Agenda

- Recap: Data types , object interaction
- Grouping obejcts: Collections
- ArrayList: Java API
- ArrayList: Declare, add, retrieve (get), size()
- …

Primitive (value) types in Java

https://freiheit.f4.htw-berlin.de/prog1/variablen/

Datentyp	Größe	Wertebereich
`boolean`	1 Byte[2]	`true` / `false`
`char`	16 bit	`0` … `65.535` (z.B. `'A'`)
`byte`	8 bit	`-128` … `127`
`short`	16 bit	`-32.768` … `32.767`
`int`	32 bit	`-2.147.483.648` … `2.147.483.647`
`long`	64 bit	`-2^63` … `2^63-1`
`float`	32 bit	`+/-1,4E-45` … `+/-3,4E+38`
`double`	64 bit	`+/-4,9E-324` … `+/-1,7E+308`

Was ist ein Datentyp?

Ein Datentyp definiert eine Menge von Werten, die zu diesem Typ gehören, sowie eine Menge von Operationen, die mit den Werten des Typs ausgeführt werden dürfen. Der Datentyp bestimmt wie viel Speicher reserviert wird (und legt die Interpretation der einzelnen Bits fest).

Primitive types vs. object types

primitive type

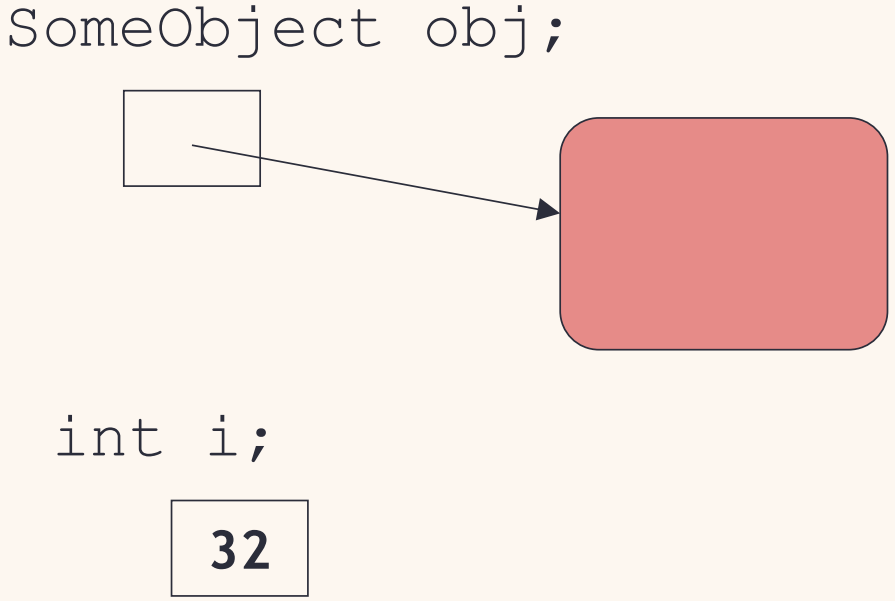
When you create a primitive variable, the actual value is stored directly in the memory location reserved for that variable.

```
int i;
```

Primitive types vs. object types

object type

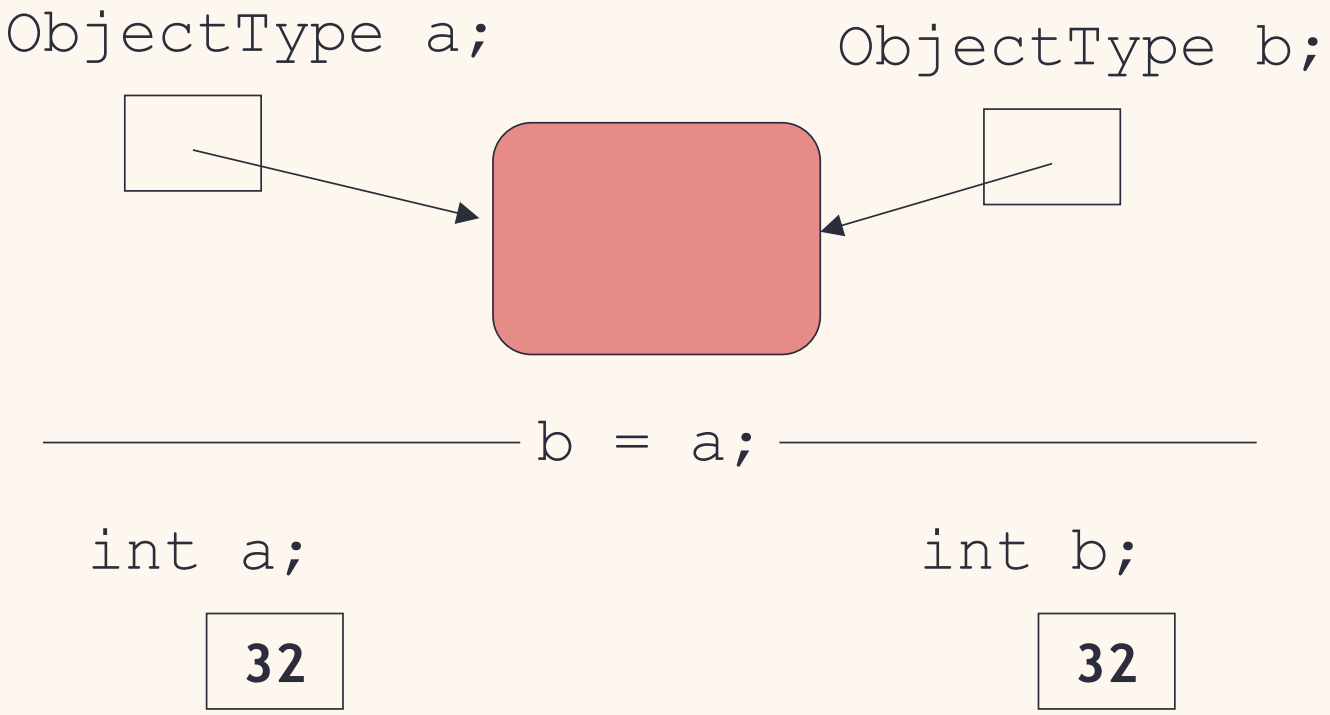
When you create an object, the variable does not store the actual object, but a reference (like an address or pointer) to where the object is stored in memory. The object itself is stored in a different area of memory called the heap.



Primitive types vs. object types (Wert- vs Referenz-Typ)

Assigning an object variable copies the reference, so two variables can refer to the same object.

Assigning a primitive variable copies the value.



Quiz: What is the output?

```
int a;
int b;
a = 32;
b = a;
a = a + 1;
System.out.println(b);
```

```
Person a;
Person b;
a = new Person("Everett");
b = a;
  a.changeName("Delmar");
  System.out.println(b.getName()
  );
```

