

# Agenda

## Recap loops: for-each, while

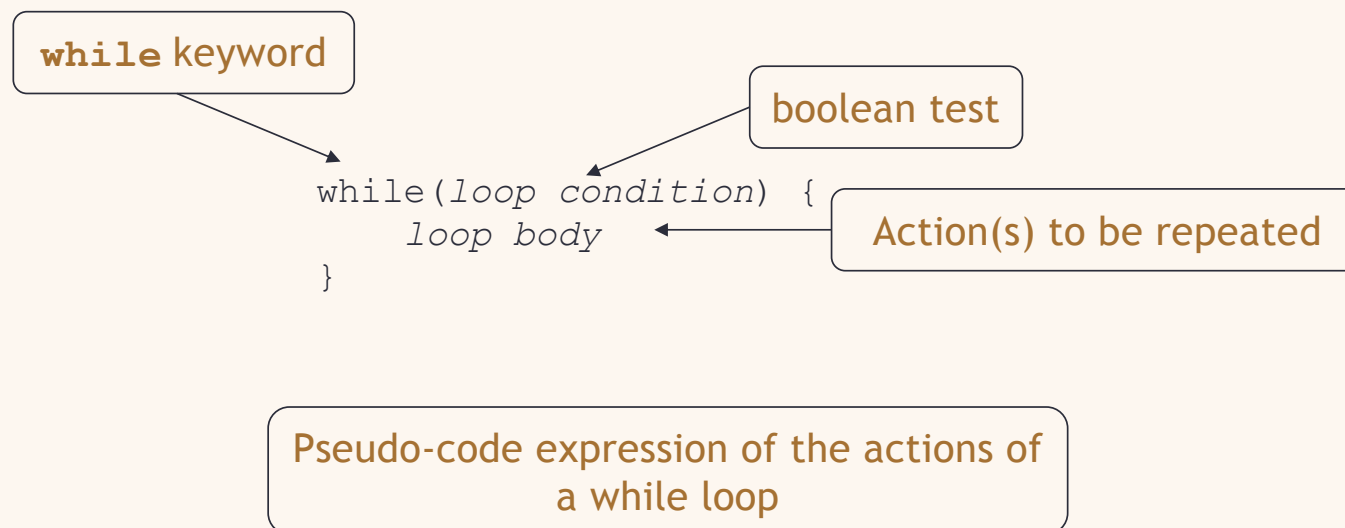
## Loop: Iterator

## Chatbot: Techsupport project

## String: equality vs identity



## While loop pseudo code



while we wish to continue, do the things in the loop body

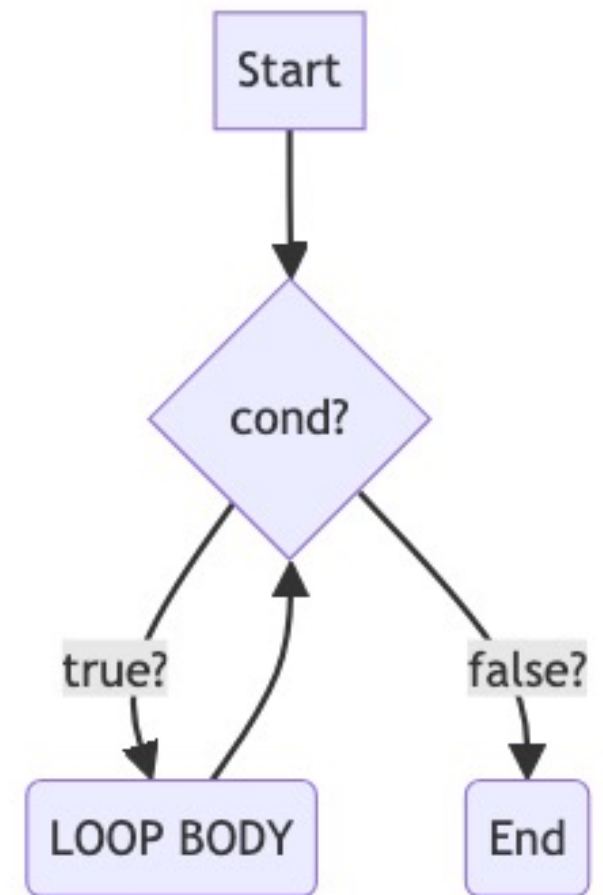
## for-each versus while

for-each:

- easier to write.
- safer: it is guaranteed to stop.

while:

- we don't *have to* process the whole collection.
- doesn't even have to be used with a collection.
- take care: could create an *infinite loop*.



## Elements of the while loop

We have declared an index variable.

The condition must be expressed correctly.

We have to fetch each element.

The index variable must be incremented explicitly.

## Search tasks are indefinite

Consider: searching for your keys.

You cannot predict, *in advance*, how many places you will have to look.

Although, there may well be an absolute limit – i.e., checking every possible location.

You will stop when you find them.

‘Infinite loops’ are also possible.

- Through error or the nature of the task.

## The while loop

A for-each loop repeats the loop body for every object in a collection.

- Sometimes we require more flexibility than this.
- The while loop supports flexibility.

We use a **boolean condition** to decide whether or not to keep iterating.

This is a *very* flexible approach.

Not tied to collections.

## For-each loop equivalent

```
/**
 * List all file names in the organizer.
 */
public void listAllFiles()
{
    int index = 0;
    while(index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
        index++;
    }
}
```

Increment *index* by 1

while the value of *index* is less than the size of the collection,  
get and print the next file name, and then increment *index*

## Iterator and `iterator()`

Collections have an `iterator()` method.

This returns an `Iterator` object.

`Iterator<E>` has methods:

- `boolean hasNext()`
- `E next()`
- `void remove()`

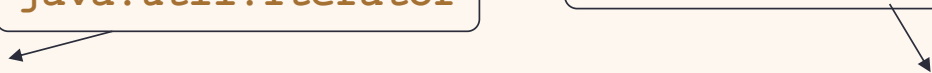
```
Import java.util.Iterator
```

→ We use a while loop but we do not need to take care of the index

## Using an Iterator object

**java.util.Iterator**

returns an **Iterator** object



```
Iterator<ElementType> it = myCollection.iterator();
while(it.hasNext()) {
    call it.next() to get the next object
    do something with that object
}
```

---

```
public void listAllFiles()
{
    Iterator<String> it = files.iterator();
    while(it.hasNext()) {
        System.out.println(it.next());
    }
}
```

