Reliable Data Services: **Abstract Data Types**

- Abstract Data Types
- The Java Collection Framework
- Generic Collections
- Iterators

I. Abstract Data Types

Remember the Public Car Park?

- Basically, a car park is like an array of Cars:

```
Car[] carpark = new Car[100];
```

- It can be used to park a car:

```
carpark[25] = myMini;
```

- And a car can be removed from it:

```
Car myCar = carpark[25];
carpark[25] = null;
```

But how do you

- *find an empty slot?*
- *find out whether there are any empty slots left?*
- *ensure that cars are only parked in empty slots?*
- *ensure that cars are not removed from empty slots?*
- *ensure that a car is not taken out without freeing its slot?*

Hide it in a CarPark Class

- Define a `CarPark` class
- Make the array private
- Define public access methods
  - that follow a filling strategy
  - ensure that cars are parked and removed according to the rules

You still have to decide on the rules:

- *How can a car be added?*
- *What happens if the CarPark is full?*
- *How can a car be identified for removal?*
- *What happens if the identified slot is empty?*

Define External Rules as Interface
(Interaction Syntax)

```
interface IndexedCarParkType {
  public int add(Vehicle vh) throws FullException;
                                  // yields index
  public int search(Vehicle vh)
                         throws NotFoundException;
                                  // uses model object
  public Vehicle show(int index)
       throws NotFoundException,IllegalIndexException;
  public void remove(int index)
       throws NotFoundException, IllegalIndexException;
  public int carCount();
}
```

Implementation using an Array

```
class CarPark implements IndexedCarParkType {
// private attributes
private Car[] cars;          // data structure
private int carCount = 0;    // semantic helper

// Default constructor, fixed size
public CarPark() {
   cars = new Car[50];
}
// constructor with size parameter
public CarPark(int size) {
   cars = new Cars[size];
}
```

```
// add if not full
public int add(Car c) throws FullException {
   if (carCount >= cars.length)
           throw new FullException();
   int empty = findNextFree();
   cars[empy] = c;
   carCount++;
   return carCount-1;
}
private int findNextFree() {
   for (int i=0; i<cars.length; i++)
        if (cars[i] == null) return i;
   throw new InconsistentListException();
}

public int carCount()
{  return carCount; }

// read if existing
public Vehicle show(int index)throws NotFoundException,
                                       IllegalIndexException {
    try {
        if (cars[index]==null)throw new NotFoundException();
        return cars[index];
    } catch (ArrayIndexOutOfBoundsException ex)
   { throw new IllegalIndexException();     }
}
```

```
public class FullException
extends Exception {}
```

```
public class NotFoundException
extends Exception {}
```

```
public class InconsistentListException
extends RuntimeException {}
```

but mind the order of exception detection...

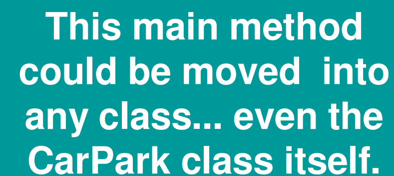
```
  public int search(Car car)  // model object
                               throws NotFoundException {
     if (car==null) throw new NoObjectException();
     for (int i=0; i<carCount; i++)
        if (cars[i].equals(car)) return i; // found
     throw new NotFoundException(); // not found
  }


  public void remove(int index)
        throws IllegalIndexException, NotFoundException{
     try {
        if (cars[index] == null)
                throw new NotFoundException();
        cars[index] = null;
        carCount--;
     } catch (IndexOutOfBoundsException e)  {
        throw new IllegalIndexException();
     }


  }

} // end of class
```

```
class NotFoundException
extends Exception {}
```

```
class NoObjectException
extends RuntimeException{}
```

Use the CarPark Type

```
class UseCarPark {
   public static void main(String[] args) {
        CarPark smallCarPark = new CarPark(12);
        try {
          for (int i=0; i<13; i++)
               smallCarPark.add(new Car());
        } catch (FullException ex)
        { System.out.println("Car park is full, sorry!"); }
   }
}
```



Data Structures – Data Types

Concepts for storing data:

- Data structure:
  - **Object** which can be used to store data
  - Example: int[] dataStruct = new int[100];

- Data type:
  - **Type** for creating data structures
  - e.g. an array: **`int []`**
    instances are data structures for storing a series of integers

  - or some „record class “:

```
public class Address {
  String name;
  String firstName;
  int age;
  // …
}
```

instances are data structures for storing two Strings and an integer

Abstract Data Type

- Type (Class) containing
  - a private data structure
  - public methods for storing and retrieving data from it.