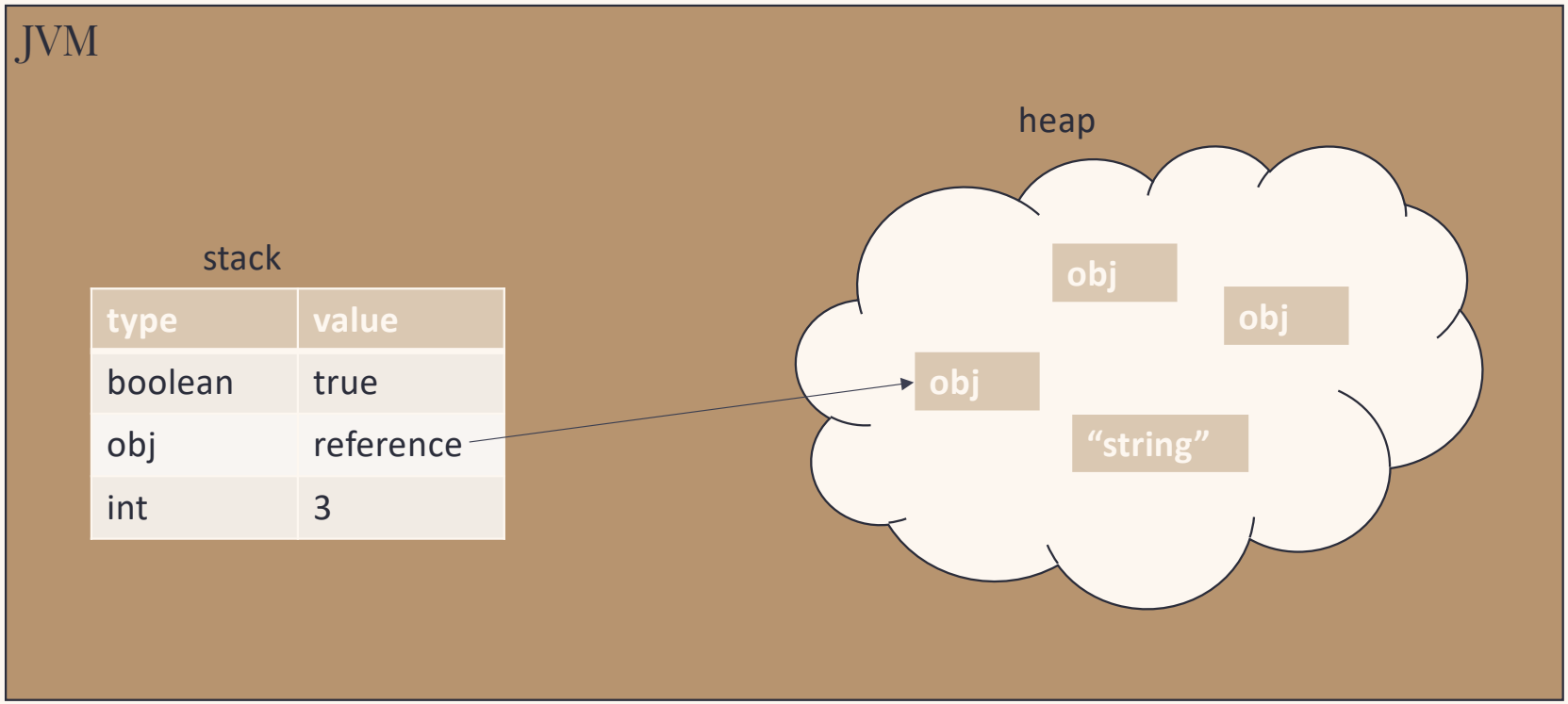
Java Reference, Stack and Heap

https://docs.oracle.com/javase/8/docs/api/

https://www.w3schools.com/java/java_ref_reference.asp

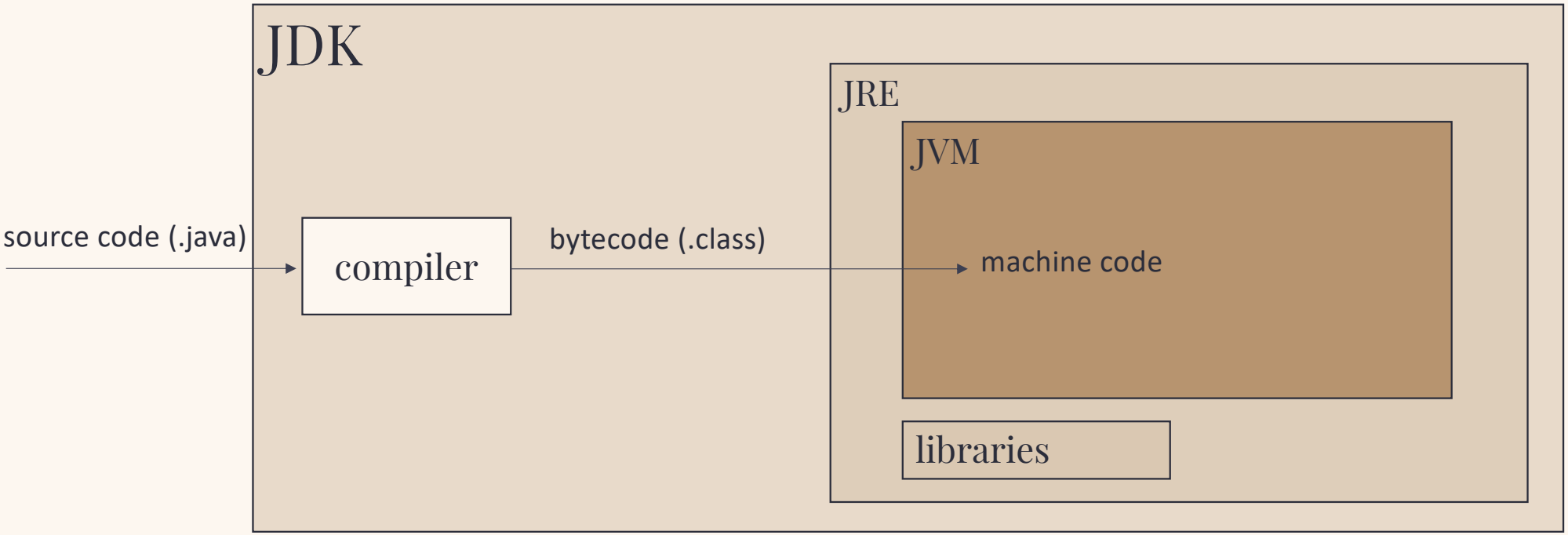
https://www.youtube.com/watch?app=desktop&v=_GK3WoFFKUE

Java Virtual Machine



Java: Development Kit / Runtime Environment / Virtual Machine

- Portability : Write once, run anywhere (WORA)



Interlude:
Some popular errors...

What's wrong here?

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(files.size() == 0); {
      System.out.println("Organizer is empty");
   }
   else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + " files");
   }
}
```

This is the same as before!

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(files.size() == 0);

   {
      System.out.println("Organizer is empty");
   }
   else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + "files");
   }
}
```

This is the same again

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(files.size() == 0)
      ;

   {
      System.out.println("Organizer is empty");
   }
   else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + "files");
   }
}
```

This time I have a boolean field called 'isEmpty' ...

What's wrong here?

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
    if(isEmpty = true) {
        System.out.println("Organizer is empty");
    }
    else {
        System.out.print("Organizer holds ");
        System.out.println(files.size() + "files");
    }
}
```

This time I have a boolean field called ‘isEmpty’...

The correct version

```
/**
 * Print out info (number of entries).
 */
public void showStatus()
{
   if(isEmpty == true) {
      System.out.println("Organizer is empty");
   }
   else {
      System.out.print("Organizer holds ");
      System.out.println(files.size() + "files");
   }
}
```

What's wrong here?

```
/**
 * Store a new file in the organizer. If the
 * organizer is full, save it and start a new one.
 */
public void addFile(String filename)
{
   if(files.size() == 100)
       files.save();
       // starting new list
       files = new ArrayList<String>();

   files.add(filename);
}
```

The correct version

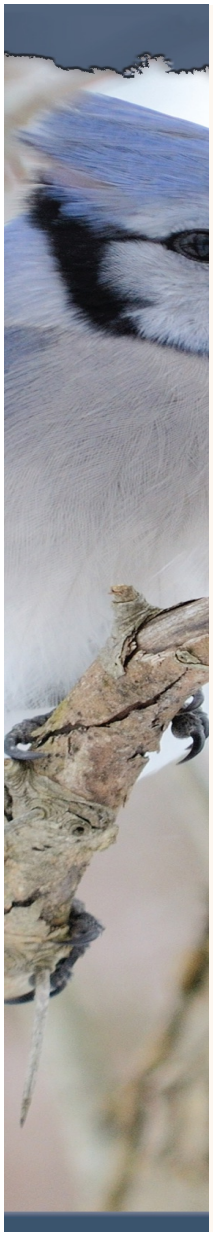
```
/**
 * Store a new file in the organizer. If the
 * organizer is full, save it and start a new one.
 */
public void addFile(String filename)
{
    if(files.size() == 100) {
        files.save();
        // starting new list
        files = new ArrayList<String>();
    }

    files.add(filename);
}
```