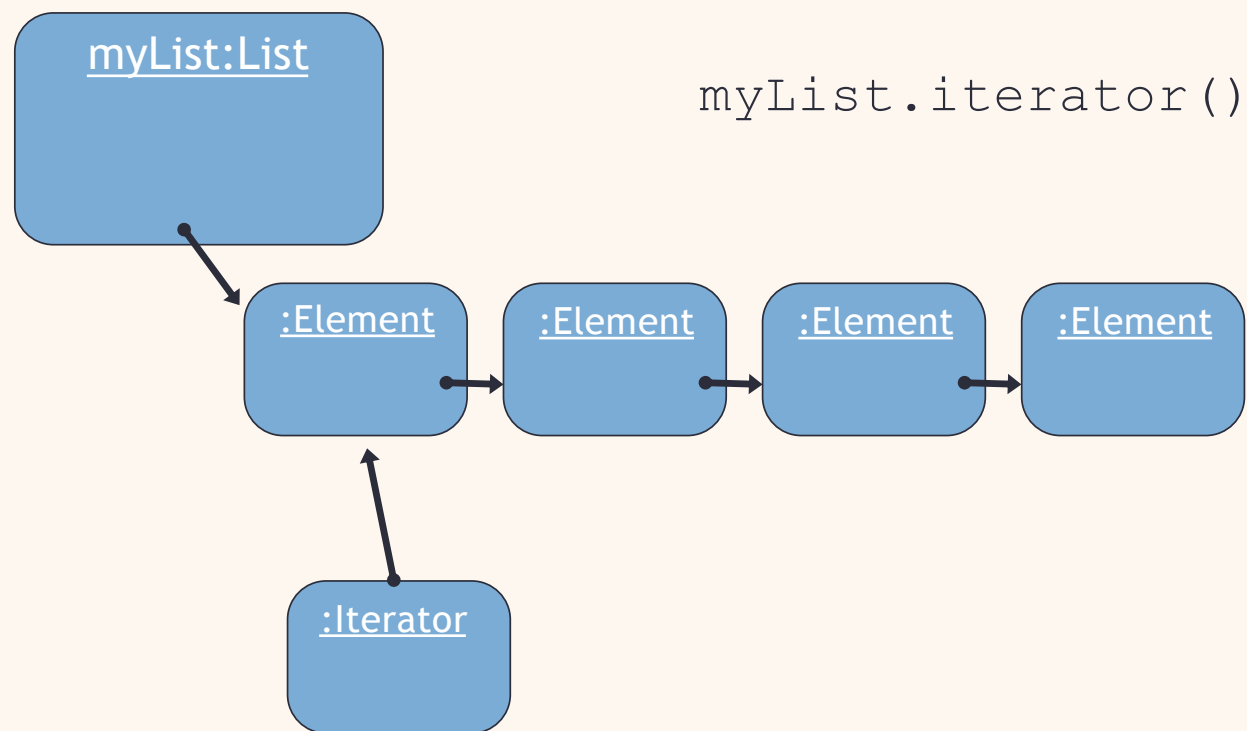
---

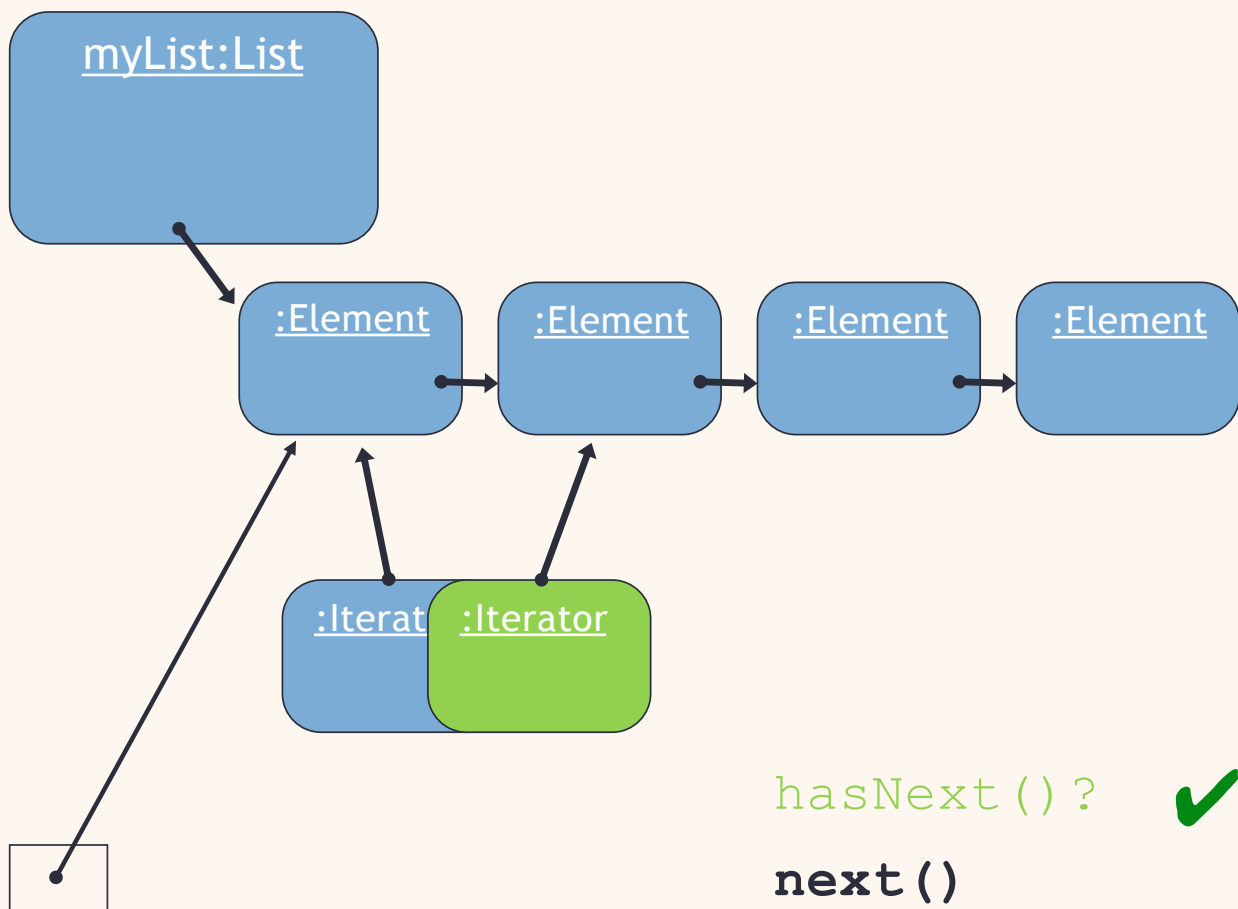
Iterator Functionality: Iterator object allows iteration through a collection, returning the next item on each call to the `next()` method.

Iterator Usage in Loops: Iterators are used in loops to access elements of a collection, eliminating the need for explicit index management.

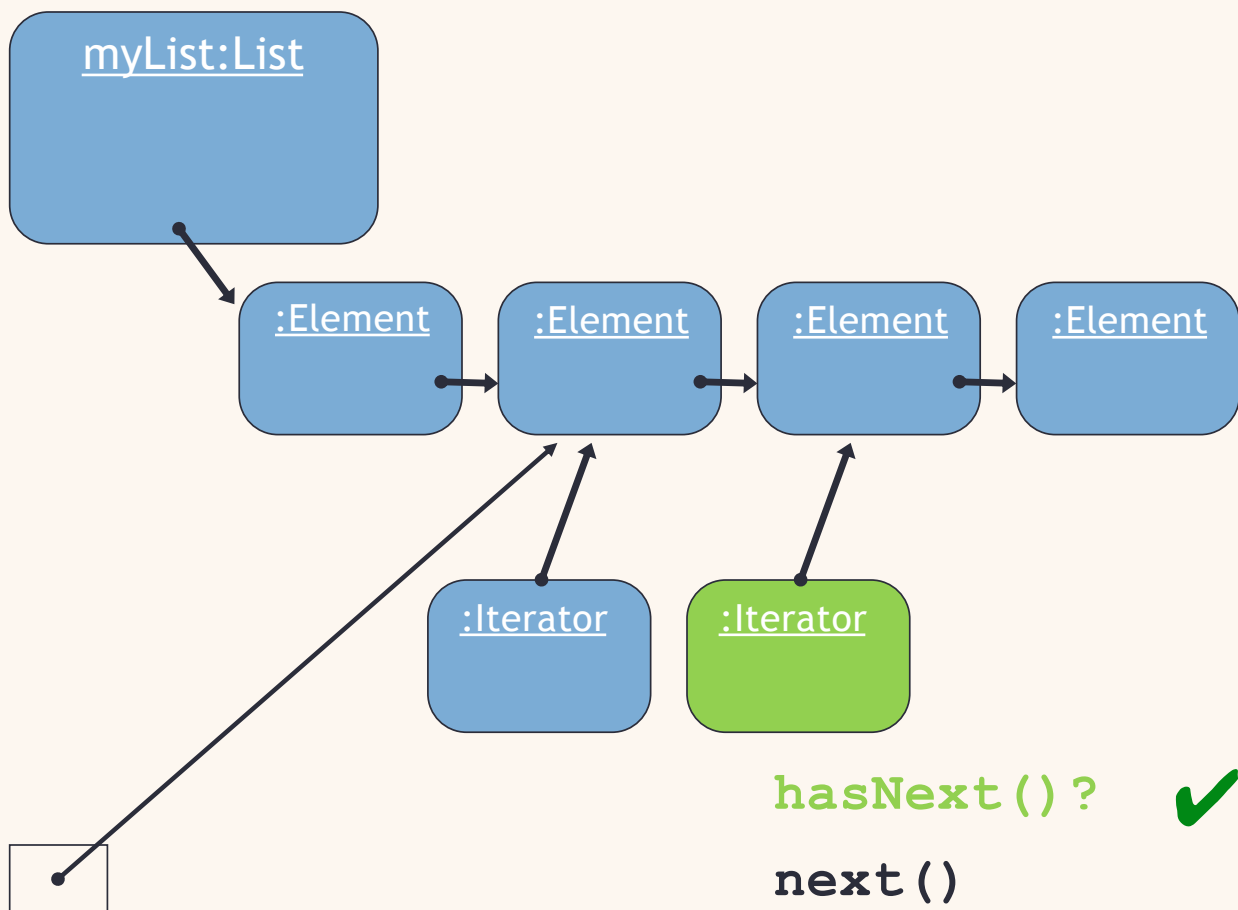
Iterator State Management: Iterators keep track of their position in the collection, determining whether there are more items and which one to return next.