- An abstract data type implements and enforces a strategy for data storage and retrieval

Some Common Container Variants

- Indexed Container
  - Data retrieval via index
- Sorted Container
  - Comparable data
  - Order (predecessor, successor)
- Set
  - No data duplication
- Bag
  - Data duplication permitted
- LIFO Container
  - Last In First Out → Stack
- FIFO Container
  - First In First Out → Queue
- Associative Container
  - Data retrieval using keys

- *All Containers have rather similar export interfaces (interaction syntax)*
- *but different behaviour (container semantics)*

Generalized Collection Type
(Syntax Scheme)

- Type that stores a variable number of elements.

- Standard operations:
  - **add element**
  - **read / search for element**
  - **remove / delete element**
  - **number of entries**
  - **(max. number of entries)**

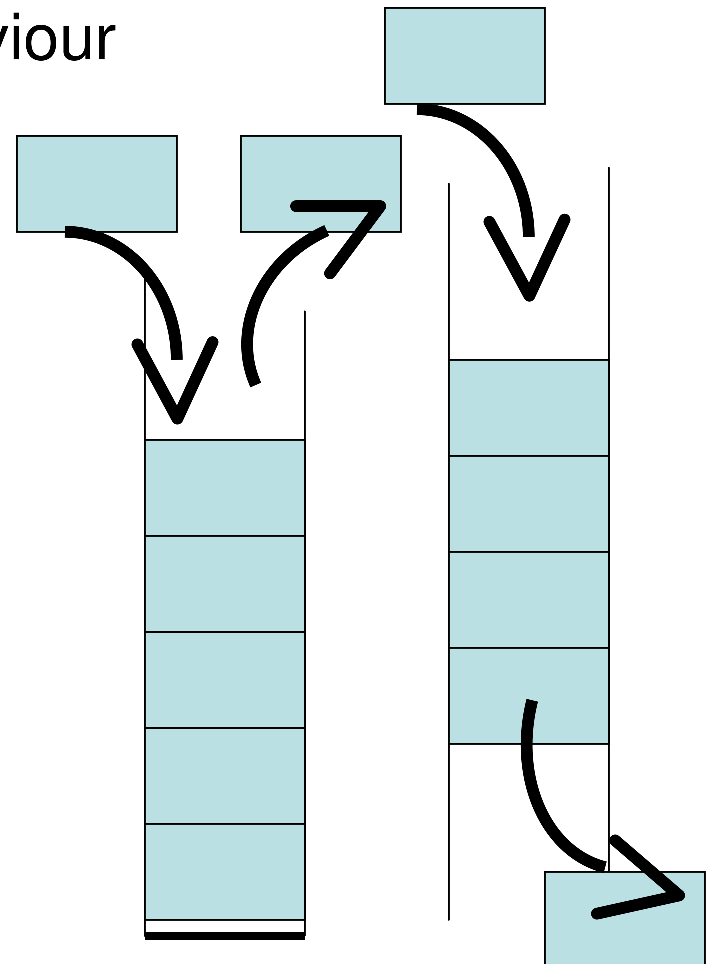
- Array is just one possible implementation technique
- Advantage: direct indexed element access
- Disadvantage:
  fixed length,
  removing from and adding into the middle are complex operations

Stack and Queue
(Stapel und Warteschlange)

- The most important collection schemes
- identical interface, different behaviour

```
interface Container {
  void add(Object o);
  Object remove()
     throws UnderflowExc;
}
```

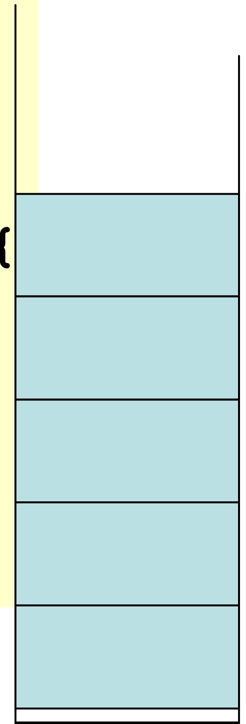
```
interface LiFo
    extends Container {}
interface FiFo
    extends Container {}
```



