University of HUDDERSFIELD

CFS2160: Software Design and Development

Lecture 9: Collections

Grouping related objects.

Tony Jenkins
A.Jenkins@hud.ac.uk

Compound Data Types

The variables we have used so far can store *one* value.

This is more complex that it seems:

- A student object might have two strings, a number, a date, and more.
- But, overall, it has a single value: its *state*.

Compound Data Types

The variables we have used so far can store *one* value.

So the value has a specific *type* - primitive type or object.

A compound data type can store several values: usually (but not always) of the same type - again, primitive or object.

Such types are commonly called "lists" or "arrays".

Compound Data Types

The variables we have used so far can store *one* value.

So the value has a specific *type* - primitive type or object.

A compound data type can store several values: usually (but not always) of the same type - again, pri

Such types are commonly called "lis

These concepts exist in all languages, although the names and the details differ.

Compound Data Types

The variables we have used so far can store *one* value.

So the value has a specific *type* - primitive type or object.

A compound data type can store several values: usually (but not always) of the same type - again, pri

Such types are commonly called "lis

In Python, for example, we met tuples, lists and dictionaries. There is no "array", although a list is very like an array.

Grouping Objects

Many applications involve collections of objects:

- Personal Organisers.
- Library Catalogues.
- Student Record Systems.

The number of items to be stored varies as items are added or deleted.

Patterns: Abstraction

Those three applications are all, basically, the same.

They all:

- Store records.
- Provide CRUD operations.
- Have some sort of reporting.

It follows that this is a *pattern*.

Patterns: Abstraction

Those three applications are all, basically, the same.

They all:

- Store records.
- Provide CRUD operations.
- Have some sort of reporting.

It follows that this is a *pattern*.

Spotting patterns like this is "Abstraction".
It's a key skill in programming.

Patterns: Abstraction

Those three applications are all, basically, the same.

They all:

- Store records.
- Provide CRUD operations.
- Have some sort of reporting.

It follows that this is a *pattern*.

And as it's a common pattern it follows that it's all been done before.
So we need to look in the package libraries.

Class Libraries

These are collections of useful classes.

Their existence removes the need to write everything from scratch.

Java calls its libraries *packages*.

Grouping objects is a recurring requirement.

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

Class Libraries

These are collections of useful classes.

Their existence removes the need to write everything from scratch.

Java calls its libraries *packages*.

Grouping objects is a recurring requi

- The `java.util` package co

Java *packages* are basically the same thing as Python *modules*.

Java Collections

As with many things in Java, there are many ways to manage collections.

Many ways.

We will use an `ArrayList` because it is general-purpose and easy to understand.

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayList.html

Creating a Collection

```
private ArrayList <String> notes;
```

This specifies:

- The type of collection: `ArrayList`.
- The type of objects it will contain: `<String>`
- The identifier we will use for it: `notes`

This creates an "ArrayList of String".

Features of the ArrayList

An `ArrayList` increases its capacity as necessary.

- ➢ The initial size can (optionally) be provided.

It keeps a private count of its size (`size()` accessor).

It keeps the objects in order.

Objects can be added, removed, searched for, and so on …

Features of the ArrayList

An `ArrayList` increases its capacity as necessary.

- ➢ The initial size can (optionally) be provided.

It keeps a private count of its size (`size()` accessor).

It keeps the objects in order.

Objects can be added, removed, sea

Details of how all this works are hidden.
This matters not: all we need to be able to do is read and understand the API docs.

Using an ArrayList

Declare an ArrayList of Strings.

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

Using an ArrayList

Declare an ArrayList of Strings.

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

What is the initial capacity of this list?
How would we find out?

Using an ArrayList

Declare an ArrayList of Strings.

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

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayList.html#%3Cinit%3E()

Using an ArrayList

Declare an ArrayList of Strings.

(That was actually a trick question: it hasn't been created yet.)

Using the default constructor.

(If we read the docs, we now know this has an initial capacity of 10.)

```
private ArrayList <String> files;

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

Using an ArrayList

Declare an ArrayList of Strings.

```
private ArrayList <String> files;

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

Using the default constructor.

```
files.add ("cheese.txt");
files.add ("toast.doc");
```

The simplest operation is to add some Strings to the list.