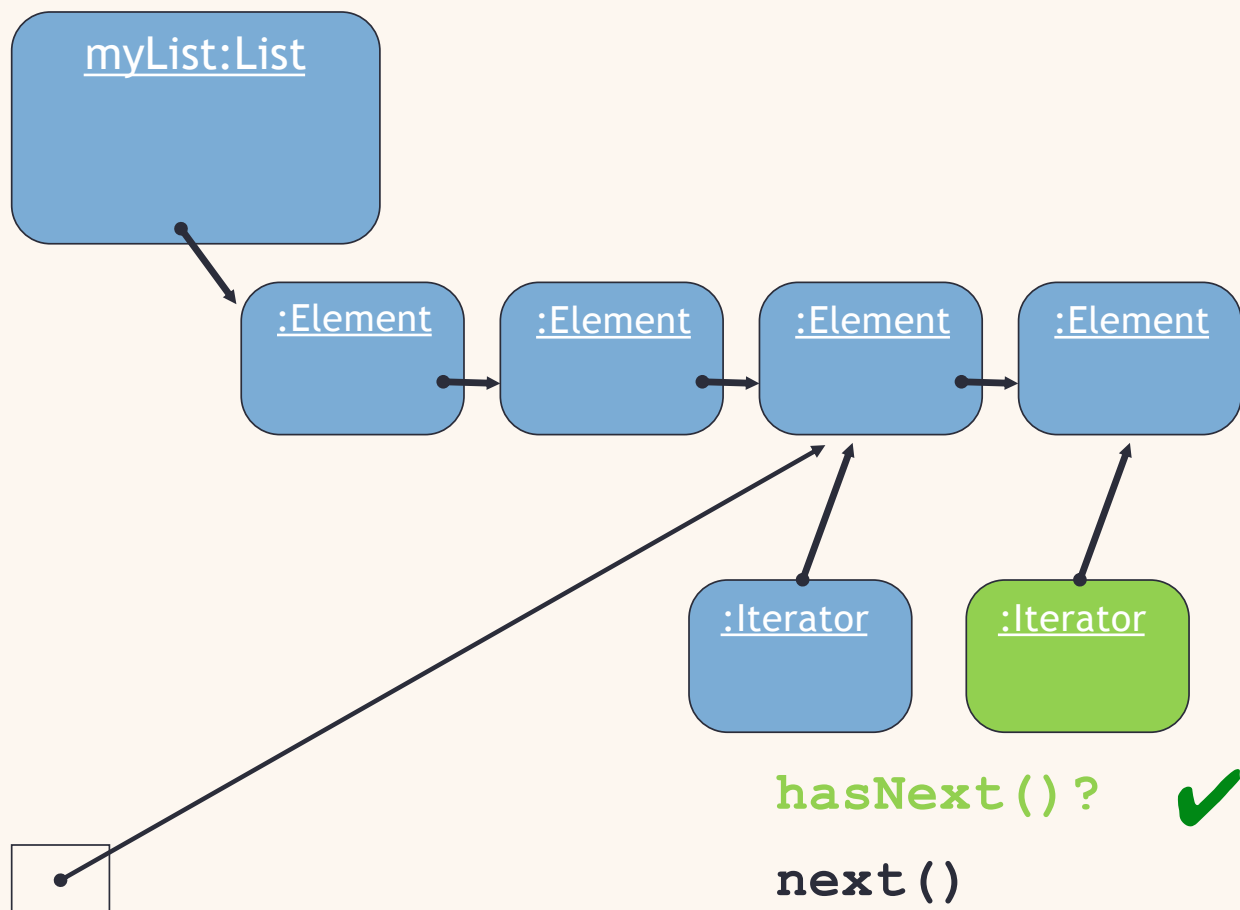
# Iterator mechanics

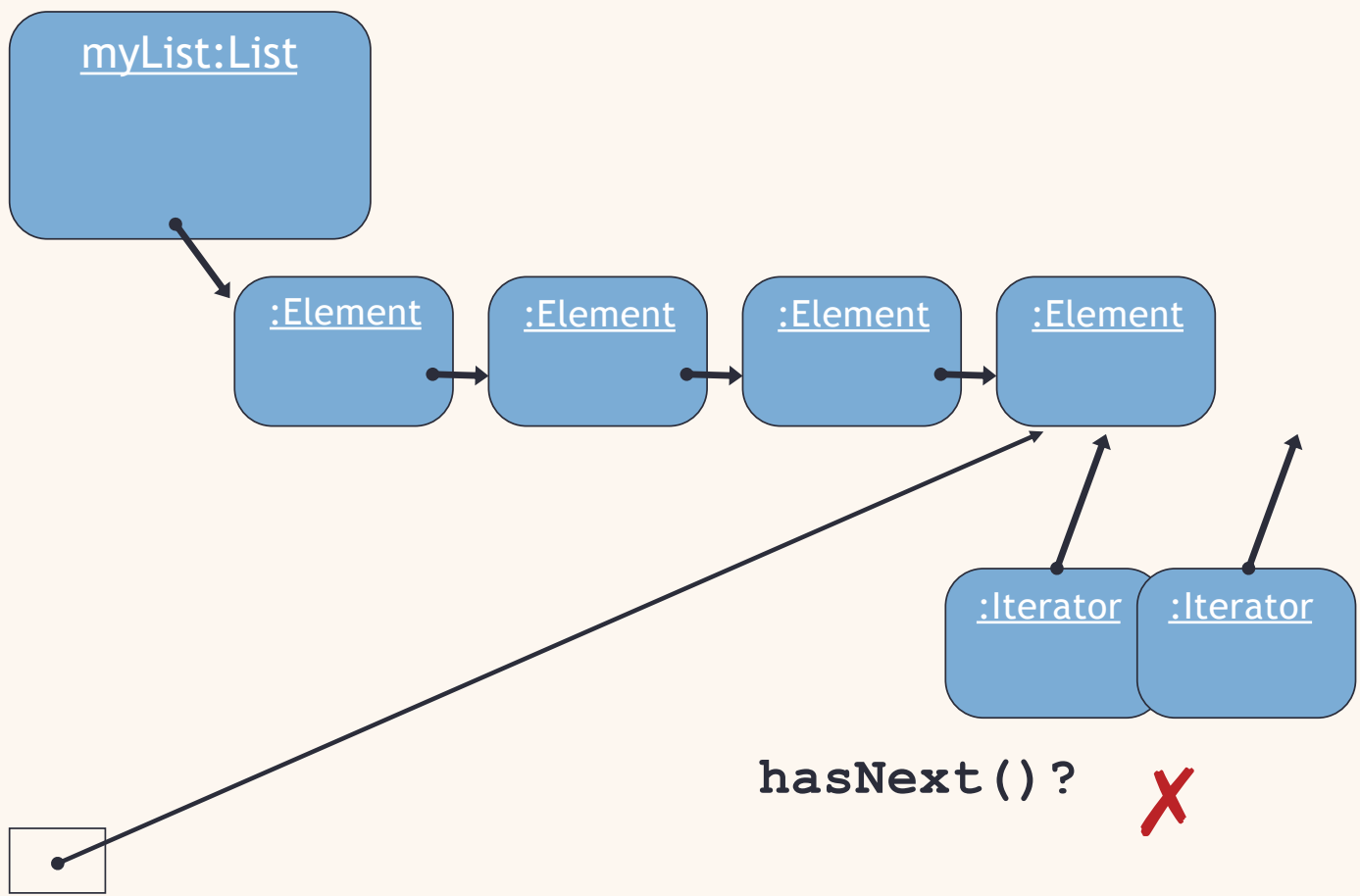




```
Element e = iterator.next();
```





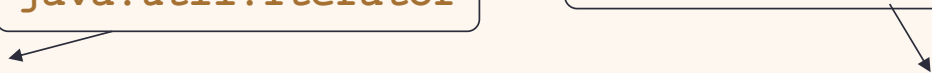


hasNext () ? X

## Using an Iterator object

**java.util.Iterator**

returns an **Iterator** object



```
Iterator<ElementType> it = myCollection.iterator();
while(it.hasNext()) {
    call it.next() to get the next object
    do something with that object
}
```

---

```
public void listAllFiles()
{
    Iterator<String> it = files.iterator();
    while(it.hasNext()) {
        System.out.println(it.next());
    }
}
```

## Index versus Iterator

Ways to iterate over a collection:

- for-each loop.
  - Use if we want to process every element.
- while loop.
  - Use if we might want to stop part way through.
  - Use for repetition that doesn't involve a collection.
- **Iterator** object.
  - Use if we might want to stop part way through.
  - Often used with collections where indexed access is not very efficient, or impossible.
  - *Use to remove from a collection.*

Iteration is an important programming *pattern*.

## Loops

All elements in a collection: use a for-each loop.

```
for(String filename : files) {  
    System.out.println(filename);  
}
```

Something that is not a collection or indefinite amount of repeats: use a while loop:

```
while(loop condition) {  
    loop body  
}
```

Definite amount of repeats: use a for loop


Remove something from a collection: Use a for loop with an iterator.

## Exercise Iterator

a) Create a method named `removeFile` that removes a specified file (`filename`) from a collection of `files` if it exists.