LiFo using Array

```
class StackArray implements LiFo {
  private Object[] stack = new Object[size];
  private int top = 0;

  public void add(Object o) {
    if (top == size) throw new OverflowExc();
    stack[top] = o;
    top++;
  }
  public Object remove() throws UnderflowExc {
    if (top==0) throw new UnderflowExc();
    top --;
    return stack[top];
  }
}
```

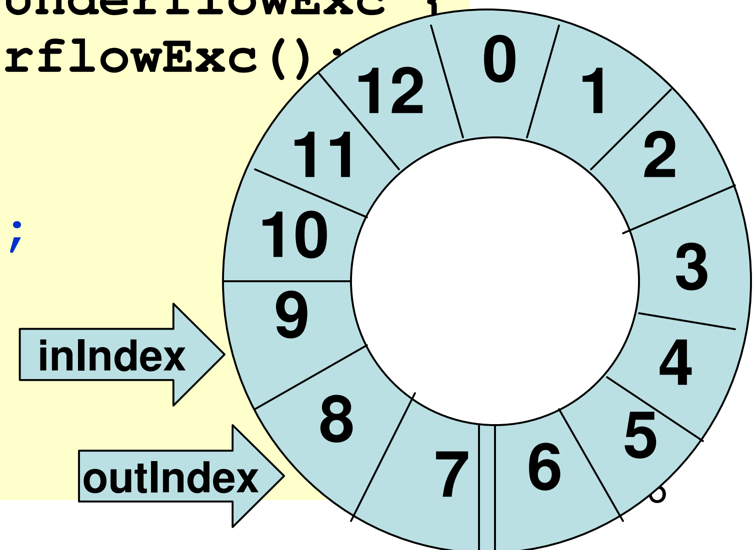
RuntimeException!



FiFo using Array

```
class QueueArray implements FiFo {
   private Object[] queue = new Object[size];
   private int inIndex=0, outIndex=0; count=0;

   public void add(Object o) {
     if (count>=size) throw new OverflowExc();
     queue[inIndex] = o;
     inIndex = (inIndex+1)%size; // modulo
     count ++;
   }
   public Object remove() throws UnderflowExc {
     if (count==0) throw new UnderflowExc();
     Object o = queue[outIndex];
     count --;
     outIndex = (outIndex+1)%size;
     return o;
   }
}
```

RuntimeException!



ADT Implementation Scheme

1. Ususally, implement a given interface
2. Define a private data structure
3. Implement required methods using that data structure.

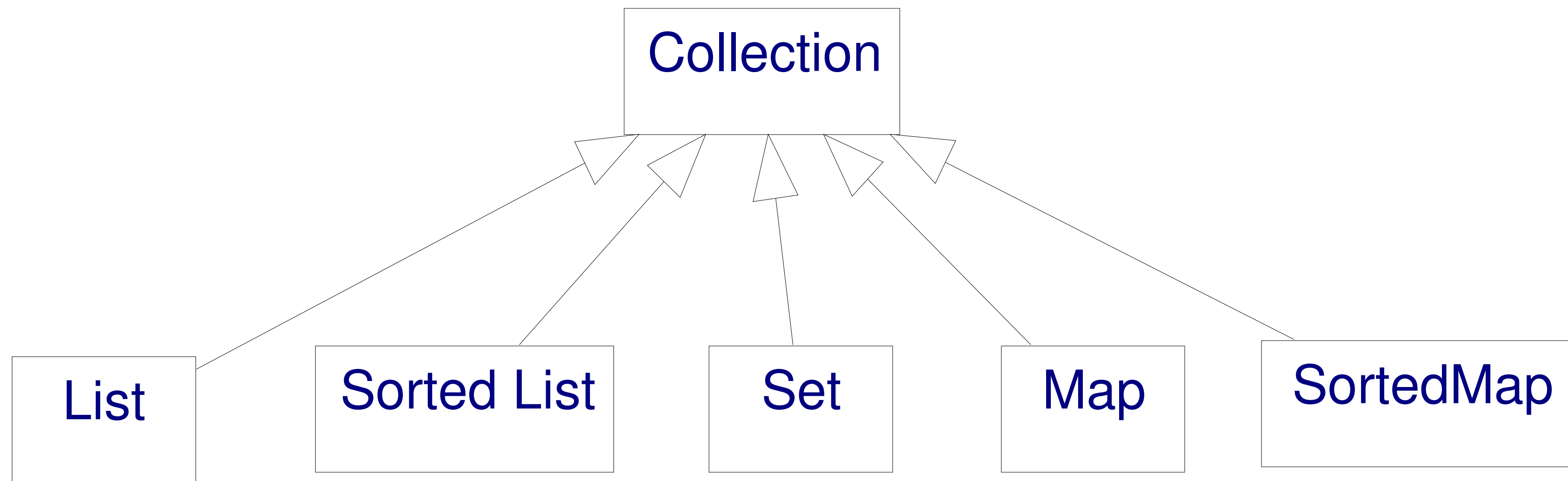
```
class QueueArray implements FiFo {
   private Object[] queue = new Object[size];
   private int inIndex=0, outIndex=0; count=0;

   public void add(Object o) {
     if (count>=size) throw new OverflowExc();
     queue[inIndex] = o;
     inIndex = (inIndex+1)%size; // modulo
     count ++;
   }
/ …
```