Using an ArrayList

Declare an ArrayList of Strings.

```
private ArrayList <String> files;

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

Using the default constructor.

The simplest operation is to add some Strings to the list.

And then maybe find out how big the collection is.

```
files.add ("cheese.txt");
files.add ("toast.doc");

System.out.println (files.size ());
```

Retrieving an Object

One (not ideal) way to retrieve an object is via its index.

Each object in the collection has an *index* value.

You will not be surprised that the indexes start at zero.

The last element is therefore at the index given by the result of the expression "`size () - 1`".

Retrieving an Object

One (not ideal) way to retrieve an object is via its index.

This is not ideal because, in general, indexes are best kept internal:

- A user will not know what they are.
- The index of an object will change as objects are deleted.

But it is a good place to start …

Retrieving an Object

Assuming we have the required index in a variable.

```
index = 3;
```

Retrieving an Object

Assuming we have the required index in a variable.

It's easy enough to check if it's in the range of valid indexes.

```
index = 3;

if(index >= 0 && index < files.size ())
{
```

Retrieving an Object

Assuming we have the required index in a variable.

It's easy enough to check if it's in the range of valid indexes.

The `get` method will find it.

And then it can be printed.

```
index = 3;

if (index >= 0 && index < files.size ()) {
    String filename = files.get (index);
    System.out.println (filename);
}
else {
    System.out.println ("Not found.");
}
```

Retrieving an Object

Assuming we have the required index in a variable.

It's easy enough to check if it's in the range of valid indexes.

The `get` method will find it.

And then it can be printed.

Without using a redundant variable!

```
index = 3;

if (index >= 0 && index < files.size ()) {

    System.out.println (files.get (index));
}
else {
    System.out.println ("Not found.");
}
```

Iteration

A better approach is to *iterate* through the collection, looking for the element we want.

Iteration

A better approach is to *iterate* through the collection, looking for the element we want.

Remembering that it might not be there at all!

Iteration

A better approach is to *iterate* through the collection, looking for the element we want.

Remembering that it might not be there at all!

In fact, depending on exactly how we're searching there could be no, one, or many matches.

Reminder: Types of Loop

There are three possibilities for a loop:

1. The number of iterations is known before the loop starts.
2. The number of iterations is unknown, but will always be at least one.
3. The number of iterations is unknown (and could be zero).

These can all be done with one type of statement but for convenience most languages provide different constructs.

Loops and Collections

So what type of loop do we need for a Collection?

A program needs to process each element of the Collection.

It could have zero, one or "many" elements.

But, before the loop begins this number *is* known - it is the current size of the collection.

This is therefore a *determinate* loop.

Looping Over a Collection

To find and print a specific item:

1. Start at the first element (index 0).
2. Stop once the required item has been found.
3. Keep going until it is found.

Note that the *maximum* number of iterations (loops) can be predicted (it's the size of the collection).

Looping Over a Collection

To find and print all items matching a pattern:

1. Start at the first element (index 0).
2. Stop once the last element has been processed.
3. Keep going by going to the next element.

Note that the *actual* number of iterations (loops) can be predicted (it's the size of the collection).

Java Loops

Java supports many kinds of loops:

- `while` loops.
  - Basically the same as in Python.
- `do … while` loops.
  - For indeterminate loops that will execute at least once.
- `for` loops.
  - Similar to `"for x in range (10)"` in Python.
- `for` each loops.
  - For collections (!).

for-each

This involves specifying the collection to be used …

… and the actions to be applied.

```
for (ElementType element : collection) {
    // Do these statements.
}
```

Note the brackets. Note no semicolon.

Iteration with for-each

```
/**
 * Print all file names in the list.
 */

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

Dissection

The variable `filename` first takes the value of the first element in the collection.

The braces {} mark out the statements to be applied.

Once complete, `filename` takes on the value of the next element in the collection.

(What order will they be in?)

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

Dissection

The variable `filename` first takes the value of the first element in the collection.

The braces {} mark out the statements to be applied

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

This snippet could be part of a handy method.

Searching

This basic structure would allow searches for Strings that, say, contained some pattern.

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

Searching

This basic structure would allow searches for Strings that, say, contained some pattern.

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

Searching

This basic structure would allow searches for Strings that, say, contained some pattern.