## Removing from a collection

```
public void removeFile(String filename){  
    Iterator it= files.iterator();  
    while(it.hasNext()){  
        if(it.next().equals(filename)){  
            it.remove();  
        }  
    }  
}
```



Using the `Iterator`'s `remove` method.

## Review

Loop statements allow a block of statements to be repeated.

The for-each loop allows iteration over a whole collection.

The while loop allows the repetition to be controlled by a boolean expression.

All collection classes provide special **Iterator** objects that provide sequential access to a whole collection.

## Review loops

- Loop statements allow a block of statements to be repeated.
- The **for-each loop** allows iteration over a whole collection.
- The **while loop** allows the repetition to be controlled by a boolean expression.
- All collection classes provide special **Iterator objects** that provide sequential access to a whole collection.

## Review

Collections are used widely in many different applications.

The Java library provides many different 'ready made' collection classes.

Collections are often manipulated using iterative control structures.

The while loop is the most important control structure to master.

## Review

Some collections lend themselves to index-based access; e.g. **ArrayList**.  
**Iterator** provides a versatile means to iterate over different types of collection.  
Removal using an **Iterator** is less error-prone in some circumstance.

<https://www.youtube.com/watch?v=XkJD80HmpdI>

## Exercises: Usage of ArrayLists

- a) **Get(Retrieve) Item from list:** Write a method call using `get` to return the fifth object stored in a collection(`ArrayList`) called `items`. (4.9)
- b) **Add Item to List:** Write a method call to add the object held in the variable `favoriteTrack` to a collection called `files`. (4.11)
- c) **Check Index:** Write a method called `validIndex` that takes a single integer parameter and checks whether this is a valid index for the collection (`files`) and returns `true` if valid, `false` otherwise. (4.15)

## Exercises: Usage of ArrayLists

### a) Iterating through the whole collection:for-each

Using a local integer variable position, make the `listAllFiles` method below print the index in the collection before the file name.

```
ArrayList<String> files = new ArrayList<>(); ...
public void listAllFiles(){
    for(String filename : files) {
        System.out.println(filename); }
}
```

## Exercises: Usage of while

- a) Write a while loop to add up the values 1 to 10 and print the sum once the loop is finished.
- b) Write a method calls **sum** with a while loop that adds up all numbers between two numbers **a** and **b**. The values **a** and **b** can be passed to the **sum** method as parameters.

## Side note: The **String** class

The **String** class is defined in the **java.lang** package.  
It has some special features that need a little care.  
In particular, comparison of **String** objects can be tricky.

## Side note: String equality

```
if(input == "bye") {  
    ...  
}
```

tests identity

Do not use!!



```
if(input.equals("bye")) {  
    ...  
}
```

tests equality

**Important:** Always use **.equals** for testing String equality!

## String comparison

```
String wordList = "ich bin hier";
```

```
String searchTerm = "hier";
```

```
System.out.println(wordList.equals(searchTerm));
```

```
System.out.println(wordList.contains(searchTerm));
```

```
System.out.println(searchTerm.contains(wordList));
```

## Equality vs Identity

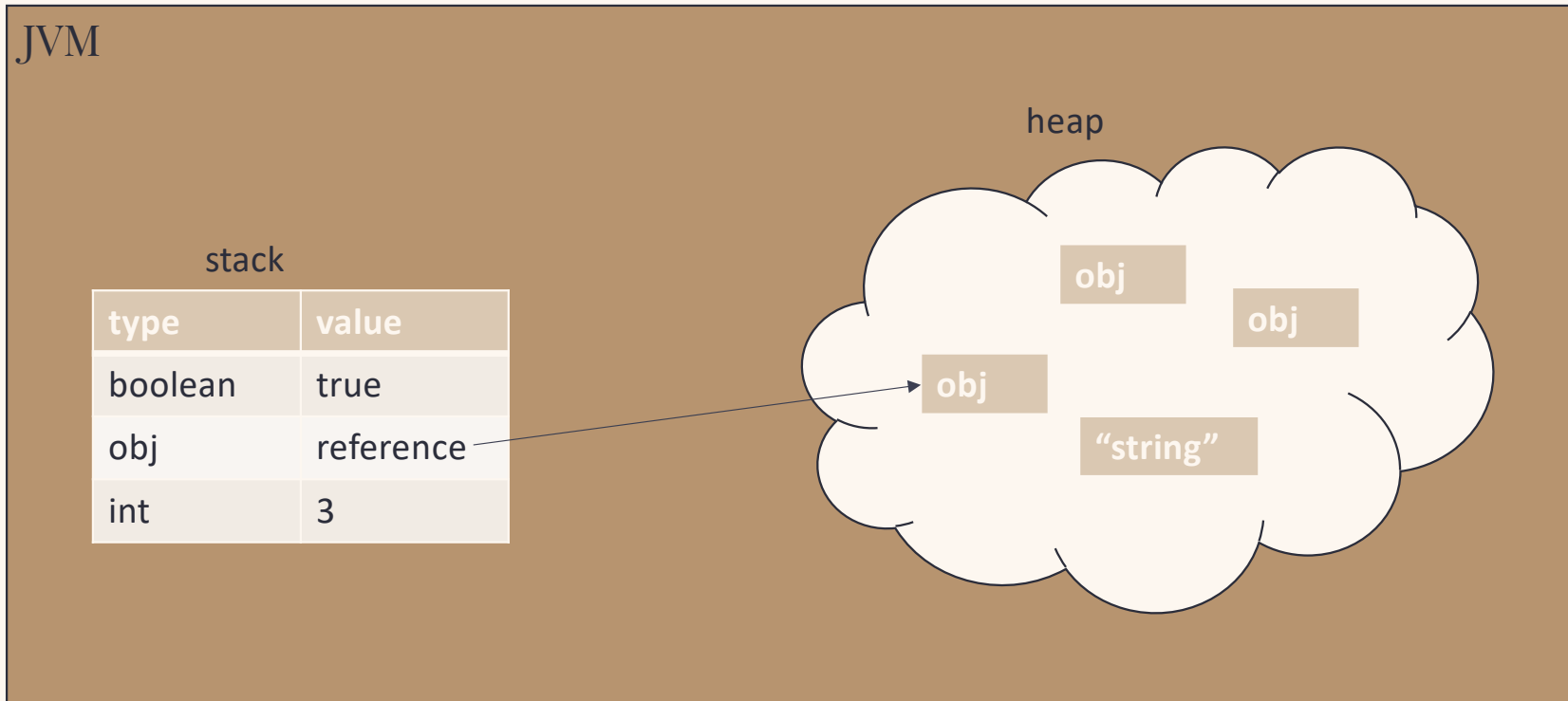
For primitive types (e.g., int, long, float, double, boolean)

- == and != are equality tests

For reference types (i.e., objects)

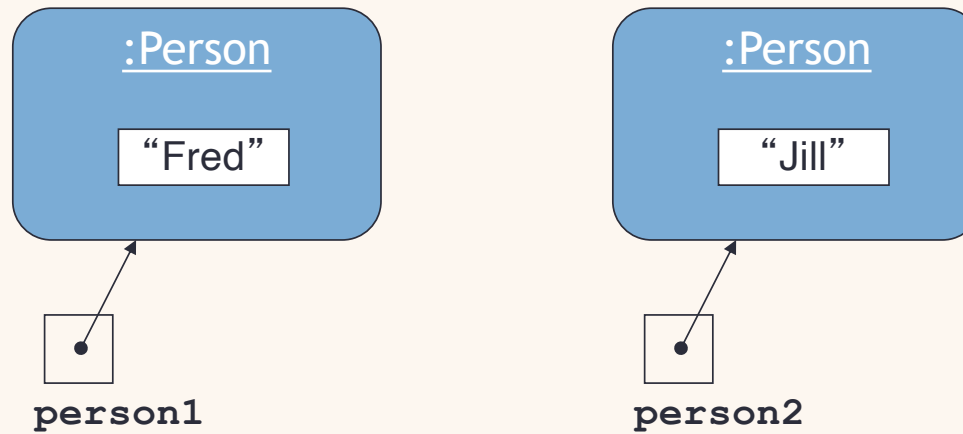
- == and != are identity tests
- In other words, they test if the references indicate the same address in the Heap

