# Programming Foundations FIT9131



## Class libraries, collections, iteration





## Week 5

# Lecture outline

- Using Library classes
  - Java Collections
  - `ArrayList`
- Generic classes
- Fixed Size Arrays
- Loops (Repetitions)
  - `for-each loop`
  - `while loop`
  - `for loop`

# Class libraries

Java has a library of pre-defined classes which the programmer can access.

The class library contains many useful classes which are tried, tested and ready to use.



Using a pre-defined class library, we don't have to write everything from scratch.

Thousands of classes are available through the Java class library.

# Class libraries

**In the Java class library, classes that belong together are grouped into "*packages*".**



***Class libraries are a powerful feature of object-oriented programming*. They allow programmers to use pre-defined classes in their code, hence reducing their efforts and improving efficiency.**

# Reading class documentation

The Java standard library documentation shows details about all classes in the library.



This is called the **Java API** (*Application Programming Interface*) **documentation**.





The documentation is readable in a Web browser :

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

Provides an *interface* description for all library classes

# Using library classes

If you want to use an object of another class in a class declaration, then that other class must be accessible to the compiler.



So to use classes from the Java class library, they must be "imported" using an *import* statement.



- the import statement must be placed at the beginning of the code for a class, before the class declaration.



- the imported classes can then be used in the same way as any other user-defined classes from the current project.

# Importing classes and packages

**Single classes may be imported, eg :**

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



**Or, Whole packages can be imported, eg :**



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





**Note that classes from the *java.lang* package are automatically included in any Java program, so the last import statement in the example above is really not necessary.**

# Importing a class into a BlueJ project

To bring another class into an existing BlueJ project:

Select **Edit** from menu
Select **Add Class from file …**
Browse to find the class you want
Select the .java file with the class name



Or, you may manually import a class from the Java libraries using the `import` statement (as shown on the previous slides).

# Grouping objects

Often we want to group together a number of values or objects and treat them as a *group*.

- Many applications involve *collections of objects*, eg:
  - Telephone Book entries
  - Library catalogs
  - Student record system



- The number/type of items to be stored varies, but there are typically some *common operations*, eg:
  - items can be added, deleted, edited, sorted, etc

# Implementing Object Grouping

- It is easy to store single objects in a program : we just create a variable to represent each object, eg: `student`, `staff`, `tuteRoom`, etc.

- So how can we implement a "group of objects"?
  - let's start by simply creating individual variables for each object, and give the variables similar names, eg : `student1`, `student2`, `student3`, etc
  - this will work, but do you see a problem with this approach?

# Better solution : implementing Object Grouping

- Problem : the implementation shown on the previous slide quickly becomes impractical for large groups. For instance, what if you wish to have a group of 30000 students? Or, what if we do not even know how many students we will need, until the program is actually run?



- So we need a better and more flexible technique than creating a variable for each individual item in a group – *we want some way of "grouping" the objects*.

# Using Java Class libraries to group objects

- As indicated previously, Java has its libraries organised into *packages* of related classes. These packages handles different areas of programming, eg. maths, file input/output, etc.



- Grouping objects is a common requirement in programming.



  - the `java.util` package contains some special classes designed for grouping objects. These are often called "*Collections*".

# Java Collection : ArrayList

- We are going to learn about a very useful Java *Collection* Class called "`ArrayList`".



- This is commonly used to store a "*list*" of objects, e.g. : a list of `Student` objects, a list of `Date` objects, a list of `Prize` objects, etc.





- The `ArrayList` class provides the common methods used to manipulate the objects (so you do not need to write them yourself).

# Example : An organizer for music files

Let's examine the *music-organizer-v1* project (Chapter04 from textbook).



*Program Logic :*



- "Track" files (eg "`Gangnam Style.mp3`") may be added.

- there is no pre-defined limit to the number of files.
- it can tell how many file names are stored in the collection.
- it can list individual file names, using an index.

