# Grouping objects

Introduction to collections – Part 2

5.0

## Grouping objects

Collections and the for-each loop

## Main concepts to be covered

- Collections
- Loops: the for-each loop

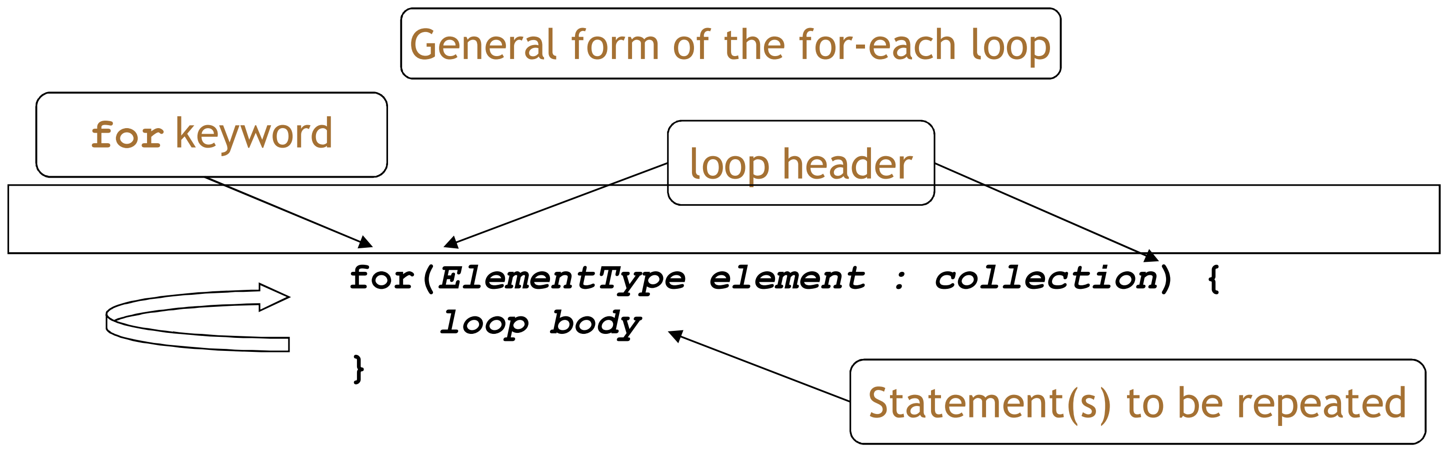
## Iteration

- 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

- We often want to repeat some actions over and over.
- Loops provide us with a way to control how many times we repeat those actions.
- With collections, we often want to repeat things once for every object in a particular collection.

# For-each loop pseudo code


General form of the for-each loop
for keyword
loop header
Statement(s) to be repeated

```
for(ElementType element : collection) {
    loop body
}
```

Pseudo-code expression of the actions of a for-each loop

For each *element* in *collection*, do the things in the *loop body*.

## A Java example

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

for each *filename* in *files*, print out *filename*

## Review

- Loop statements allow a block of statements to be repeated.
- The for-each loop allows iteration over a whole collection.

## Selective processing

- Statements can be nested, giving greater selectivity:

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

## Critique of for-each

- Easy to write.
- Termination happens naturally.
- The collection cannot be changed.
- There is no index provided.
  - Not all collections are index-based.
- We can't stop part way through;
  - e.g. find-the-first-that-matches.
- It provides 'definite iteration' – aka 'bounded iteration'.

## Grouping objects

Indefinite iteration - the while loop

## Main concepts to be covered

- The difference between definite and indefinite (unbounded) iteration.
- The while loop

## Search tasks are indefinite

- We cannot predict, *in advance*, how many places we will have to look.
- Although, there may well be an absolute limit – i.e., checking every possible location.
- '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 each object in a collection.
- Sometimes we require more variation than this.
- We use a boolean condition to decide whether or not to keep going.
- A while loop provides this control.

# While loop pseudo code

General form of a while loop

**while** keyword

boolean test

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

Statements to be repeated

Pseudo-code expression of the actions of a while loop

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

## Looking for your keys

```
while(the keys are missing) {
    look in the next place;
}
```

Or:

```
while(not (the keys have been found)) {
    look in the next place;
}
```

## Looking for your keys

```
boolean searching = true;
while(searching) {
    if(they are in the next place) {
        searching = false;
    }
}
```

Suppose we don't find them?