II. The Java Collection Framework

Professional Laziness: The Java Collection Framework

- Generalized data collections for universal use
- ready cut, with many features – and well tested
- different properties – for different applications
- about 50 classes – all following a uniform pattern.

The Java-Collections Framework: Interfaces



The List Interface

javax.naming.spi
javax.net
javax.net.ssl
javax.print
javax.print.attribute
javax.print.attribute.standard
javax.print.event
javax.rmi
javax.rmi.CORBA

Collection
Comparator
Enumeration
EventListener
Iterator
List
ListIterator
Map
Map.Entry
Observer
RandomAccess
Set
SortedMap
SortedSet

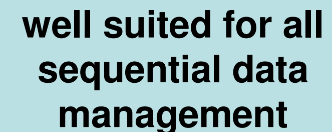
Classes
AbstractCollection
AbstractList
AbstractMap
AbstractSequentialList
AbstractSet
ArrayList
Arrays
BitSet
Calendar

Method Summary

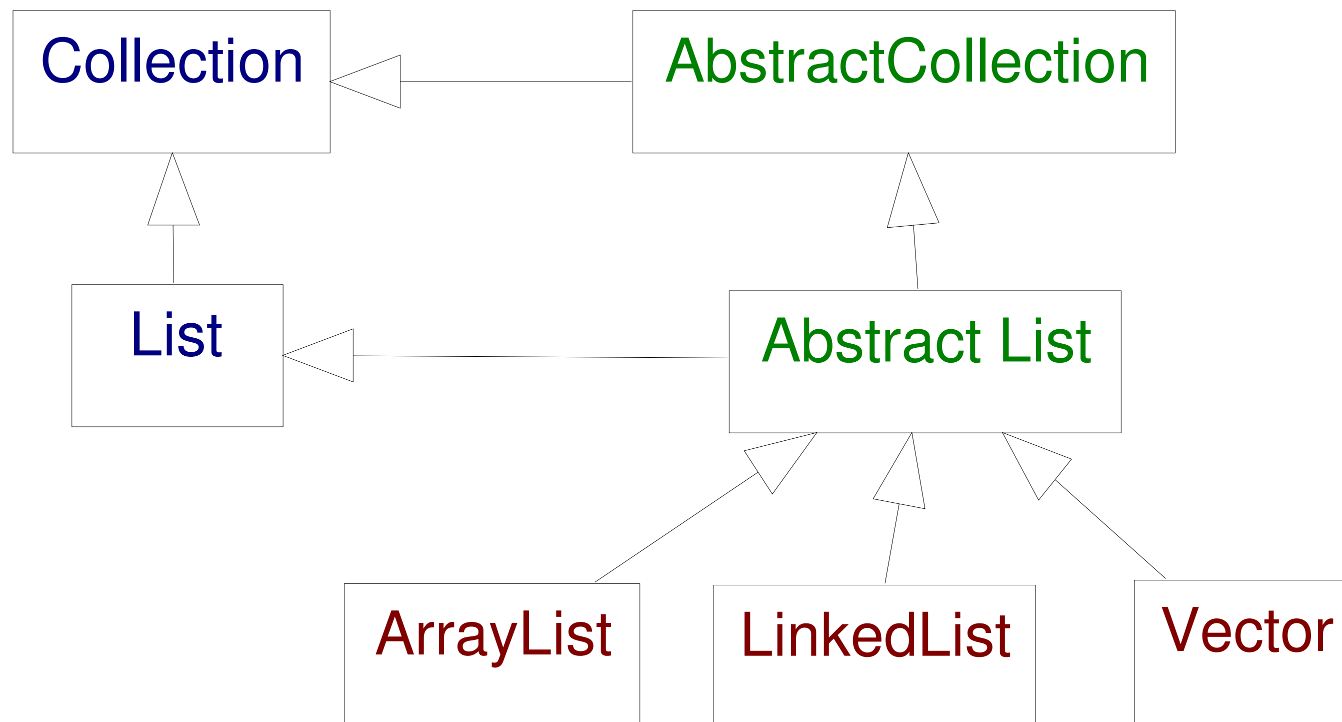
void	**add**(int index, Object element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**add**(Object o) Appends the specified element to the end of this list (optional operation).
boolean	**addAll**(Collection c) Appends all of the elements in the specified collection to the end of this list, in the orde that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection c) Inserts all of the elements in the specified collection into this list at the specified positic (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.
boolean	**containsAll**(Collection c) Returns `true` if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
Object	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if this does not contain this element.
boolean	**isEmpty**()