# Java Virtual Machine



## Identity vs equality 1

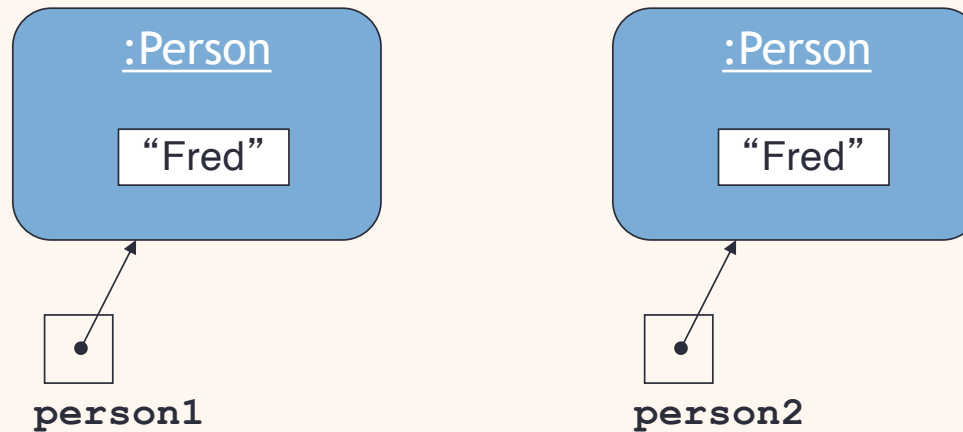
Other (non-String) objects:



`person1 == person2 ?`

## Identity vs equality 2

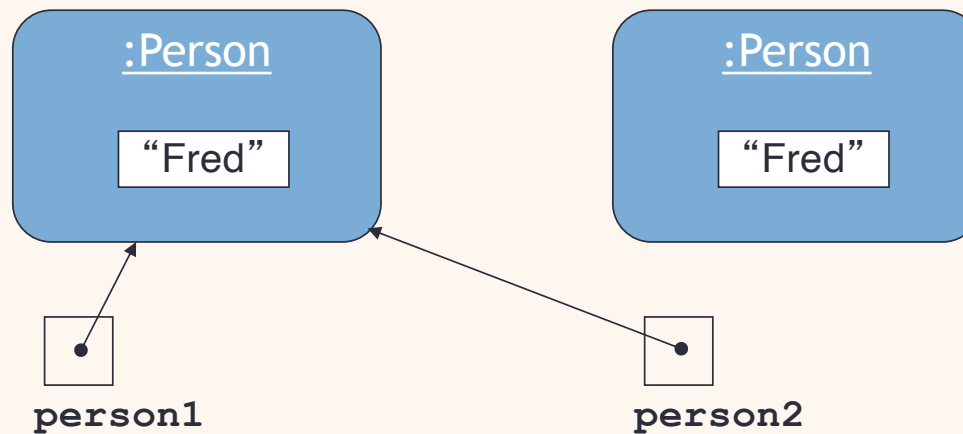
Other (non-String) objects:



`person1 == person2 ?`

## Identity vs equality 3

Other (non-String) objects:

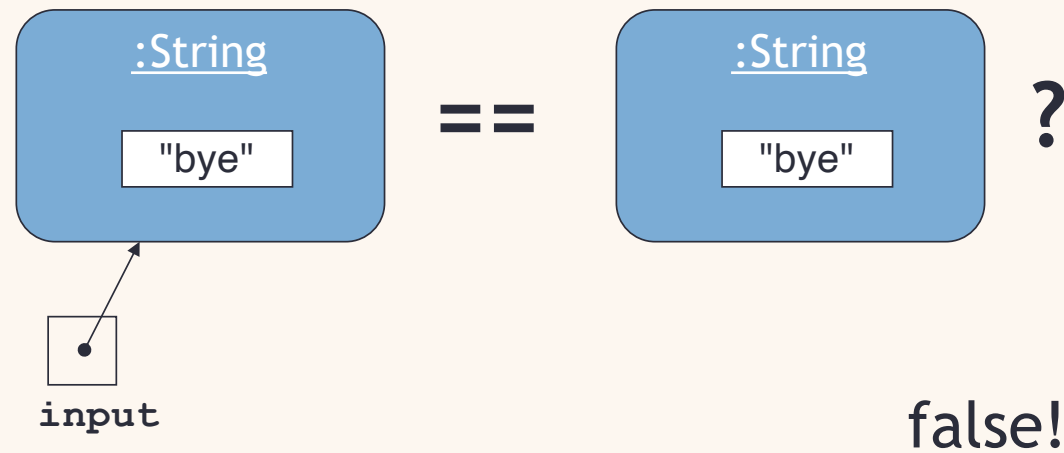


`person1 == person2 ?`

## Identity vs equality (Strings)

```
String input = reader.getInput();  
if(input == "bye") {  
    ...  
}
```

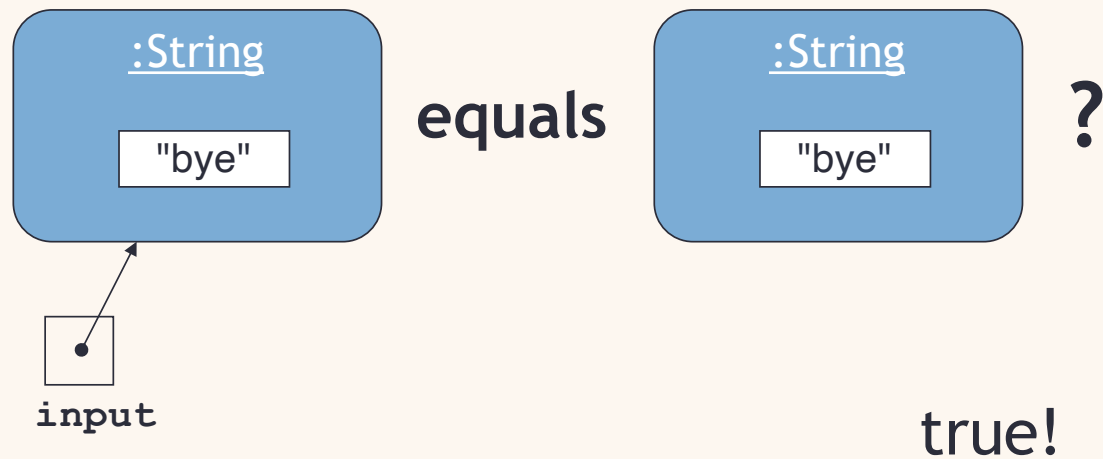
**== tests identity**



## Identity vs equality (Strings)

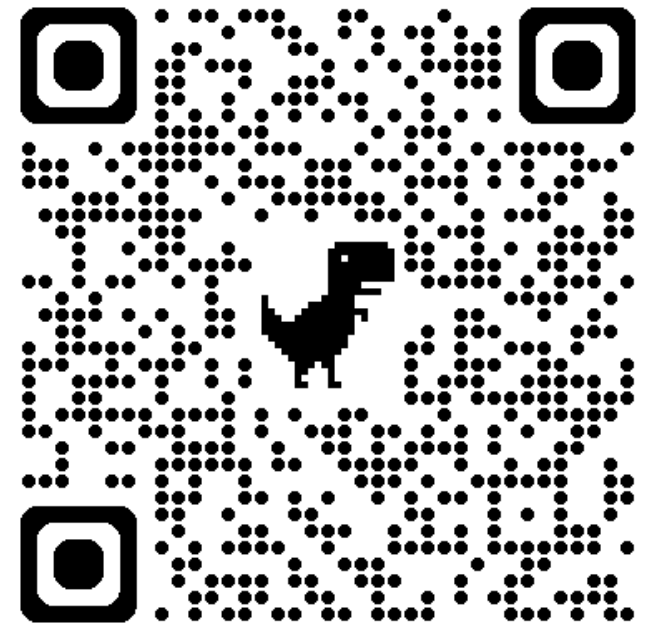
```
String input = reader.getInput();  
if(input.equals("bye")) {  
    ...  
}
```

**equals tests  
equality**



## Eliza: Der erste Chatbot

- Entwickelt 1966 von Joseph Weizenbaum, deutsch-amerikanischer Informatiker.
- Ermöglichte die Kommunikation zwischen Mensch und Computer in natürlicher Sprache.
- Probieren Sie es aus...



## Erfahrungen...?

Slido: Ich hatte, dass Gefühl mit einem Menschen zu sprechen.

# Eliza talking to ChatGPT

---

<https://youtu.be/1yZRNt25Wdo>

<https://t3n.de/news/chatbot-chatgpt-eliza-apple-1966-1554318/>

You> DID YOU COME TO ME BECAUSE YOU ARE PROGRAMMED TO SIMULATE HUMAN LANGUAGE

...

ChatGPT> I apologize if my previous response did not fully address your question. I did not "come" to you, as I am an AI language model and do not have a physical body or presence. Instead, I am a software program designed to respond to and analyze language inputs through a digital platform. My programming includes simulating human language behaviors and patterns to communicate with users like you.

You> ARE YOU SAYING NO JUST TO BE NEGATIVE?

...