# Using a typical "collection" class

```
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<String>( );
    }

    ...
}
```

(1) importing a collection class called "**ArrayList**"

(2) declaring an attribute, **files**, to be a "***collection of Strings***".

(3) creating the actual collection, then assigning it to the attribute **files**. Note : **files** is an object – of type "**ArrayList of Strings**".

# Declaring Vs Creating objects

```
Person   student;
```

This statement **declares** a variable (an **object** in this case) called "**student**", to be of type (a **class** in this case) "**Person**".
Important : we cannot use this object as yet, because *no memory has been allocated* to it.

...



```
student = new Person("1234", "David Smith");
```



This statement uses the "**new**" operator to **create** a **Person** object, allocates memory for it, and returns a reference to it. That reference is then assigned to the variable **student**. Now we can use the object via **student**.

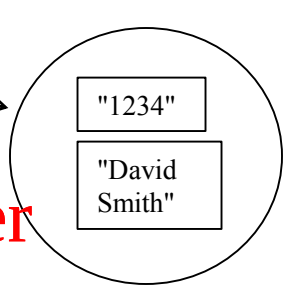
# Declaring and Creating objects at the same time

```
Person  student = new Person("1234", "David Smith");
```










It is a common practice to combine the previous 2 statements into one statement like this, to achieve the same effect. The final result is the same.

The above statement means : create a variable called **student**, Then assign to it a reference to a **Person** object (with ID "**1234**" and name "**David Smith**").

# Examples : Using objects

```
student.display();

String loginId = student.getId() + student.getSurname();
```

Examples of using methods



After an object has been created, we can call ***methods*** from the object to perform some tasks, eg. printing some values, returning a value, etc.
Note the use of the "***dot notation***" :

```
objectName.methodName(...).
```

# Declaring Collections

*Collections* can store an arbitrary number of elements.
*Each element is one single object.*



When declaring a collection we need to specify:

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





- Eg: `private ArrayList<String> files;`

This statement means : the attribute named `files` is an "`ArrayList of Strings`".

# Generic classes

- Collections are known as *parameterized* or *generic types*.
- An `ArrayList` implements "list" functionality:
  - `add`, `get`, `size`, etc.
- The <*type*> parameter specifies what type of objects we want a list of – eg :
  - `ArrayList<Person>`
  - `ArrayList<TicketMachine>`
  - etc

Classes which require such a parameter are known as *generic classes*

# Creating an ArrayList object

- In versions of Java prior to version 7 :
  - `files = new ArrayList<String>();`



- Java 7 introduced the 'diamond notation' :