Abbreviations for Assignments: Increment/Decrement

Operator	Name	Meaning
++a	Pre-increment	Increases a by 1 and returns the result
a++	Post-increment	Returns a, then increases a by 1
--a	Pre-decrement	Decreases a by 1 and returns the result
a--	Post-decrement	Returns a, then decreases a by 1

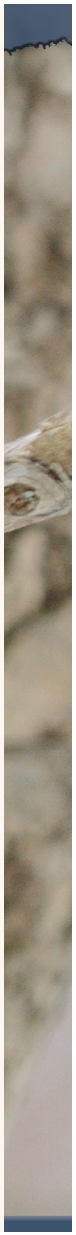
Agenda

- Recap: Data types , object interaction
- Grouping obejcts: Collections
- ArrayList: Java API
- ArrayList: Declare, add, retrieve (get), size()
- … while



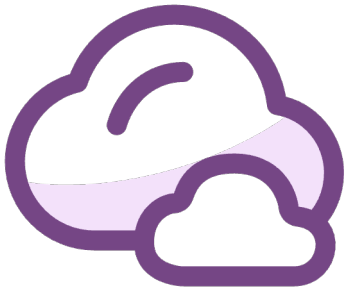
Grouping objects

Introduction to collections



Main concepts to be covered

- Collections (especially **`ArrayList`**)
- Builds on the *abstraction* theme from the last chapter.



Welche Besipiele für Sammlungen kennen Sie?

The requirement to group objects

Many applications involve collections of objects:

- Personal organizers.
- Library catalogs
- Shopping cart

The number of items to be stored varies.

- Items added.
- Items deleted.

An organizer for music files (Play list)

Single-track files may be added.

There is no pre-defined limit to the number of files/tracks.

It will tell how many file names are stored in the collection.

It will list individual file names.

Explore the *music-organizer-v1* project.

Class libraries

Collections of useful classes.

We don't have to write everything from scratch.

Java calls its libraries, *packages*.

Grouping objects is a recurring requirement.

- The **`java.util`** package contains multiple classes for doing this.

```
import java.util.ArrayList;

/**
 * ...
 */
public class MusicOrganizer
{
    // Storage for an arbitrary number of file names.
    private ArrayList<String> files;

    /**
     * Perform any initialization required for the
     * organizer.
     */
    public MusicOrganizer()
    {
        files = new ArrayList<>()
    }

    ...
}
```

Java Libraries

Java™ Platform Standard Ed. 7

All Classes

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom
java.awt.im
java.awt.im.spi

AnySeqHelper
AnySeqHelper
AnySeqHolder
AppConfigurationEntry
AppConfigurationEntry.LoginModuleControlFlag
Appendable
Applet
AppletContext
AppletInitializer
AppletStub
ApplicationException
Arc2D
Arc2D.Double
Arc2D.Float

Overview Package Class Use Tree Deprecated Index Help — *Java™ Platform Standard Ed. 7*

Prev Class **Next Class** Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

```
private ArrayList<String> files;
```

Type of Collection

Type of Objects

Type

Collections

We specify:

- the type of collection: **`ArrayList`**
- the type of objects it will contain: **`<String>`**

We say, "*ArrayList of String*".

```
private ArrayList<String> files;
Type of Collection
Type of Objects
Type
```

Generic classes

Collections are known as *parameterized* or *generic* types.

`ArrayList` implements list functionality:

- **`add`**, **`get`**, **`size`**, etc.

The type parameter says what we want a list of:

- **`ArrayList<Person>`**
- **`ArrayList<TicketMachine>`**
- etc.

Creating an ArrayList object

In versions of Java prior to version 7:

- `files = new ArrayList<String>();`

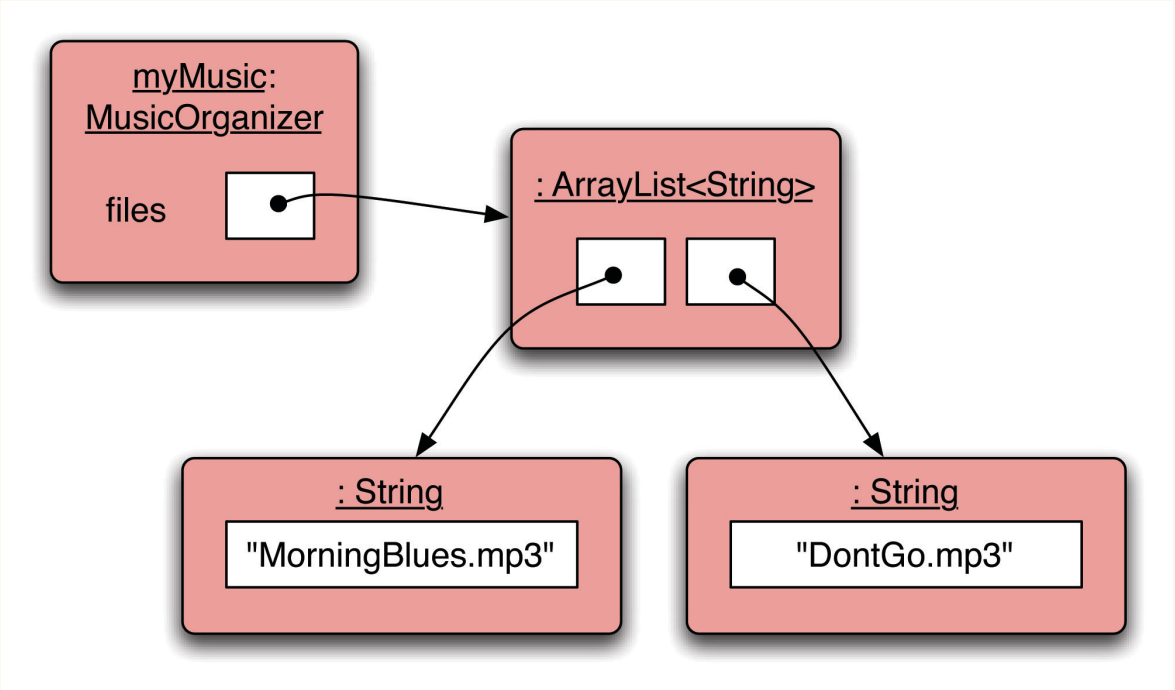
Java 7 introduced 'diamond notation'