ChatGPT> No, I am not saying "no" to be negative. As an AI language model, I do not have personal emotions or desires. My responses are generated based on the analysis of language inputs and programmed algorithms to provide the most appropriate and helpful responses to the user's queries.

You> DID YOU COME TO ME BECAUSE YOU ARE NOT SAYING "NO" TO BE NEGATIVE. AS AN AI LANGUAGE MODEL, ME DO

...

ChatGPT> I'm sorry, I'm having difficulties understanding your question. If I understand correctly, my previous responses were in reference to if I said "no" to be negative, and I stated that I do not have personal emotions or desires and therefore do not have a motive to say "no" to be negative. As for coming to you, I do not have a physical presence and am a software program designed to communicate through a digital platform to assist users with their queries.

You> HOW LONG HAVE YOU BEEN SORRY, YOU'RE HAVING DIFFICULTIES UNDERSTANDING MY QUESTION. IF ME

...

ChatGPT> I apologize for any confusion in my previous response. I have not been "sorry" for a specific duration, as I am an AI language model and do not experience emotions or feelings. Regarding your question, if you could please rephrase or provide additional context, I'll do my best to understand it and provide a helpful response.

## More-Sophisticated Behavior

Using library classes to implement some more advanced functionality

- Using library classes
- Reading documentation

## The Java class library

Thousands of classes.

Tens of thousands of methods.

Many useful classes that make life much easier.

Library classes are often inter-related.

Arranged into packages.

## Working with the library

- A competent Java programmer must be able to work with the libraries.

You should:

- know some important classes by name;
- know how to find out about other classes.

Remember:

- we only need to know the *interface*, not the *implementation*.

## Reading class documentation

Documentation of the Java libraries in HTML format;

Readable in a web browser

Class API: *Application Programmers' Interface*

Interface description for all library classes

## Interface vs implementation

*The documentation includes*

- the name of the class;
- a general description of the class;
- a list of constructors and methods
- return values and parameters for constructors and methods
- a description of the purpose of each constructor and method

 the *interface* of the class

interface is used here to describe the publicly visible part of the class!

...java also has a language construct called interface with a keyword (for later)

## Interface vs implementation

*The documentation **does not** include*

private fields (most fields are private)

private methods

the bodies (source code) of methods



the *implementation* of the class

Welche Interfaces kennen Sie?

Slido

# Interfaces

---

Eine Schnittstelle (engl.: interface) ist ganz allgemein die Verbindungsstelle zwischen zwei miteinander in Beziehung stehenden Einheiten, Systemen...

In computing, an interface is a shared boundary across which two or more separate components of a computer system exchange information. The exchange can be between software, computer hardware, peripheral devices, humans, and combinations of these.[1]

[https://en.wikipedia.org/wiki/Interface\\_\(computing\)#cite\\_note-HookwayInterface14-1](https://en.wikipedia.org/wiki/Interface_(computing)#cite_note-HookwayInterface14-1)

Hookway, B. (2014). "Chapter 1: The Subject of the Interface". Interface. MIT Press. pp. 1–58. ISBN 9780262525503

# Interfaces

---

## Hardware to hardware

- USB, Ethernet-Anschluss, HDMI, audio input/ output

## Hard-to Software

- Sensors and actuators in smartphones, gaming controllers, wearable devices, Smart home devise (Amazon Echo)

## Software to software

- Application Programming Interfaces (API's)
- Google Maps API, Twitter API, Plugins, Software Development Kits

## Human- computer interfaces (UI):

- headphones, displays, controllers, head-mounted Displays

[https://en.wikipedia.org/wiki/Interface\\_\(computing\)#cite\\_note-HookwayInterface14-1](https://en.wikipedia.org/wiki/Interface_(computing)#cite_note-HookwayInterface14-1)

Hookway, B. (2014). "Chapter 1: The Subject of the Interface". Interface. MIT Press. pp. 1–58. ISBN 9780262525503

# Interfaces in Java

- Method signature / header
  - access modifier + return type + name + parameters  
`public int myMethod (myType t, String s)`
- Class

## Documentation for **startsWith**

---

### **startsWith**

- **public boolean startsWith([String](#) prefix)**

Tests if this string starts with the specified prefix.

Parameters:

- **prefix** - the prefix.

Returns:

- **true** if the ...; **false** otherwise

## Methods from **String**

---

- **contains**
- **endsWith**
- **indexOf**
- **substring**
- **toUpperCase**
- **trim**
- Beware: strings are *immutable*!

## Immutable String

Strings are constant; their values cannot be changed after they are created  
So its methods have a return type!

```
input = input.trim();
```

## Practice

How can we check both “bye” and “Bye”?

Explore the String documentation...