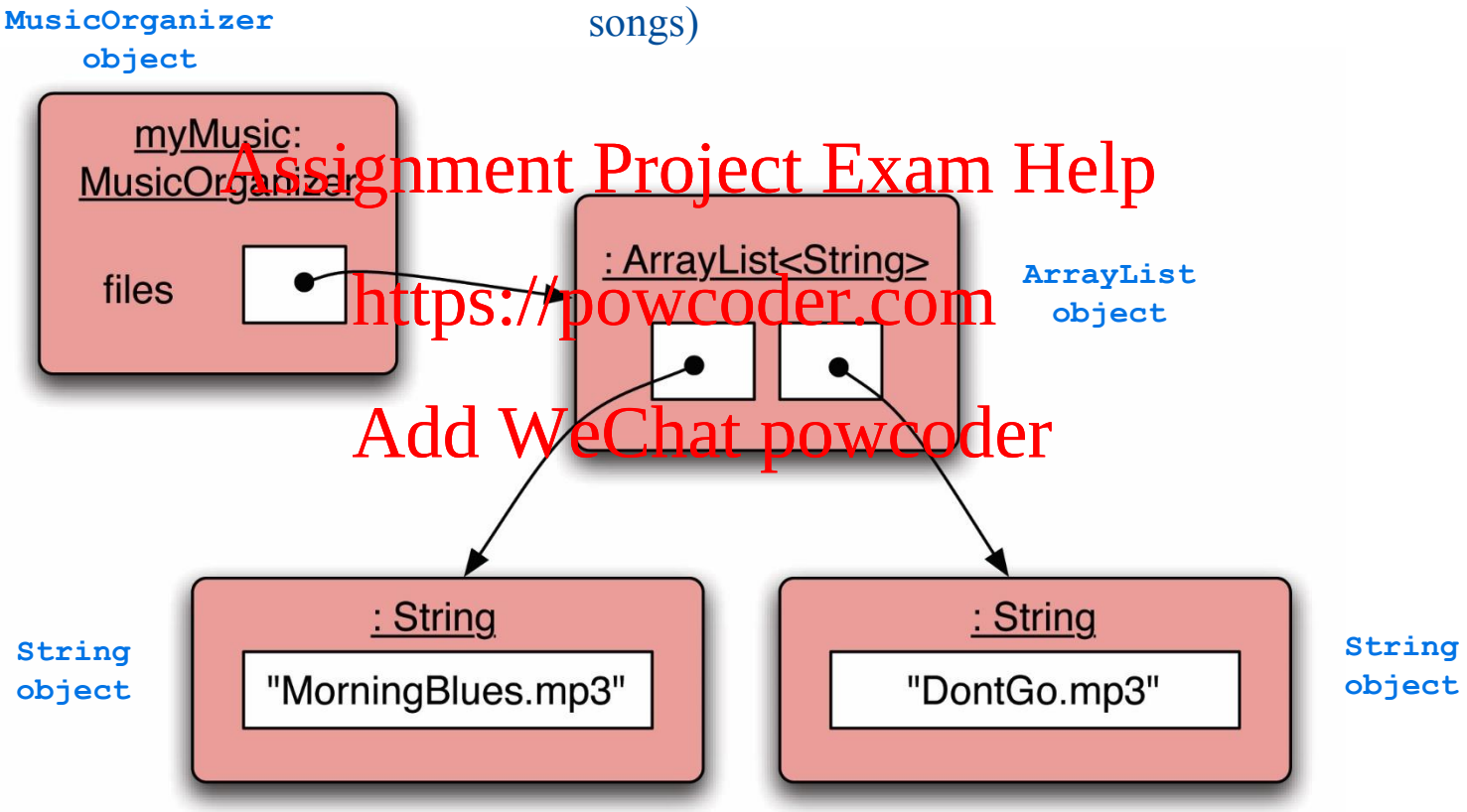
  - `files = new ArrayList<>();`



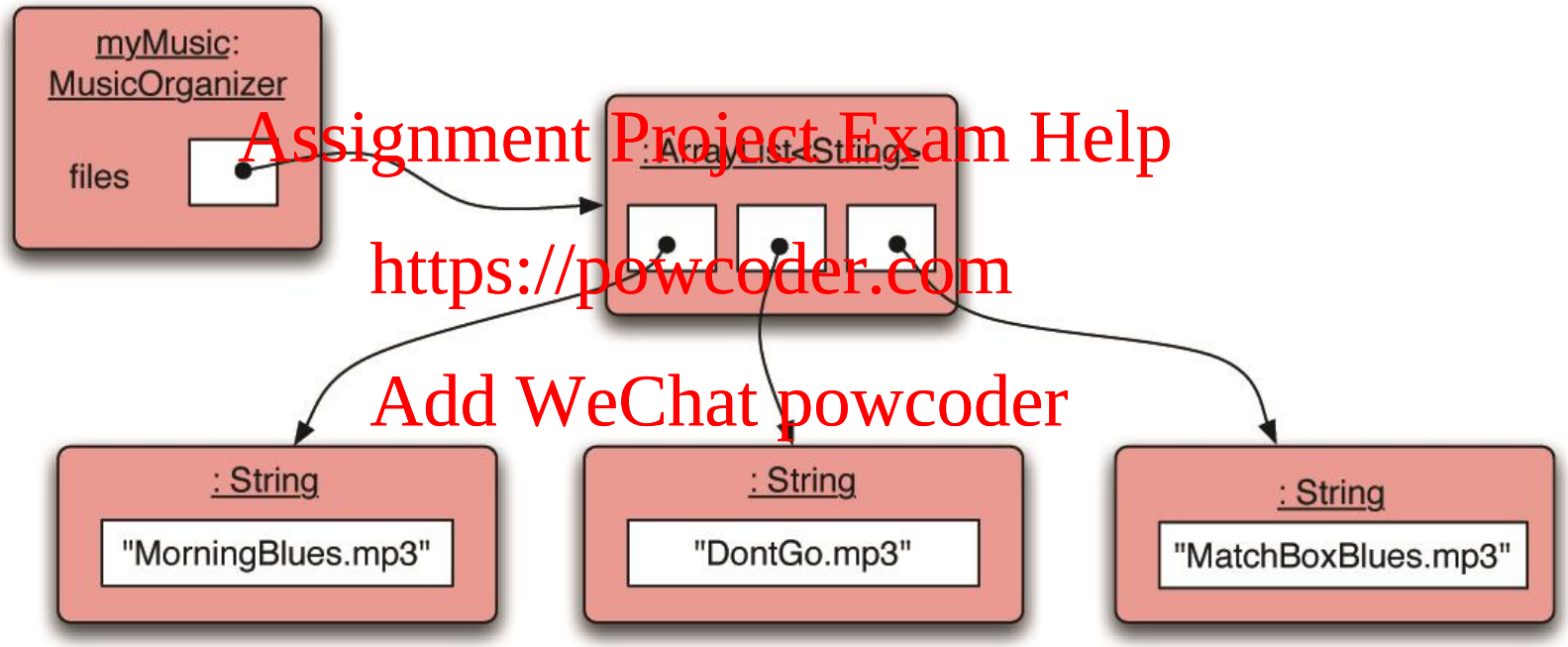
  - the type parameter can be inferred from the variable being assigned to
  - more convenient to declare and use

# Object structures with collections

For instance, a typical **`MusicOrganizer`** object may look like this. Note that this object contains an **`ArrayList`** of 2 **`String`** objects (representing 2 songs)



# Adding a third music file



# Features of the ArrayList collection

An `ArrayList` can increase its capacity as necessary.

It keeps a private count of its objects (accessed by the `size()` method).

It keeps the objects in order – objects may be retrieved in the same order they were inserted.



Details of *how* all this is done are hidden.

- Does this matter? Do we really need to know?
- Does not knowing *how it works* prevent us from using this collection?

# Using the collection

Note : the `add()` and `size()` methods shown below come from the `ArrayList` class.

