator har en **tredje metod**

```
package java.util;
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void remove();
}
```

Ta bort elementet
som just returnerades av next()

```
static void removeAllCopies(Iterable<String> haystack, Object needle) {
    Iterator<String> iter = haystack.iterator();
    while (iter.hasNext()) {
        String element = iter.next();
        if (element.equals(needle)) {
            iter.remove();
        }
    }
}
```

Tas bort **via iteratorn** → ingen ConcurrentModificationException

Iteration 8: ListIterator

- ListIterator har fler metoder
  - **Ändra** senast returnerade element
  - **Lägg till** element innan senast returnerade element
  - Gå **framåt och bakåt** i listan

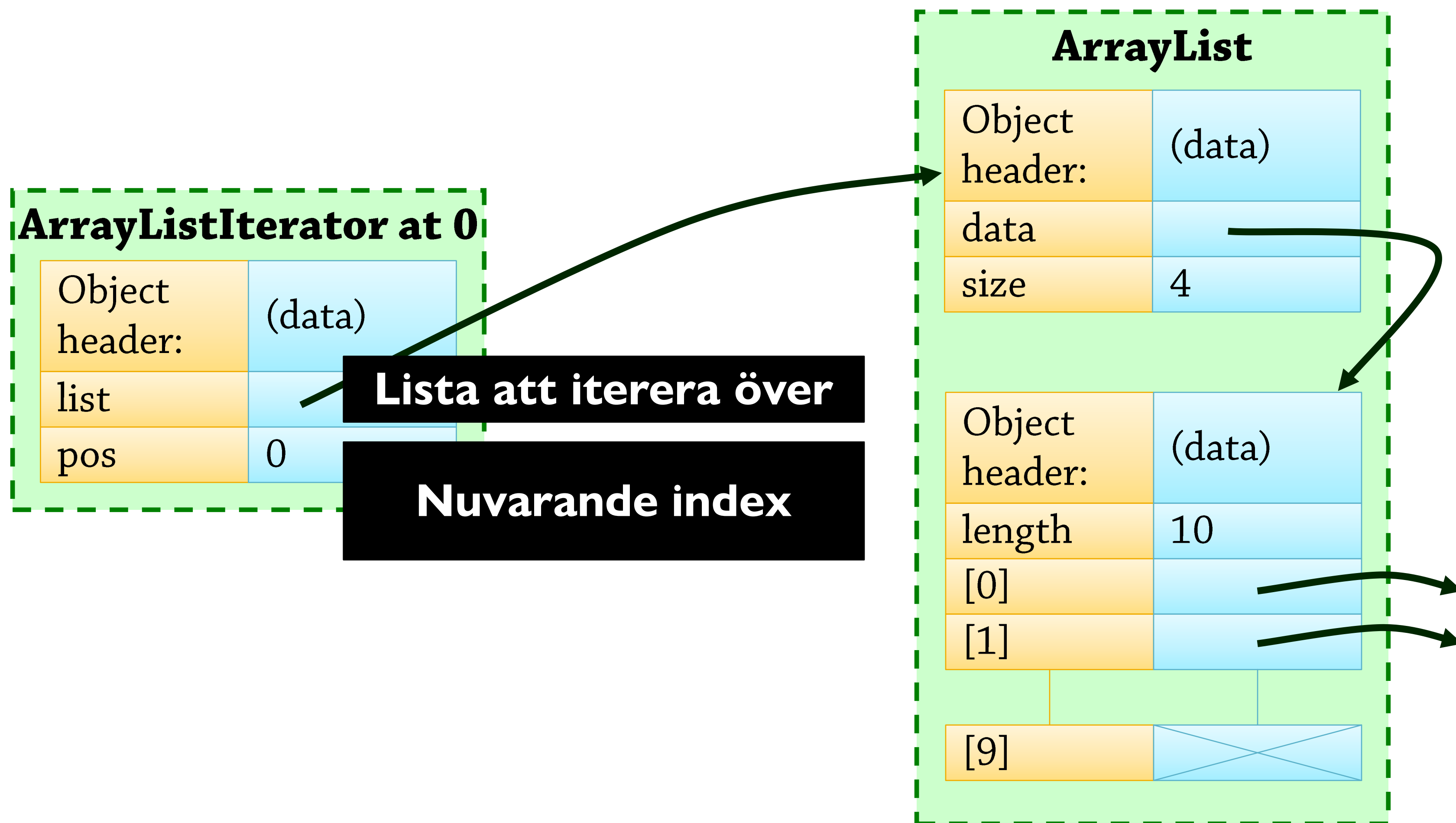
```
List<String> greetings = new ArrayList<>();
ListIterator<String> iter = greetings.listIterator();
```

En egen iterator (eller två)

Iterator för ArrayList 1

jonkv@ida

- Vad behöver en iterator för ArrayList känna till?