The List Interface

```
Interface List extends Collection{
  void add(Object o);
  void add(Object o, int index);
  void addAll(Collection c);
  void clear();
  boolean contains(Object o);
  void remove (Object o);
  Object remove(int index);
  int size();
  Object[] toArray();
  // ... and many other methods
}
```



The Java-Collections-Framework: List Implementations



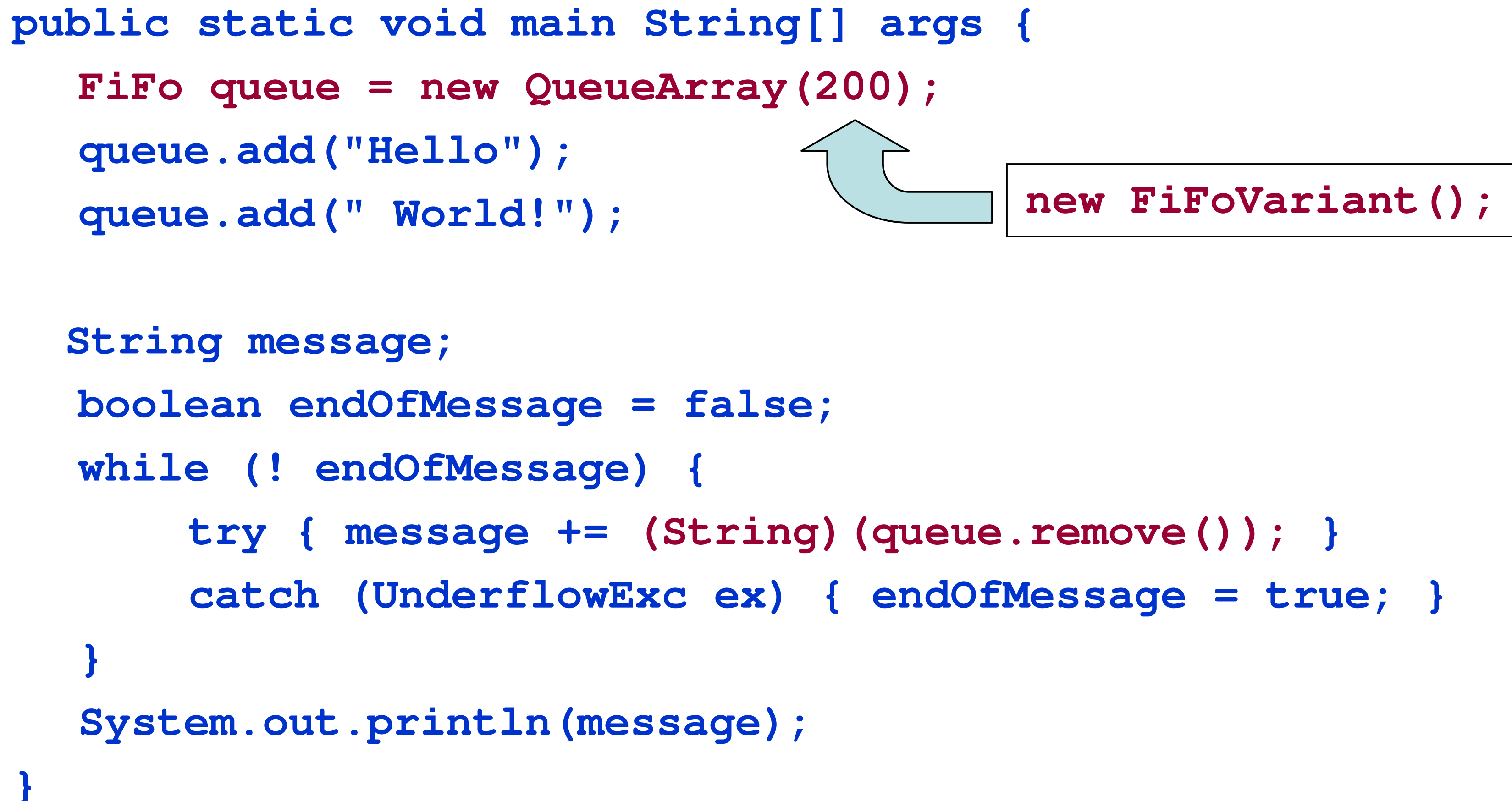
Using Collections: LiFo Container (Stack)

```
import java.util.*;  // Java Collections

class LifoVariant implements LiFo{
  private List list = new ArrayList();
              // or Vector or LinkedList
  public void add(Object obj) {
     list.add(obj,0); // add at front end
  }
  public Object remove() throws UnderflowExc{
     if (list.isEmpty()) throw new UnderflowExc();
     return list.remove(0); // remove from front end
  }
}
```