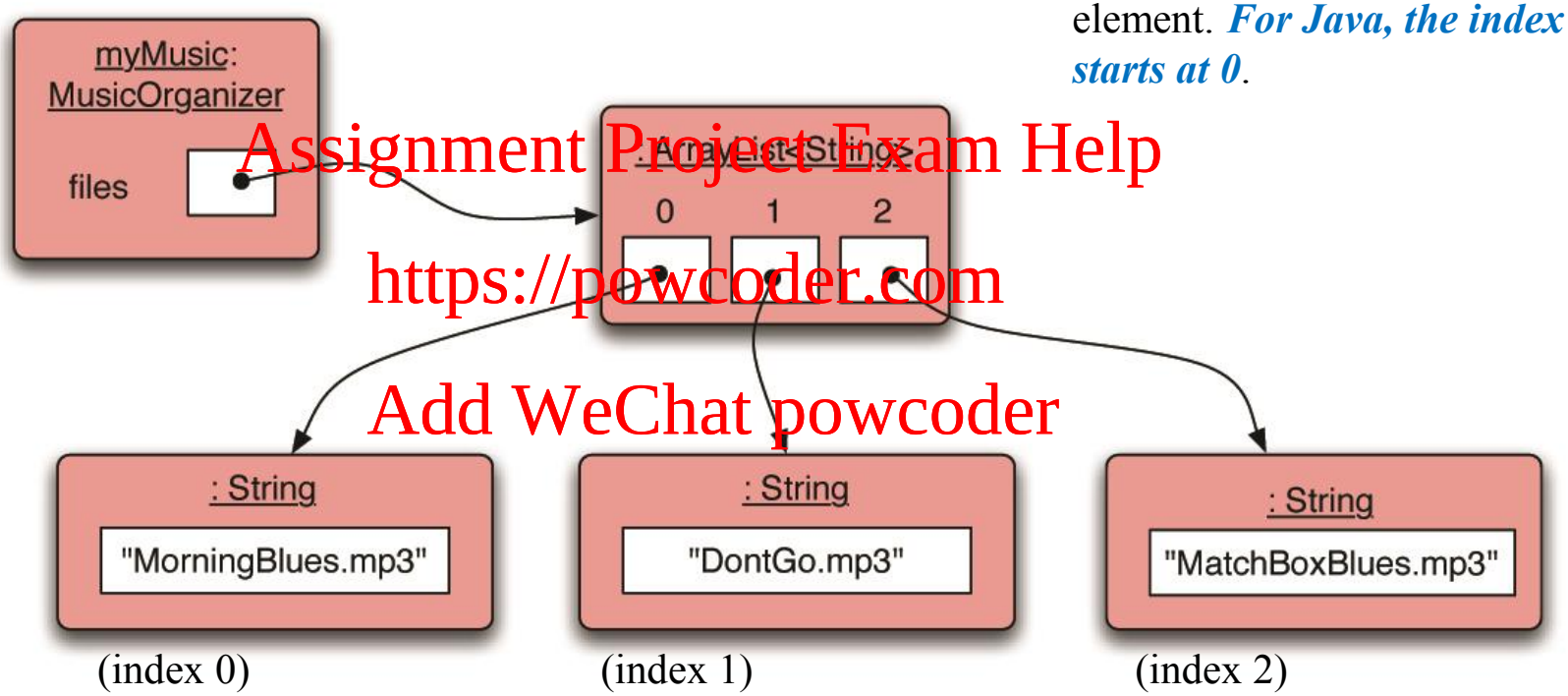
```
public class MusicOrganizer
{
    private ArrayList<String> files;

    ...

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

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

    ...
}
```

Adding a new file

Returning the number of files

# ArrayList Index numbering

Each element in an **`ArrayList`** has an ***index***, which is used to refer to the element. ***For Java, the index starts at 0***.