## A Java example

```
/**
 * 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*

## 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 be an *infinite loop*.

## Searching a collection

- A fundamental activity.
- Applicable beyond collections.
- Necessarily indefinite.
- We must code for both success and failure – exhausted search.
- Both must make the loop's condition *false*.
- The collection might be empty.

## *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

- With a while loop 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 found = false;
while(index < files.size() && !found) {
    String file = files.get(index);
    if(file.contains(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 will be executed *zero or more* times.

## While without a collection

```
// Print all even numbers from 2 to 30.
int index = 2;
while(index <= 30) {
    System.out.println(index);
    index = index + 2;
}
```

## 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

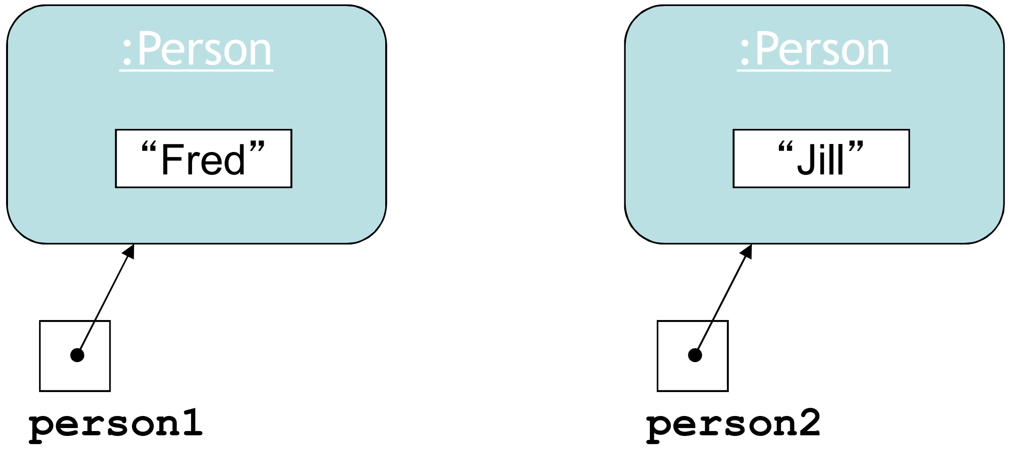
```
if(input.equals("bye")) {
    ...
}
```

tests equality

**Always** use `.equals` for text equality.

## Identity vs equality 1

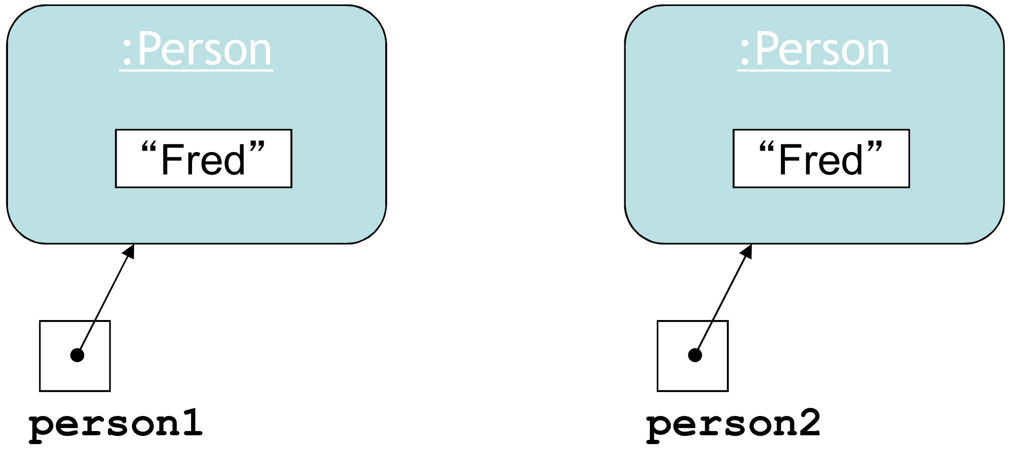
Other (non-String) objects:



**`person1 == person2 ?`**

## Identity vs equality 2

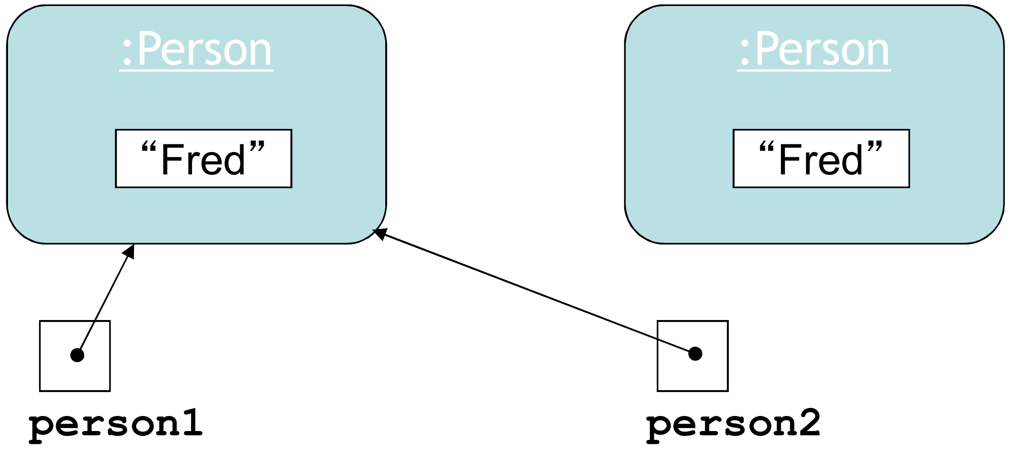
Other (non-String) objects:



**`person1 == person2 ?`**

## Identity vs equality 3

Other (non-String) objects:

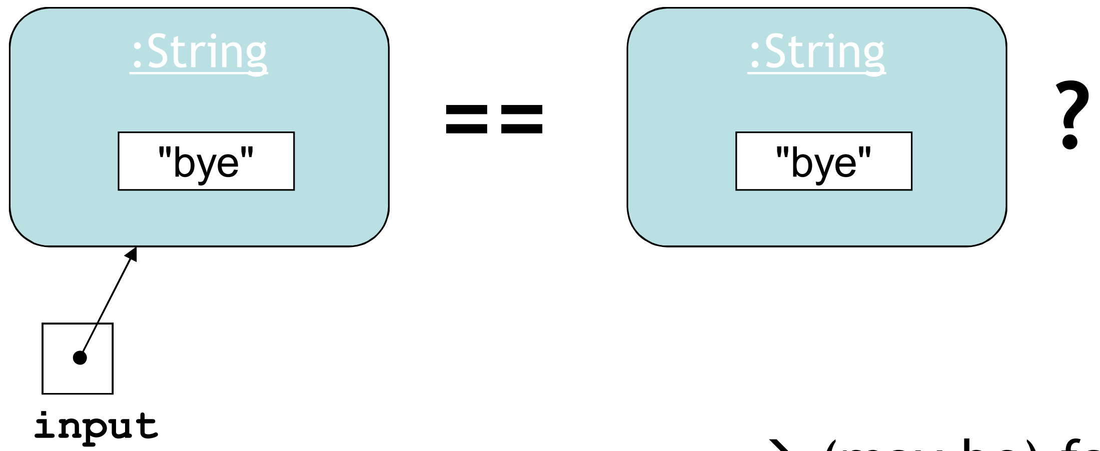