Using Collections: FiFo Container (Queue)

```
import java.util.*;  // Java Collections

class FifoVariant implements FiFo{
  private List list = new ArrayList();
                  // or Vector or LinkedList
  public void add(Object obj) {
      list.add(obj);  // add at back end (default)
  }
  public Object remove() {
      if (list.isEmpty()) throw new UnderflowExc();
      return list.remove(0);   // remove from front end
  }
}
```

Exchanging Implementations



III. Generic Containers

Type Safety

Type Safety is the modern programming language‘s response to **data misinterpretations**:

1. A variable can only hold values of its type
2. A value can only be used in operations defined for its type.

Universal transport types (Object...) **undermine** this appraoch, yet...

1. in Java, explicit type casting is secured (ClassCastException).
2. operational security is reinforced through the java syntax..

...But typesafe collections would be better...

Element Types

- You will implement a **String Stack** in your lab.
  - Following the same pattern, you could implement a Vehicle, Account or Ball stack.
  - Inheritance does not do the trick – you have to **rewrite the class**.
  - Each stack will be **typesafe** – i.e. the user can rely on the fact that all elements have the expected type.

- Standard collections usually work with the most general element type: **Object**.
  - can be used for any element type
  - no extra work on entry:

    ```
    queue.add("Hello");
    ```

  - "downcasting" required after reading an element

    ```
    ((String)queue).remove();
    ```

  - **not typesafe** (but type secured)

Type Parameterized Classes (Generic Classes)

- New in Java 5

- Instead of programming a stack for every element type
- \- you parameterize its definition with a formal element type

```
class Stack <Elementtype> { ...}
```

- You choose the actual element type on instantiation:

```
Stack <String> texts =
          new Stack <String>();
Stack <Account> acc =
          new Stack <Account>();
```

- `Stack <Elementtype>` is called a generic Class:
  a class which defines several types.

Type Instantiation

- If you don't use a parameter on instantiation, the element type is `Object.`
- This is called the "raw type"

```
Stack rawStack = new Stack();
```

- Two instances with different actual parameters are type-incompatible

```
Stack <String> texts =
          new Stack <String>();
Stack <Account> acc =
          new Stack <Account>();
texts = acc;  // TYPE ERROR
```

- Parameterized instances are compatible with the raw type,

```
raw = acc;
```

but handle with care, it has tricky implications!

Generic Standard Collections

- All Classes in the Java Collection Framework are generic.