Iterator för ArrayList 2

```
public class ArrayList<E> implements List<E> {
    public Iterator<E> iterator() {
        return new ArrayListIterator<E>(this);
    }
    private class ArrayListIterator<E> implements Iterator<E> {
        private ArrayList<E> list;              // Which list should we iterate over?
        private int pos = 0;                    // How many element have we returned so far?
        public ArrayListIterator(final ArrayList<E> list) { this.list = list; }

        public boolean hasNext() {
            return (pos < list.size());
        }

        public E next() {
            final E element = list.get(pos);
            pos++;
            return element;
        }
    }
}
```

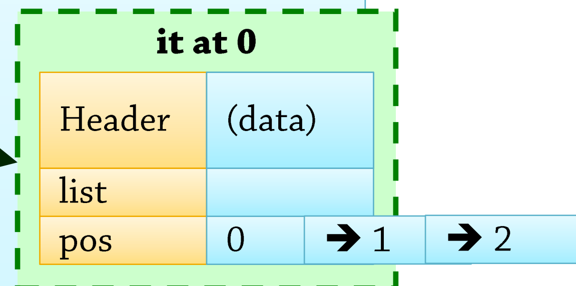
Finns det flera element?

Vad är nästa element?

Iterator för ArrayList 3

Anta list=["hello","world"].
Iterator it = list.iterator();

```
while (it.hasNext()) {
  System.out.println(it.next());
}
```



Första iterationen: it=[list,0].
Testar: it.hasNext() ➔ 0 < list.size() ➔ returnerar true
Anropar: it.next() ➔ sätter pos = 1, returnerar "hello".

Andra iterationen: it=[list,1].
Testar: it.hasNext() ➔ 1 < list.size() ➔ returnerar true
Anropar: it.next() ➔ sätter pos = 2, returnerar "world".

Tredje iterationen: it=[list,2].
Testar it.hasNext() ➔ 2 < list.size() ➔ returnerar false.
Avslutar loopen.

OBS: Själva listan ändras aldrig!

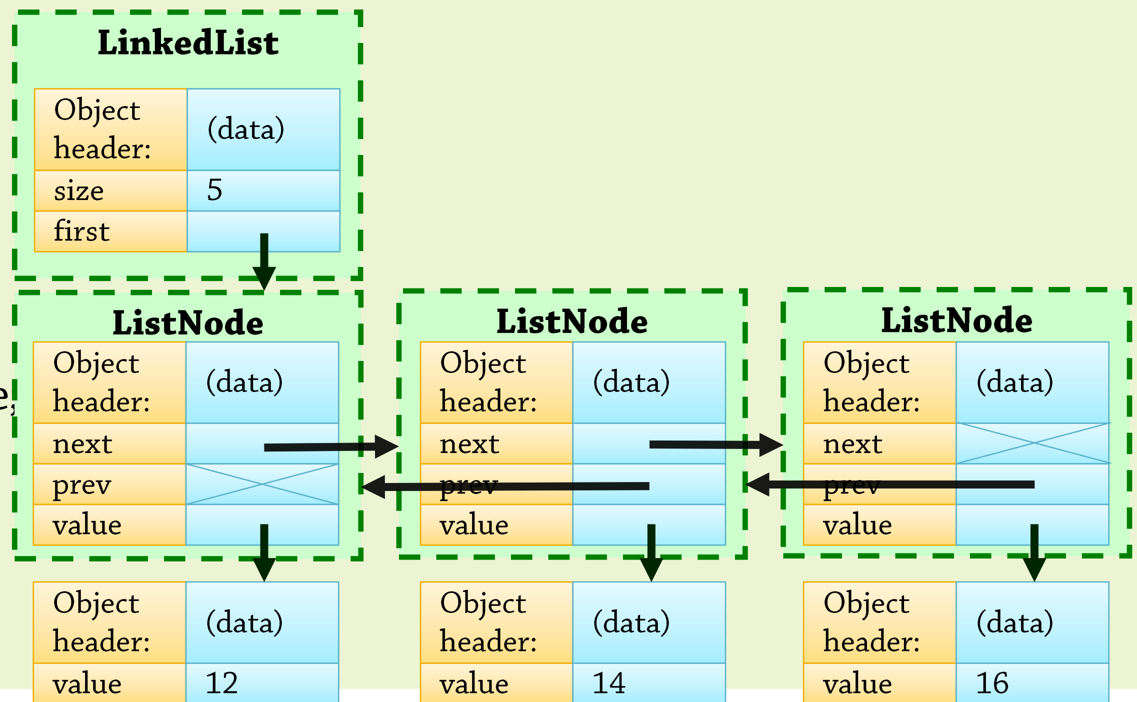
Iterator för LinkedList


```
public class LinkedList<E> implements List<E> {
    public Iterator<E> iterator() {
        return new LinkedListIterator<E>(this);
    }
    private class LinkedListIterator<E> implements Iterator<E> {
        private ListNode<E> node;            // Which list node are we currently at?

        public LinkedListIterator(final LinkedList<E> list) { this.node = list.first; }

        public boolean hasNext() {
            return node != null;
        }

        public E next() {
            final E element = node.value;
            node = node.next;
            return element;
        }
    }
}
```



Iterator för textfiler

- En iteratorklass för textrader i en fil:

```
public class LineIterator implements Iterator<String> {
    private final BufferedReader reader;
    public LineIterator(String filename) { … }

    public boolean hasNext()          { return !reader.eof(); }
    public String next()              { return reader.readLine(); }
    …
}
```