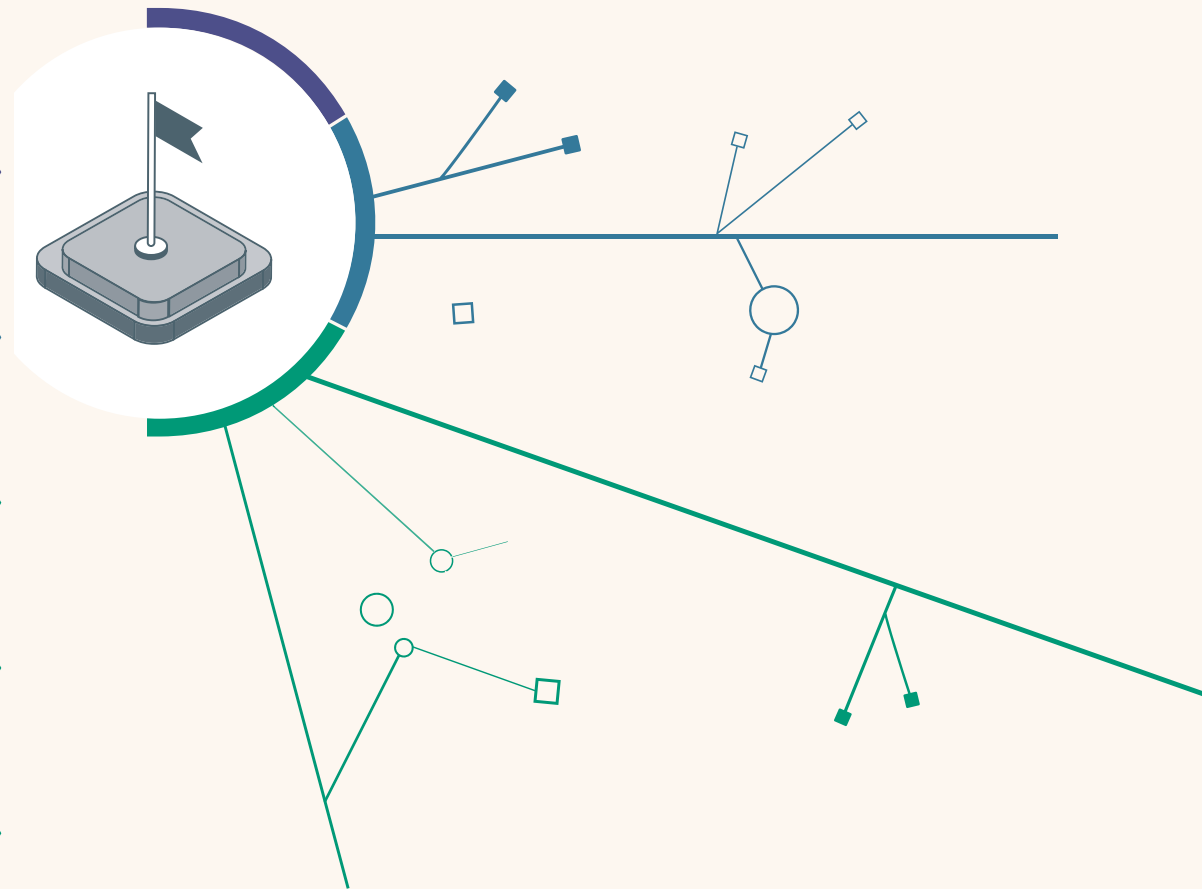
# Agenda

TechSupport Project

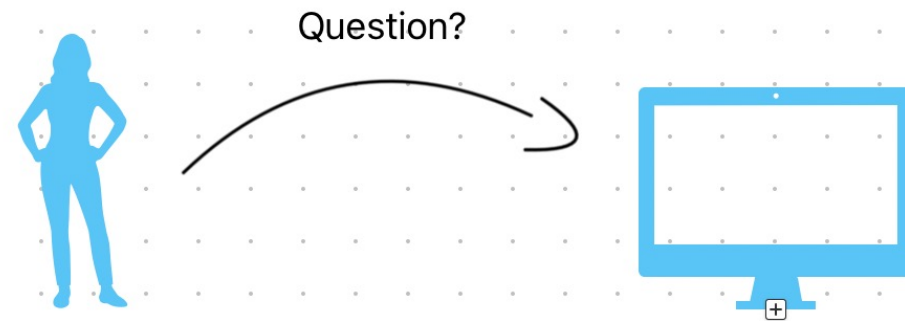
Random

HashMap

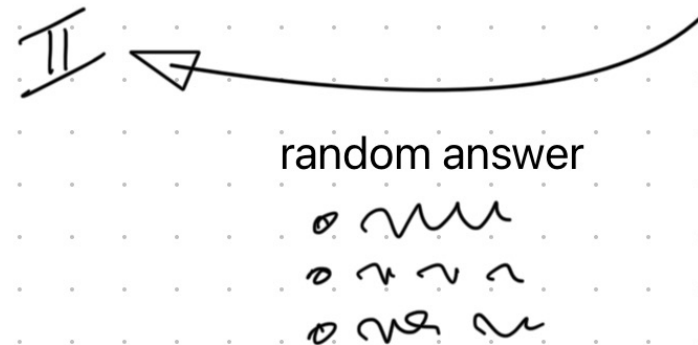
HashSet



## Iteratives Vorgehen I ,II

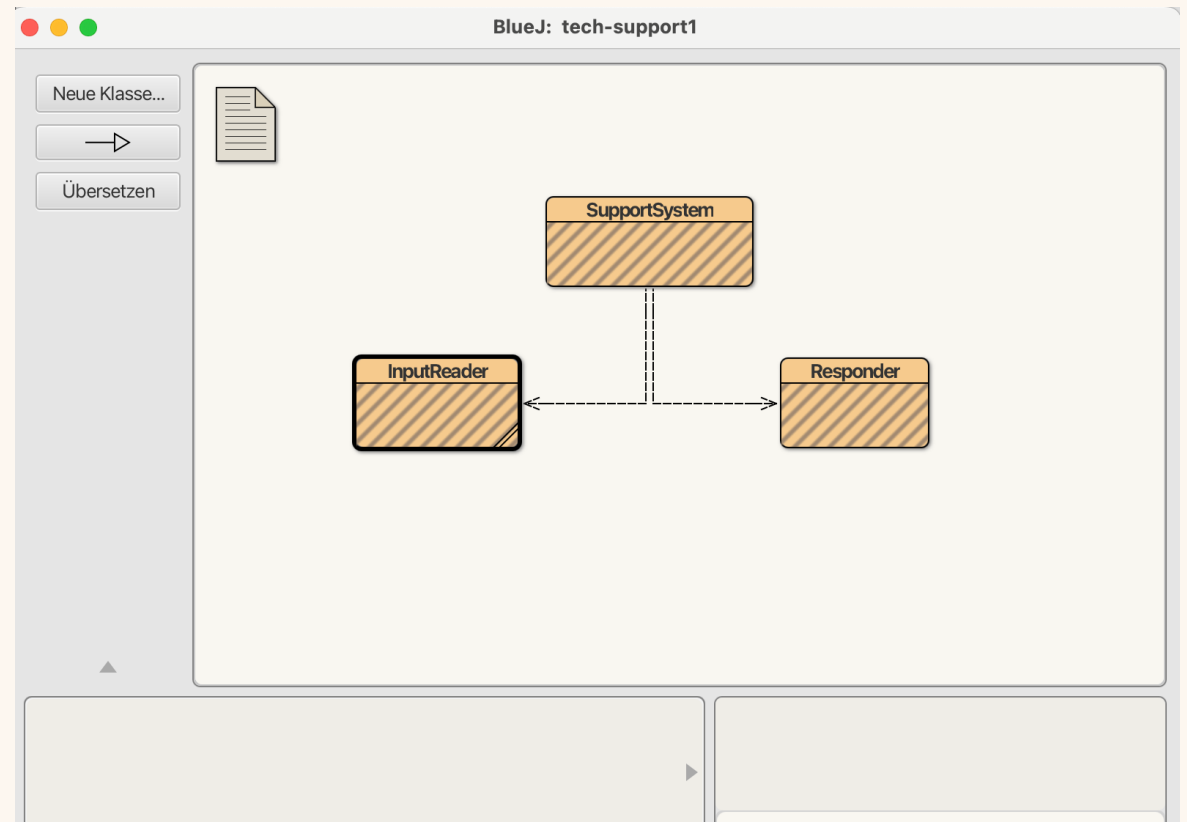


- String



- ArrayList
- Random

# Tech-support1



## The exit condition

```
String input = reader.getInput();
```

```
if(input.startsWith("bye")) {  
    finished = true;  
}
```

Where does '**startsWith**' come from?

What is it? What does it do?

How can we find out?

## Main loop body in **SupportSystem**


```
String input = reader.getInput();
```

```
...
```

```
String response = responder.generateResponse();
```

```
System.out.println(response);
```

NB: input is ignored by the Responder in this version



# Programmierung / Algorithmus / Eigenschaften

## notwendig:

- **Eindeutigkeit**  
keine Zweifel, was wann zu tun ist
- **Parametrisierbarkeit**  
für mehrere „ähnliche“ Aufgaben einsetzbar
- **Finitheit**  
Endliche Beschreibung
- **Ausführbarkeit**  
lauffähig auf Computer
- **Terminierung**  
endet irgendwann
- **Determiniertheit**  
arbeitet bei jedem Durchlauf gleich

## wünschenswert:

- korrekt
- vollständig
- effizient (Zeit)
- sparsam (Speicher)
- erweiterbar
- wiederverwendbar
- portabel
- verständlich

## randomness vs determinism

Computer programs can not act random (->Determinacy)

So the act **pseudorandom**

## Adding random behavior

The library class **Random** can be used to generate random numbers:

```
import java.util.Random;  
...  
Random rand = new Random();  
...  
int num = rand.nextInt();  
int value = 1 + rand.nextInt(100);  
int index = rand.nextInt(list.size());
```

## Using library classes

Classes organized into packages.

Classes from the library must be *imported* using an **import** statement;

- except from the **java.lang** package.

They can then be used like classes from the current project.

## Packages and import

Single classes may be imported:

```
import java.util.ArrayList;
```

Whole packages can be imported:

```
import java.util.*;
```

Importation does not involve source code insertion.

## Random within a predefined range

First and last possible number created here?

```
int value = 1 + rand.nextInt(100);
```

## Practice

- a) Schreiben Sie eine Methode `throwDice`, die Werte von 1 bis einschließlich 6 zurückliefert.
- b) Schreiben Sie eine Methode `generateResponse`, die zufällig eine der Zeichenketten "ja", "nein", "vielleicht" zurückliefert.
- c) Erweitern Sie Ihre Methode `generateResponse` so, dass sie eine `ArrayList` mit einer beliebigen Anzahl an Antworten benutzt, aus denen sie zufällig eine auswählt.

## Selecting random responses

```
public Responder()
{
    randomGenerator = new Random();
    responses = new ArrayList<>();
    fillResponses();
}

public void fillResponses()
{
    fill responses with a selection of response strings
}

public String generateResponse()
{
    int index = randomGenerator.nextInt(responses.size());
    return responses.get(index);
}
```

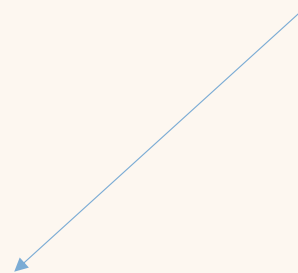
## Selecting random responses

```
public Responder()
{
    randomGenerator = new Random();
    responses = new ArrayList<>();
    fillResponses();
}

public void fillResponses()
{
    fill responses with a selection of response strings
}

public String generateResponse()
{
    int index = randomGenerator.nextInt(responses.size());
    return responses.get(index);
}
```