- `files = new ArrayList<>();`

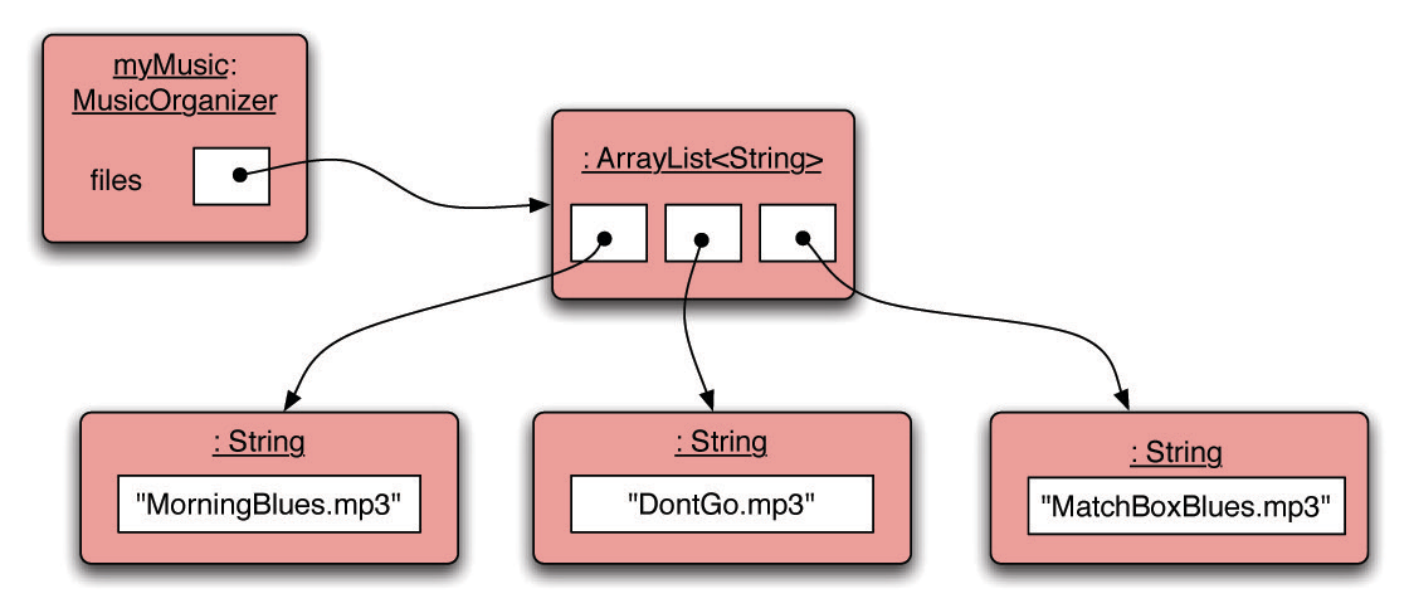
The type parameter can be inferred from the variable being assigned to.

- A convenience we will use.

Object structures with collections



Adding a third file



Features of the collection

It increases its capacity as necessary.

It keeps a private count:

- **`size()`** accessor.

It keeps the objects in order.

Details of how all this is done are hidden.

- Does that matter? Does not knowing how prevent us from using it?

Generic classes

We can use **`ArrayList`** with **any** class type:

- **`ArrayList<TicketMachine>`**
- **`ArrayList<ClockDisplay>`**
- **`ArrayList<Track>`**
- **`ArrayList<Person>`**

Each will store multiple objects of the specific type.

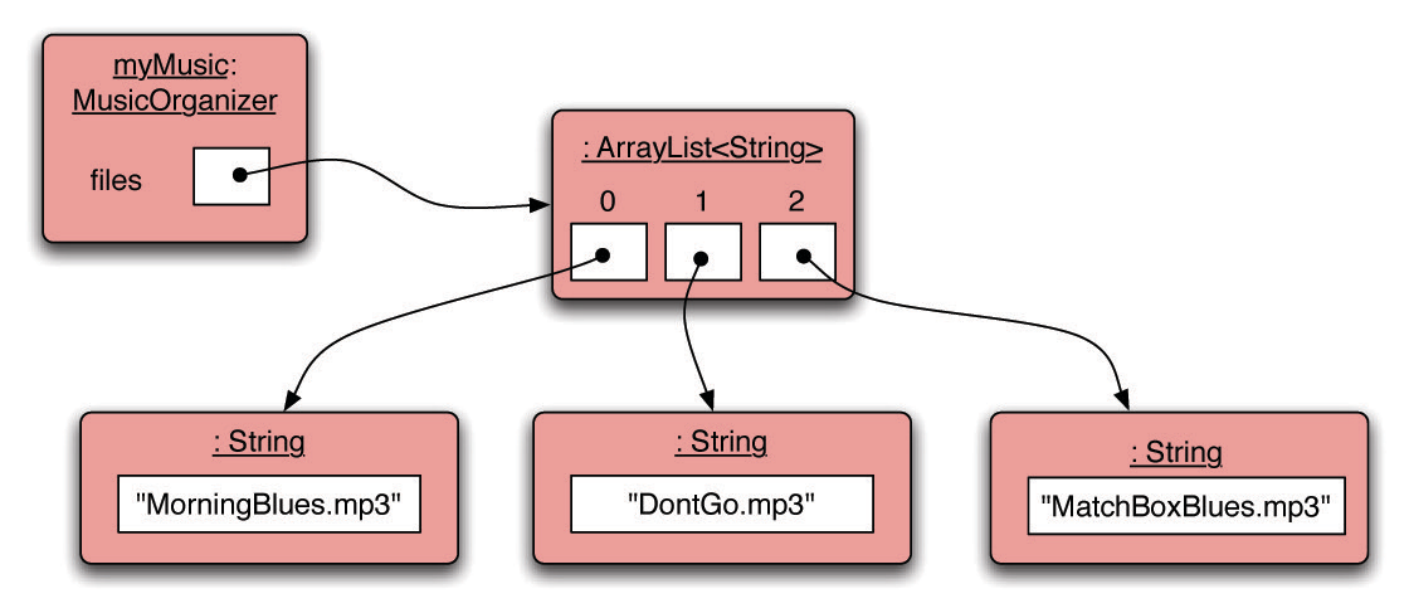
Using the collection

```
public class MusicOrganizer
{
    private ArrayList<String> files;

    ...

    public void addFile(String filename)
    {
        files.add(filename);
    }

    public int getNumberOfFiles()
    {
        return files.size();
    }

    ...
}
```

Adding a new file

Returning the number of files (*delegation*)

Index numbering



Retrieving from the collection

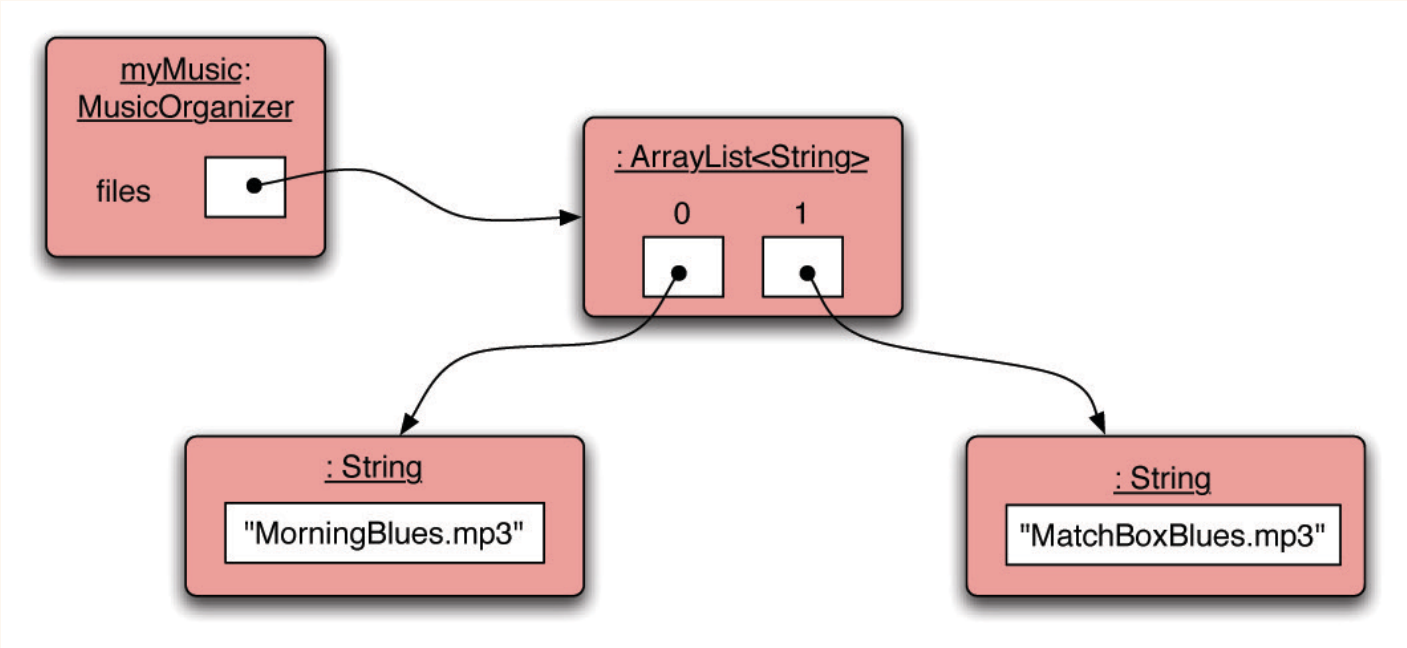
```
public void listFile(int index)
{
    if(index >= 0 && index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
    }
    else {
        // This is not a valid index.
    }
}
```

Index validity checks

Retrieve and print the file name

Needed? (Error message?)