```
person1 == person2 ?
```

## Identity vs equality (Strings)

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

== tests identity

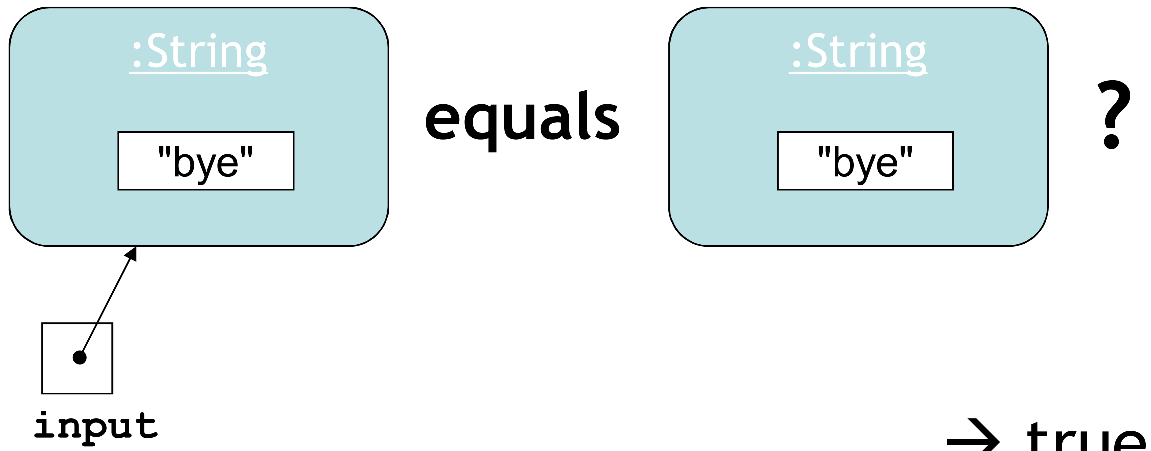


→ (may be) false!

## Identity vs equality (Strings)

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

equals tests equality



→ true!

## The problem with Strings

- The compiler merges identical **`String`** literals in the program code.
  - The result is reference equality for apparently distinct **`String`** objects.
- But this cannot be done for identical strings that arise outside the program's code;
  - e.g., from user input.

## Moving away from String

- Our collection of String objects for music tracks is limited.
- No separate identification of artist, title, etc.
- A **`Track`** class with separate fields:
  - **`artist`**
  - **`title`**
  - **`filename`**