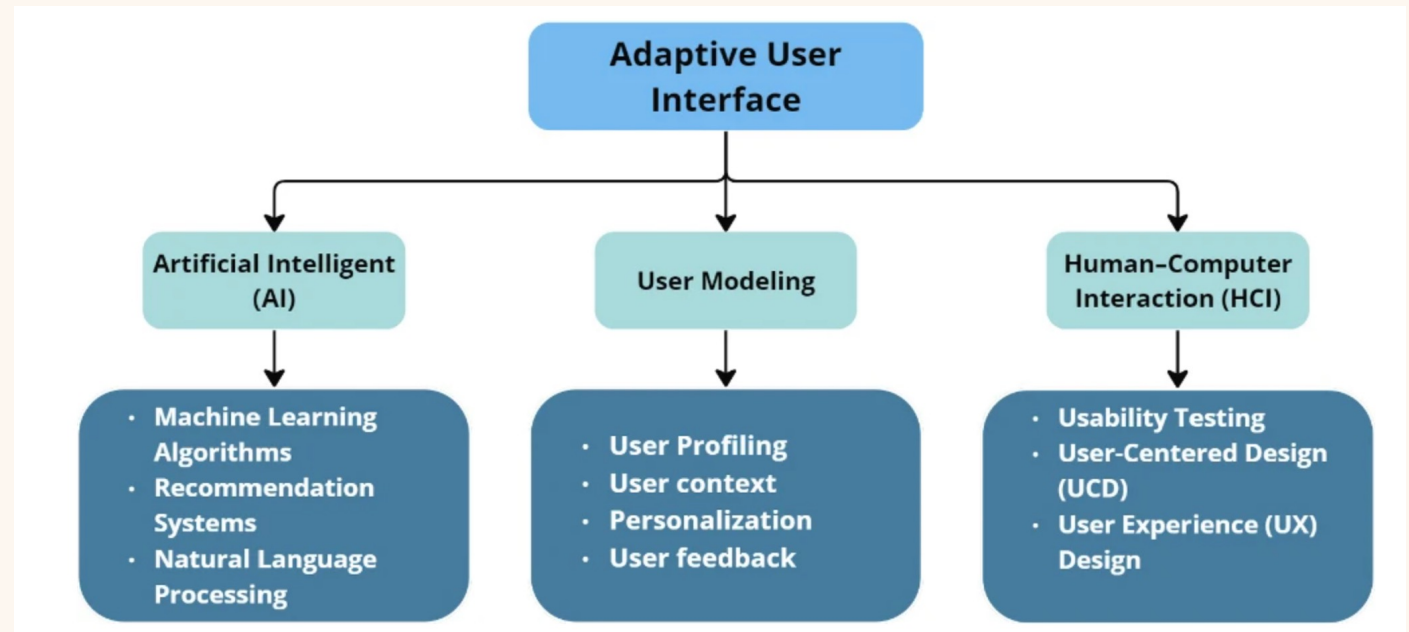
**ArrayIndexOutOfBoundsException**



## Practice

Was passiert, wenn Sie der Liste mehr (oder weniger) mögliche Antworten hinzufügen? Wird die Auswahl einer zufälligen Antwort (`generateResponse()`) dann noch richtig funktionieren?

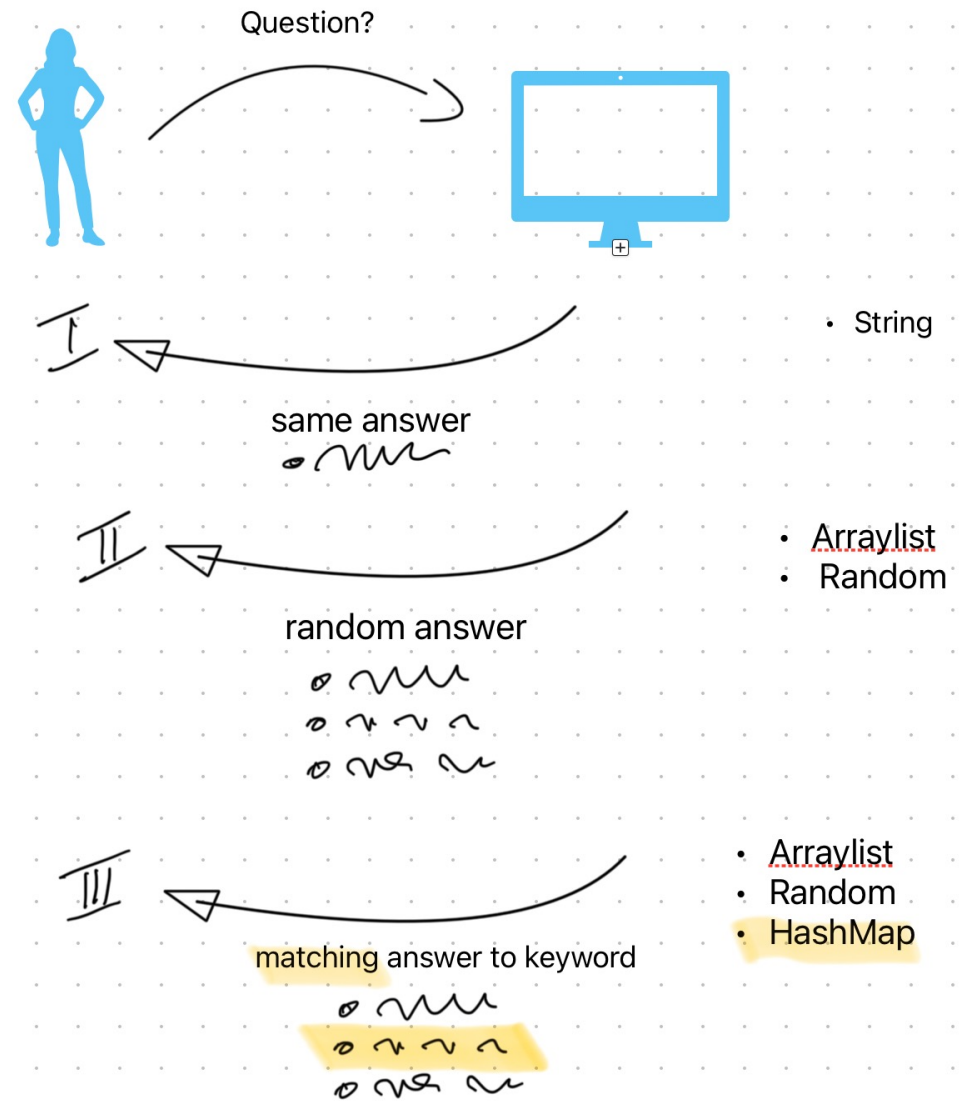
# Adaptive systems



Carrera-Rivera, Angela, Felix Larrinaga, Ganix Lasa, Giovanna Martinez-Arellano, and Gorka Unamuno. "AdaptUI: A Framework for the Development of Adaptive User Interfaces in Smart Product-Service Systems." *User Modeling and User-Adapted Interaction*, August 12, 2024.  
<https://doi.org/10.1007/s11257-024-09414-0>.

## Iteratives Vorgehen III

[https://www.youtube.com/watch?v=NY33-ErTvju&list=PL9HfA4ZKbzin-tNeJi09vJxII\\_RiOn9eB&index=5](https://www.youtube.com/watch?v=NY33-ErTvju&list=PL9HfA4ZKbzin-tNeJi09vJxII_RiOn9eB&index=5)



## Maps (assoziative Suchtabellen)

- Maps are collections that contain *pairs* of objects.
- Like ArrayList stores a flexible number of entries but each entry is not an object but a pair of objects.
- Parameterized with *two* types.
- Pairs consist of a *key* and a *value* (*Schlüssel-Wert-Paare* ).
- Lookup works by supplying a key, and retrieving a value.
- Example: a contacts list.

## Using maps

A map with strings as keys and values

One way  
lookup!

:HashMap

"Charles Nguyen"	"(531) 9392 4587"
------------------	-------------------

"Lisa Jones"	"(402) 4536 4674"
--------------	-------------------

"William H. Smith"	"(998) 5488 0123"
--------------------	-------------------

## Using maps

```
HashMap <String, String> contacts = new HashMap<>();
```

```
contacts.put("Charles Nguyen", "(531) 9392 4587");
```

```
contacts.put("Lisa Jones", "(402) 4536 4674");
```

```
contacts.put("William H. Smith", "(998) 5488 0123");
```

```
String number = contacts.get("Lisa Jones");
```

```
System.out.println(number);
```



key

## Practice

- a) What happens if you add an entry to a map with a key that already exists in the map?
- b) What happens when you add an entry to a map with two different keys?
- c) How do you check whether a given key is contained in a map? (Give a java code example)
- d) What happens when you try to look up a value and the key does not exist in the map?
- e) How do you check how many entries are contained in a map?
- f) How do you print out all keys currently stored in a map?

## Practice

---

- a) What happens if you add an entry to a map with a key that already exists in the map?
  - It overwrites the previous value associated with the key.
- b) What happens when you add an entry to a map with two different keys?
  - Both values stay in the map. HashMaps only uses the key to distinguish entries - not the values.
- c) How do you check whether a given key is contained in a map? (Give a java code example)
  - `myMap.containsKey("key");`
- d) What happens when you try to look up a value and the key does not exist in the map?
  - Returns null
- e) How do you check how many entries are contained in a map?
  - `myMap.size()`
- f) How do you print out all keys currently stored in a map?
  - ```
for (String name : myMap.keySet()) {  
    System.out.println(name);  
}
```

## HashMap in Responder

```
private HashMap <String, String> responseMap;

...

responseMap.put("crash", "Which version is this?");

...

String response = responseMap.get(word);
if(response != null) {
    ...
}
```

# Iteratives Vorgehen