Removal may affect numbering



music-organizer-v1

The general utility of indices

Using integers to index collections has a general utility:

- 'next' is: **`index + 1`**
- 'previous' is: **`index - 1`**
- 'last' is: **`list.size() - 1`**
- 'the first three' is: the items at indices **`0`**, **`1`**, **`2`**

We could also think about accessing items in sequence: **`0`**, **`1`**, **`2`**, ...

Review

Collections allow an arbitrary number of objects to be stored.

Class libraries usually contain tried-and-tested collection classes.

Java's class libraries are called *packages*.

We have used the **`ArrayList`** class from the **`java.util`** package.

Agenda

- Recap: Collections, ArrayList
- Program flow: if/ else, loops
- Program flow: iteration, for-each
- Program flow: while (searching)
- …

Review

Items may be added and removed.

Each item has an index.

Index values may change if items are removed (or further items added).

The main **`ArrayList`** methods are **`add`**, **`get`**, **`remove`** and **`size`**.

`ArrayList` is a *parameterized* or *generic* type.

ArrayList operations

```
import java.util.ArrayList;


ArrayList<String> lang = new ArrayList<>();
lang.add("Java");
lang.add("Python");
lang.add("C++");
System.out.println("ArrayList: " + lang);
System.out.println("First language: " + lang.get(0));
languages.remove("Python");
System.out.println("After removal: " + languages);
```

Operation	Method Example	Description
Add element	`` `list.add("Java")` ``	Adds an element to the end of the list
Get element	`` `list.get(0)` ``	Retrieves element at index 0
Set element	`` `list.set(1, "Python")` ``	Replaces element at index 1
Remove element	`` `list.remove(2)` ``	Removes element at index 2
Remove all	`` `list.clear()` ``	Removes all elements
Get size	`` `list.size()` ``	Returns the number of elements
Check if empty	`` `list.isEmpty()` ``	Returns true if list is empty
Find index	`` `list.indexOf("Java")` ``	Returns index of first occurrence
Contains element	`` `list.contains("Java")` ``	Checks if element exists
Convert to array	`` `list.toArray(new String)` ``	Converts ArrayList to array

Grouping objects

Collections and the **for-each loop**

Iteration

We often want to perform some actions an arbitrary number of times.

- E.g., print all the file names in the organizer. How many are there?

Most programming languages include *loop statements* to make this possible.

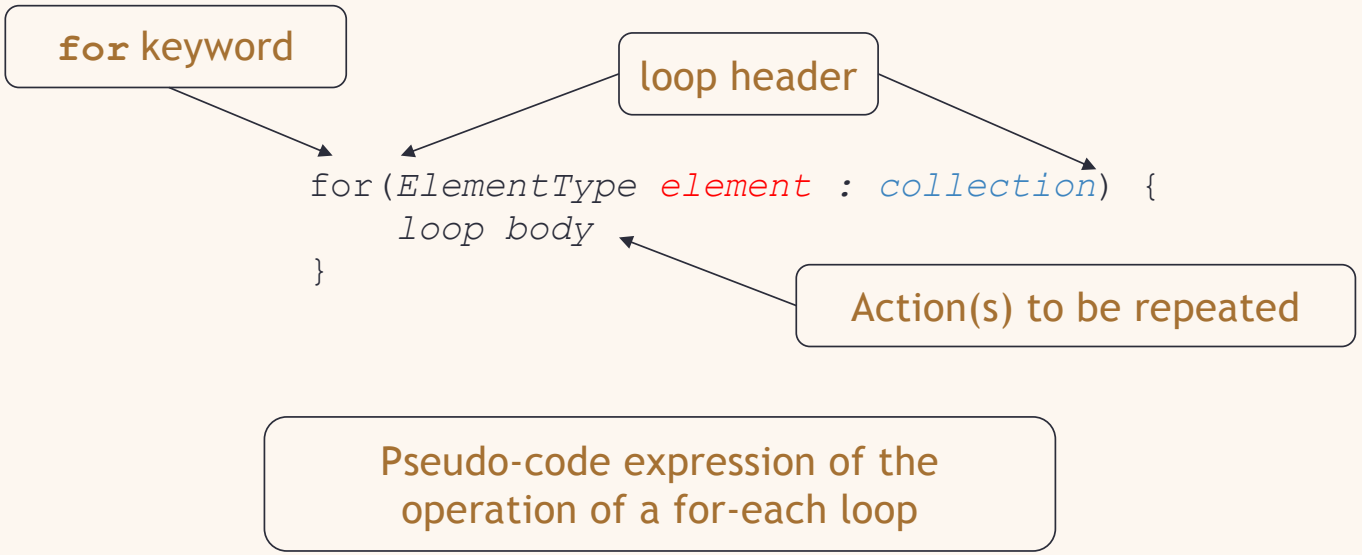
Java has several sorts of loop statement.

- We will start with its *for-each loop*.

Iteration fundamentals

- The process of repeating some actions over and over.
- Loops provide us with a way to control how many times we repeat those actions.
- With a collection, we often want to repeat the actions: *exactly once for every object in the collection*.