Or were equal to some provided value.

```
public void findString (String search)
{
  for (String filename : files) {
    if (filename.equals (search)) {
      System.out.println (filename);
    }
  }
}
```

Searching

This basic structure would allow searches for Strings that, say, contained some pattern.

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

With nested code like this is it *essential* to keep on top of indentation so as to show the code structure.

Summarising For-each

This loop is easy to write. It's a standard *recipe*.

Termination happens naturally.

There is no need to access the index (some collections are not index-based).

It will work with many "Collection" types - refer to the docs for all the gory details.

Other Loops

for-each works through all elements in the collection.

This is fine, but is sometimes not what is required.

- Find the *first* matching file in the list.

We *could* program round this, but this would be cumbersome, inefficient and wasteful.

(This is an indeterminate loop.)

while Loops

```
while (condition) {
    // Statements executed.
    // Must eventually change "condition".
}
```

The statements execute as long as the condition is true.

The condition is checked *before* the first statement is executed.

Therefore, if the condition is initially false, they execute not at all.

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

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

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

See that the loss of for-each means that we have to fall back on indexes.

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  while (!found) {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  }
}
```

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

We create a *local* variable to serve as the condition for the loop.

It's often called a *sentinel*.

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  while (!found) {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  }
}
```

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

We create a *local* variable to serve as the condition for the loop.

We also need a counter so we can access each element.

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  while (!found) {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  }
}
```

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

We create a *local* variable to serve as the condition for the loop.

We loop so long as the sentinel is false.

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  while (!found) {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  }
}
```

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

We create a *local* variable to serve as the condition for the loop.

And increment the counter if we don't find the value we seek.

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  while (!found) {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  }
}
```

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

We create a *local* variable to serve as the condition for the loop.

Flipping the sentinel if we do find the value we seek.

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  while (!found) {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  }
}
```

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

We create a *local* variable to serve as the c

F

v

This is a <u>really bad</u> example.

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  while (!found) {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  }
}
```

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

We create a *local* variable to serve as the c

F

v

This is a <u>really bad</u> example.
Why?
Because this loop will always execute at least once.

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  while (!found) {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  }
}
```

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

We create a *local* variable to serve as the c

F

v

This is a <u>really bad</u> example.
Why?
And the code will fail if the value we seek is not in the collection.

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  while (!found) {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  }
}
```

do Loops

So, in this case we can use an alternative:

We use this:

```
do {
    // Statements.
    // Which must change the value of condition.
} while (condition);
```

Note the position of the condition gives a clue as to what is happening.

Searching

Remember that this code found *all* the filenames that contained the string.

We will refactor to find the first.

And use the best structure ...

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  do {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  } while (!found);
}
```

Searching

Remember that this code found *all* the file names that contained the string.

We will refactor to find the first.

And use the best structure ...

We could have written this code with any of the three types of loop.
This one feels best, but there's not much in it!

```
public void findString (String search)
{
  boolean found = false;
  int i = 0;

  do {
    String filename = files.get (i);
    if (filename.contains (search)) {
      System.out.println (filename);
      found = true;
    else {
      i ++;
    }
  } while (!found);
}
```

Searching

This is how it would work with for-each.
This is quite Pythonic, but maybe not the "done thing" in Java.

```
public void findString (String search)
{
  for (String filename : files) {
    if (filename.contains (search)) {
      System.out.println (filename);
      break;
    }
  }
}
```

Collections of Objects

We have used an ArrayList of Strings.

How would we make an ArrayList of `Employee` objects or, if it comes to that, `BankAccount` objects?

Collections of Objects

We have used an ArrayList of Strings.

How would we make an ArrayList of `Employee` objects or, if it comes to that, `BankAccount` objects?

Simples!

Strings are objects. So `BankAccount` objects work just the same way.

Sample Application

Imagine a simple "Phone Book" application.

It will store details of my contacts.

Sample Application

Imagine a simple "Phone Book" application.

It will store details of my contacts.

A Contact consists of two things: a name and a phone number, both of which are strings.

Sample Application

Imagine a simple "Phone Book" application.

It will store details of my contacts.

A Contact consists of two things: a name and a phone number, both of which are strings.

I need to be able to add to my Phone Book, print it out, and search it. (We'll ignore deletion for now.)

IntelliJ Demo Time