- So, in order to define a StringStack, you use a parameterized instance of ArrayList as data structure:

```
class StringStack implements LiFo <String>{
      private ArrayList <String> list =
                         new ArrayList<String> ();
      ...
}
```

- This class does not implement the original LiFo interface, because the element type is no longer `Object`
- Define a generic interface instead:

```
interface LiFo <Elementtype> {  ...  }
```

IV: Loops and Iterators

Simple For Loop through a List

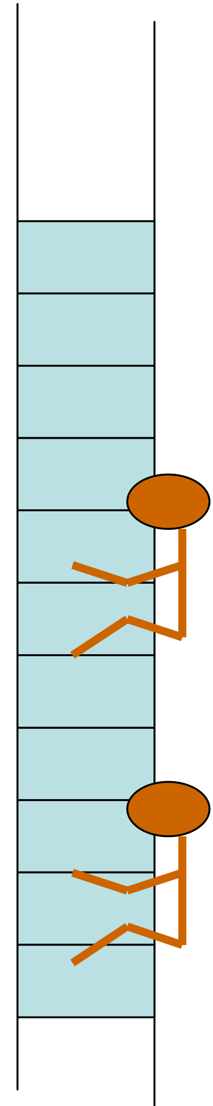
- Using the get() method, you can write a simple for loop just as with arrays:

```
public void print(List<Element> list)
{  for (int i=0; i<list.length(); i++)
   { System.out.println(list.get(i).toString(); }
}
```

Iterators

- Iterating through a collection independent of implementation type
- Iterator: "intelligent" object that lets you navigate through a collection, read, add and remove elements at any position
- Rembers position for the next call.

- Multiple iterators per collection possible.
- Like monkeys on a palm tree.
- Static function `iterator()` generates a "monkey" of type `Iterator` and launches it at the beginning of the List.
- Static function `listIterator()` generates a "monkey" of type `ListIterator` – which has lots of extra methods.



Iterator

public interface Iterator <T>

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 (optional operation).

Method Detail

Simple For-Loop using an Iterator

```
public void print(List<Element> list)
{   for (Iterator monkey = list.iterator();
          monkey.hasNext();    )
    { System.out.println(monkey.next().toString() }
}
```

Type Safeness: Generic Iterator

```
public void enter(List<Element> list, Element elem)
{   for (Iterator<Element> monkey = list.iterator();
          monkey.hasNext();    )
    { System.out.println(monkey.next().toString() }
}
```

ListIterator

public interface ListIterator<T> extends Iterator <T>

Method Summary	
void	**add**(Object o) Inserts the specified element into the list (optional operation).
boolean	**hasNext**() Returns `true` if this list iterator has more elements when traversing the list in the forward direction.
boolean	**hasPrevious**() Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
Object	**next**() Returns the next element in the list.
int	**nextIndex**() Returns the index of the element that would be returned by a subsequent call to `next`.
Object	**previous**() Returns the previous element in the list.
int	**previousIndex**() Returns the index of the element that would be returned by a subsequent call to `previous`.
void	**remove**() Removes from the list the last element that was returned by `next` or `previous` (optional operation).
void	**set**(Object o) Replaces the last element returned by `next` or `previous` with the specified element (optional operation).

Sorted List Entry using a ListIterator

- Let `list` be a (descending) sorted list, and `elem` an element to be entered.
- Iterator-Functions:
  - `next()` "climbs" one step and yields the element read
  - `nextIndex()` yields the next index without "climbing"
  - `hasNext()` checks for the end of the list

```
public void enter(List<Element> list, Element elem)
{   ListIterator monkey = list.listIterator();
    while (monkey.hasNext()
           && monkey.next().greater(elem))
    { */ just climb on */ }
    list.add(elem, monkey.nextIndex()-1);
    /* monkey.add(elem) would add it "too late" */
}
```

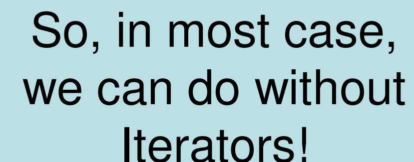
Java 5 Simplification
The For-In-Loop

- Remember the for-in-loop for arrays?

```
String[]items; ...
for (String thing : items)
    System.out.println(thing);
```

- The same kind of loop works for Collections!

```
List<String> assets; ...
for (String thing : assets)
     System.out.println(thing);
```



That's enough!

...but that was only the theory,

you need to gain practical experience with collections!