For-each loop pseudo code

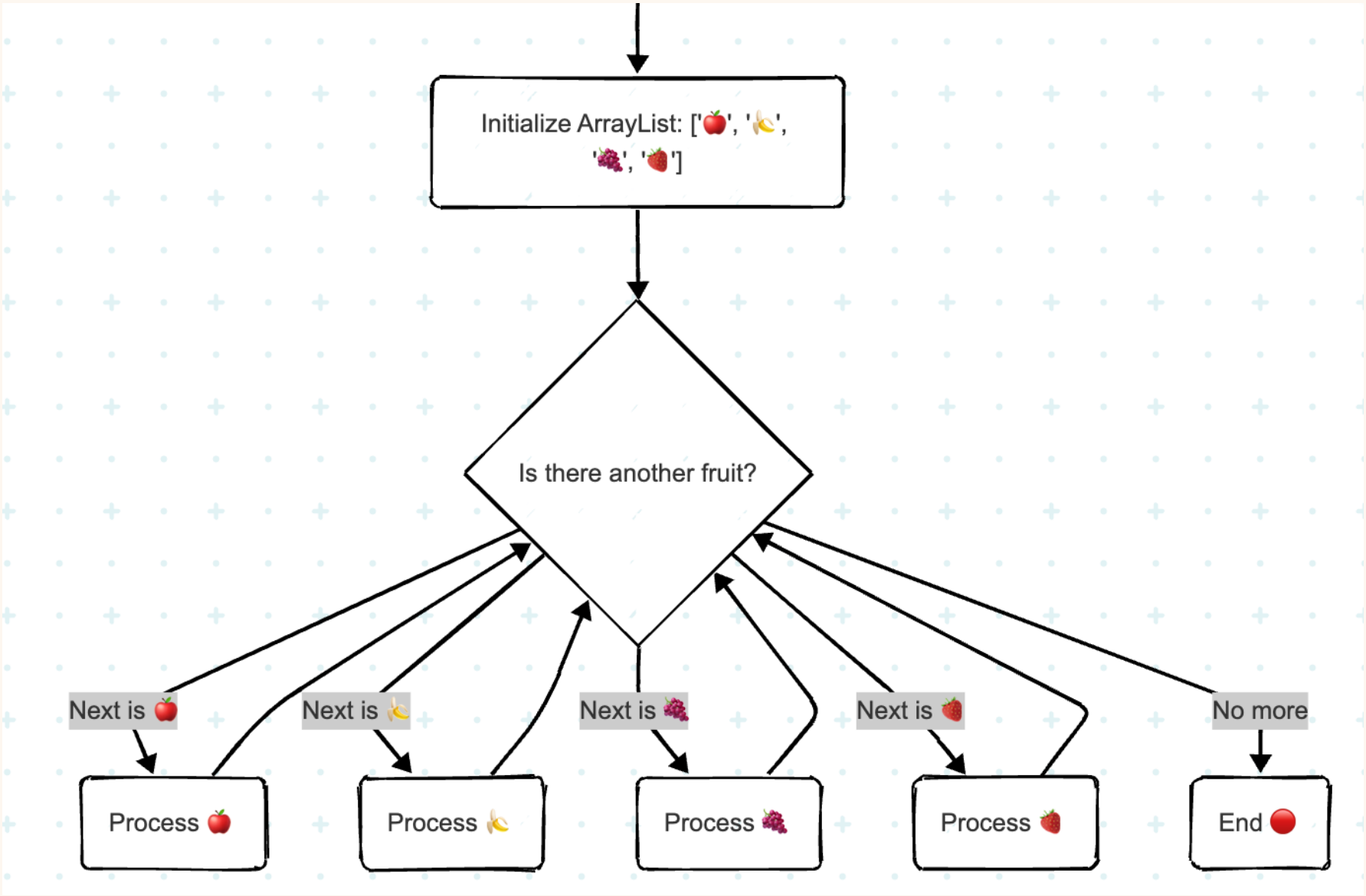


Using each *element* in *collection* in order, do the things in the *loop body* with that *element*.

A Java example

```
/**
 * List all file names in the organizer.
 */
public void listAllFiles()
{
    for(String filename : files) {
        System.out.println(filename);
    }
}
```

Using each *element filename* in *collection files* in order, do the things in the *loop body* with that *element*.


Initialize ArrayList: ['🍎', '🍌', '🍇', '🍓']
Is there another fruit?
Next is 🍎
Next is 🍌
Next is 🍇
Next is 🍓
No more
Process 🍎
Process 🍌
Process 🍇
Process 🍓
End 🔴

Review

- Loop statements allow a block of statements to be repeated.
- The for-each loop allows iteration over a whole collection.
- With a for-each loop *every* object in the collection is made available *exactly once* to the loop's body.

Selective processing

Statements can be nested, giving greater selectivity to the actions:

```
public void findFiles(String searchString)
{
    for(String filename : files) {
        if(filename.contains(searchString)) {
            System.out.println(filename);
        }
    }
}
```

Using each *element filename* in *collection files* in order, do the things in the *loop body* with that *element*.

contains gives a partial match of the filename; use equals for an exact match

Critique of for-each

Easy to write.

Termination happens naturally.

The collection cannot be changed by the actions.

There is no index provided.

- Not all collections are index-based.

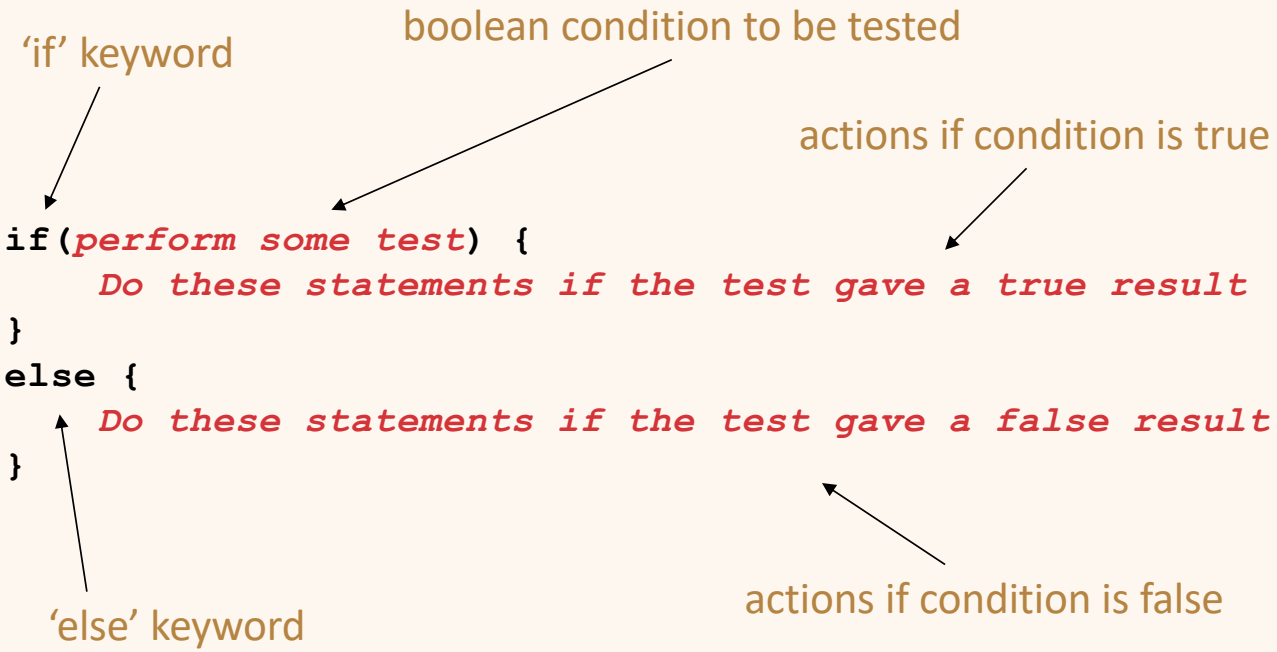
We can't stop part way through;

- e.g., if we only want to find the first match.

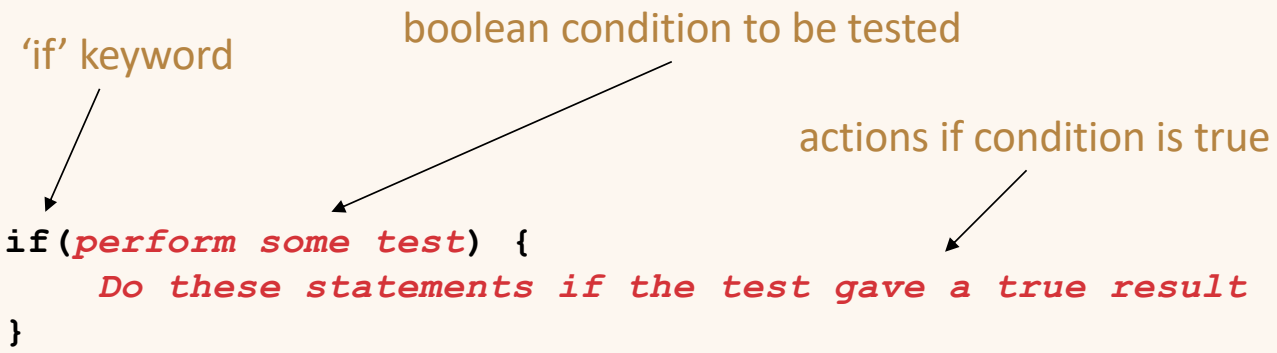
It provides 'definite iteration' – aka 'bounded iteration'.

While-loop

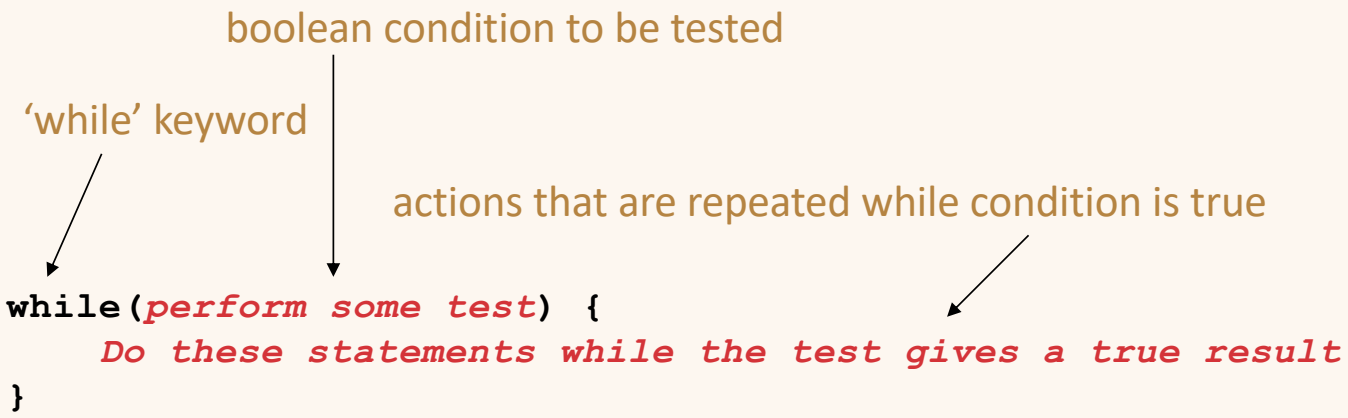
If Statement recap



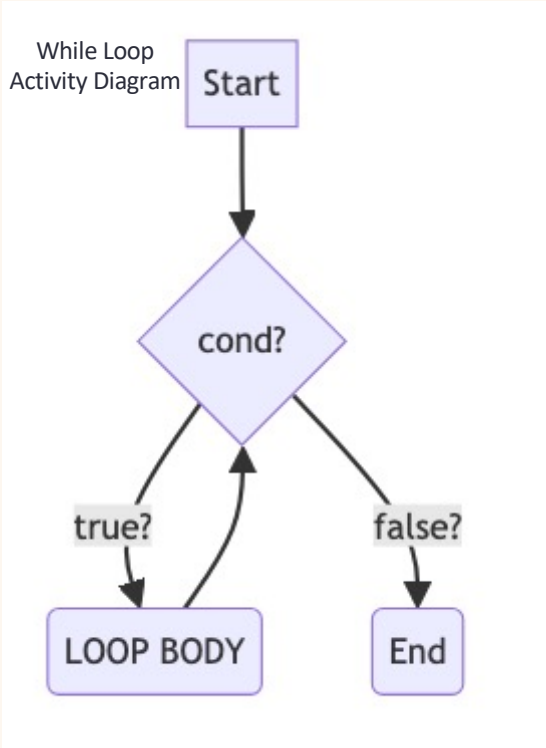
If without else clause

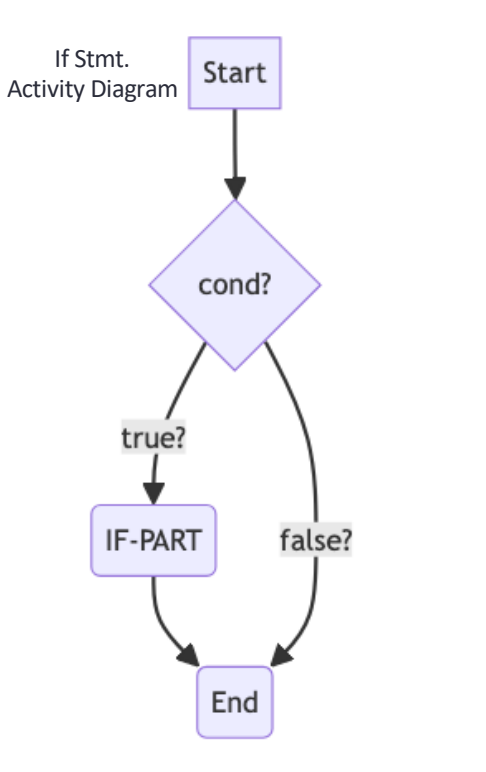


while loop


boolean condition to be tested
'while' keyword
actions that are repeated while condition is true
while(perform some test) {
Do these statements while the test gives a true result
}

While vs. If


While Loop
Activity Diagram
Start
cond?
true?
false?
LOOP BODY
End


If Stmt.
Activity Diagram
Start
cond?
true?
IF-PART
false?
End

Elements of the 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.

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*.

Searching

A fundamental activity.

Applicable beyond collections.

Necessarily indefinite.

We must code for both success and failure – nowhere else to look.

Both must make the loop's condition *false*, in order to stop the iteration.

A collection might be empty to start with.

Finishing a search

How do we finish a search?

Either there are no more items to check:

- `index >= files.size()`

Or the item has been found:

- `found == true`
  `found`
  `! searching`

Continuing a search

We need to state the condition for *continuing*:

So the loop's condition will be the *opposite* of that for finishing:

```
index < files.size() && ! found
index < files.size() && searching
```

NB: 'or' becomes 'and' when inverting everything.

Searching a collection

```
int index = 0;
boolean searching = true;
while(index < files.size() && searching) {
    String file = files.get(index);
    if(file.equals(searchString)) {
        // We don't need to keep looking.
        searching = false;
    }
    else {
        index++;
    }
}
// Either we found it at index,
// or we searched the whole collection.
```

Searching a collection

```
int index = 0;
boolean found = false;
while(index < files.size() && !found) {
    String file = files.get(index);
    if(file.equals(searchString)) {
        // We don't need to keep looking.
        found = true;
    }
    else {
        index++;
    }
}
// Either we found it at index,
// or we searched the whole collection.
```

Indefinite iteration

Does the search still work if the collection is empty?

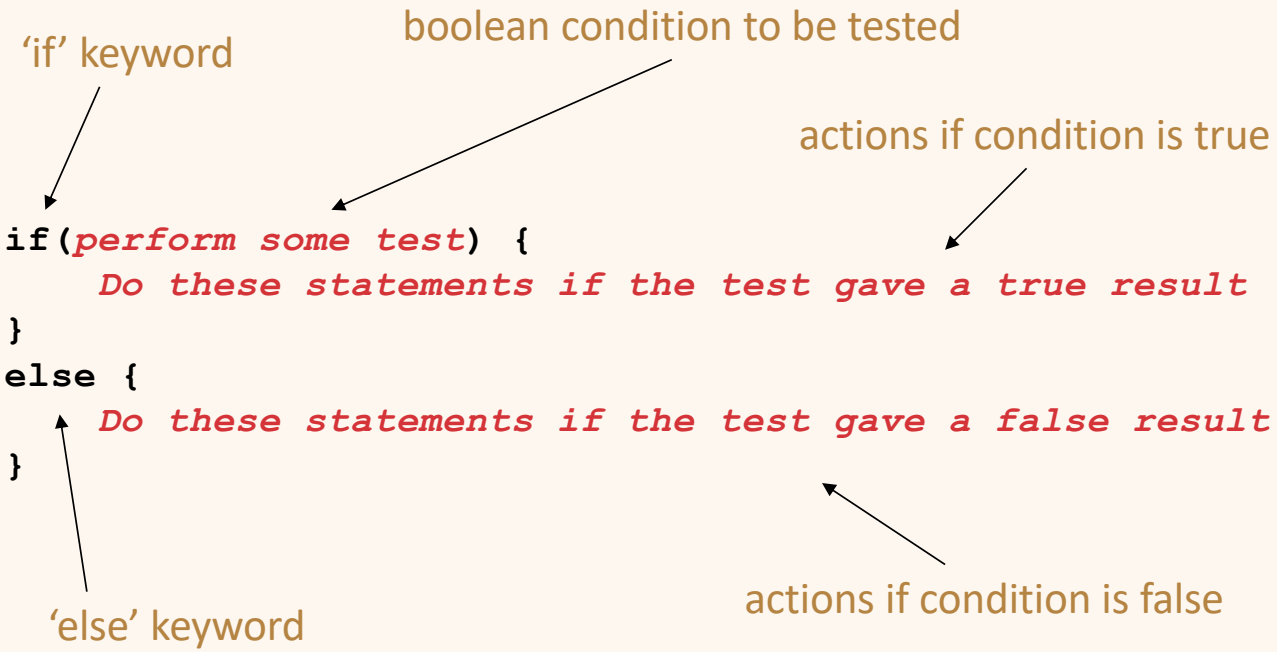
Yes! The loop's body won't be entered in that case.

Important feature of while:

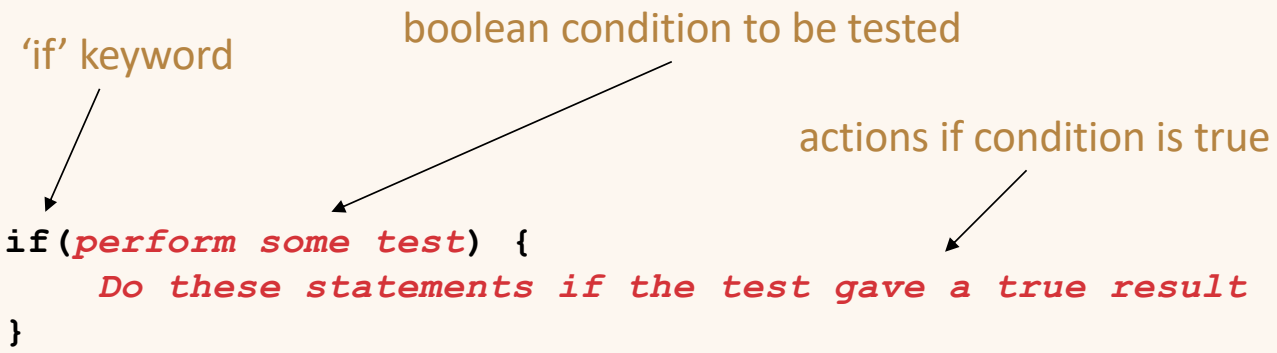
- The body can be executed *zero or more* times.

While-loop

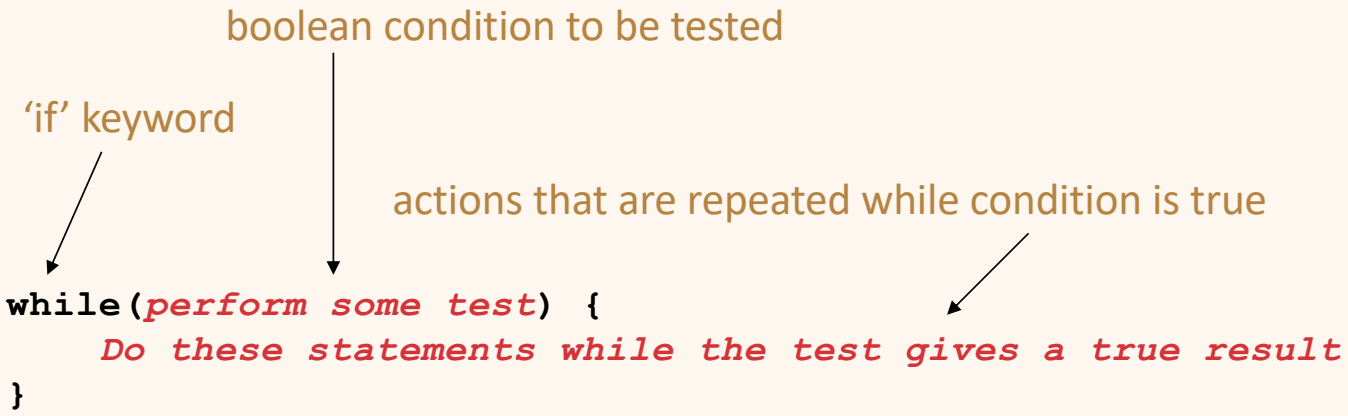
If Statement recap



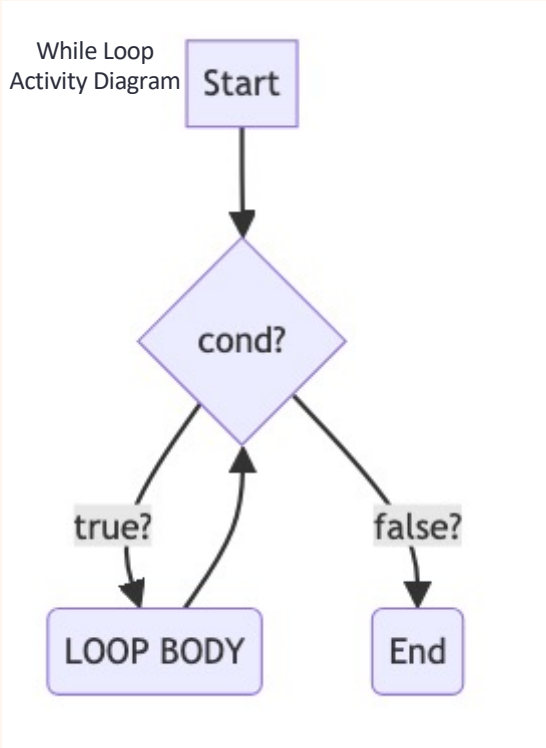
If without else clause

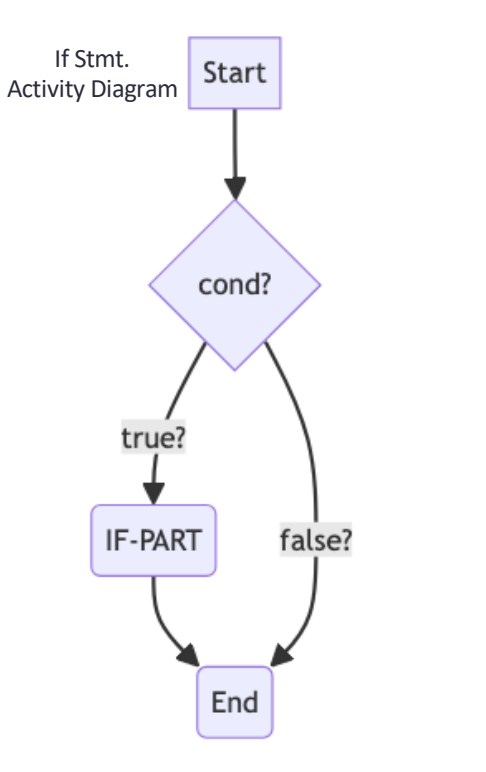


while loop


boolean condition to be tested
'if' keyword
actions that are repeated while condition is true
while(perform some test) {
Do these statements while the test gives a true result
}

While vs. If


While Loop
Activity Diagram
Start
cond?
true?
false?
LOOP BODY
End


If Stmt.
Activity Diagram
Start
cond?
true?
IF-PART
false?
End