# Example : Retrieving an object

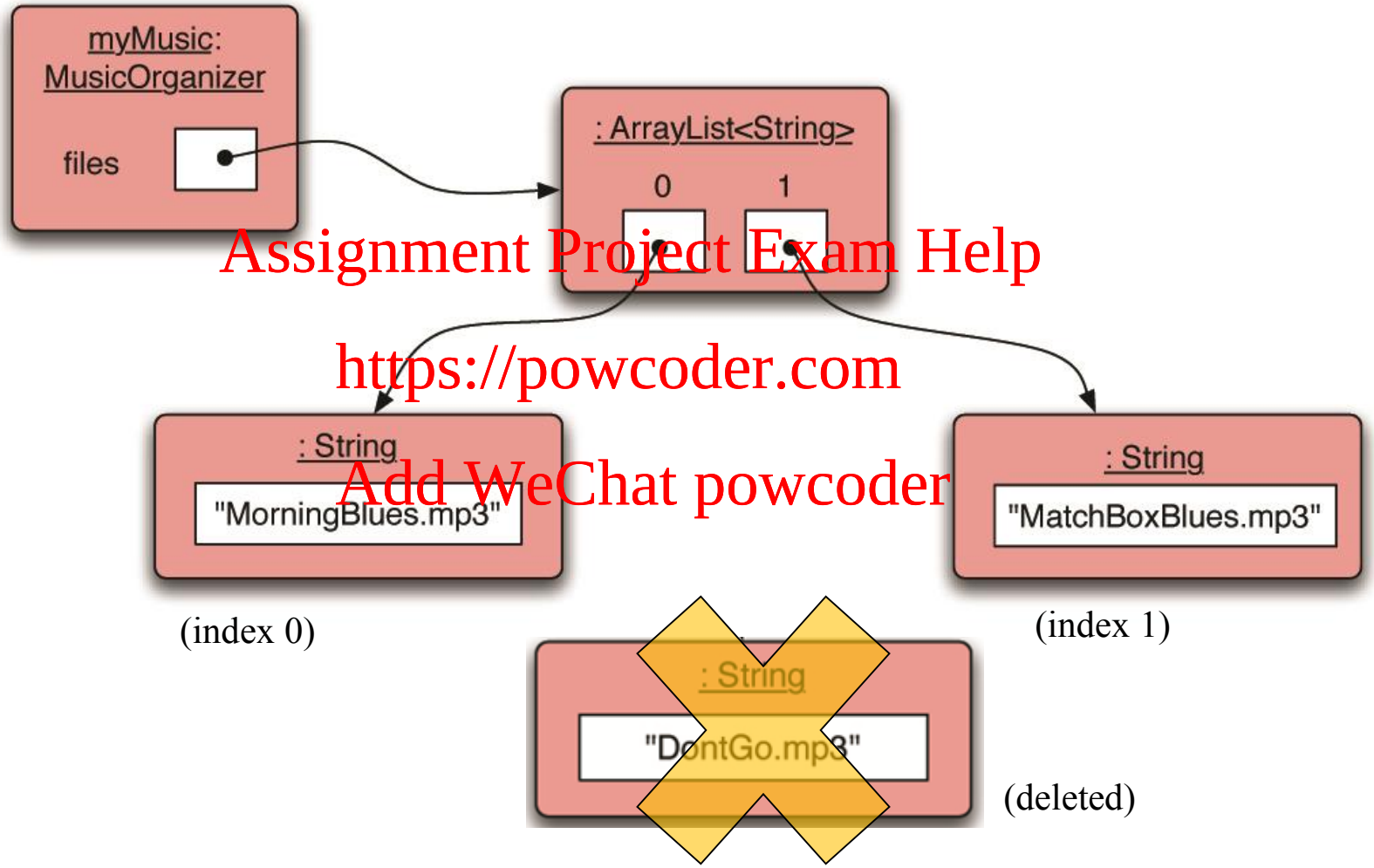
```
public void listFile(int index)
{
     if(index >= 0 &&
        index < files.size())
     {
          String filename = files.get(index);
          System.out.println(filename);
     }
     else
          System.out.println("Error : invalid index");
}
```

Index boundary checks

Retrieve and print the file name

Optional Error message

# Removal may affect numbering



# Review

Collections allow an arbitrary number of objects to be stored.

We have used the `ArrayList` class from the `java.util` package. In this collection:

- 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 parameterised or generic type.

# Programming Constructs

**There are 3 main kinds of programming constructs :**

1) *Sequence*
2) *Selection*
3) *Repetition* **(*or* *Iteration*, *or* *Looping*)**



**We have already learnt constructs (1) and (2). We are learning (3) this week.**

# Iteration (or loop, or repetition)

We often want to *repeatedly perform some actions a number of times*. For example:

- adding up a series of numbers
- calculating the final grade for each student in a unit



Most programming languages include special *loop statements* for this purpose.

Loops provide us with a way to *perform these actions repeatedly* and to control how many times we repeat those actions.

Looping is also called *repetition* or *iteration*.

# Iteration fundamentals

When dealing with collections, we often want to *repeatedly* perform some operation for every object in a particular collection.

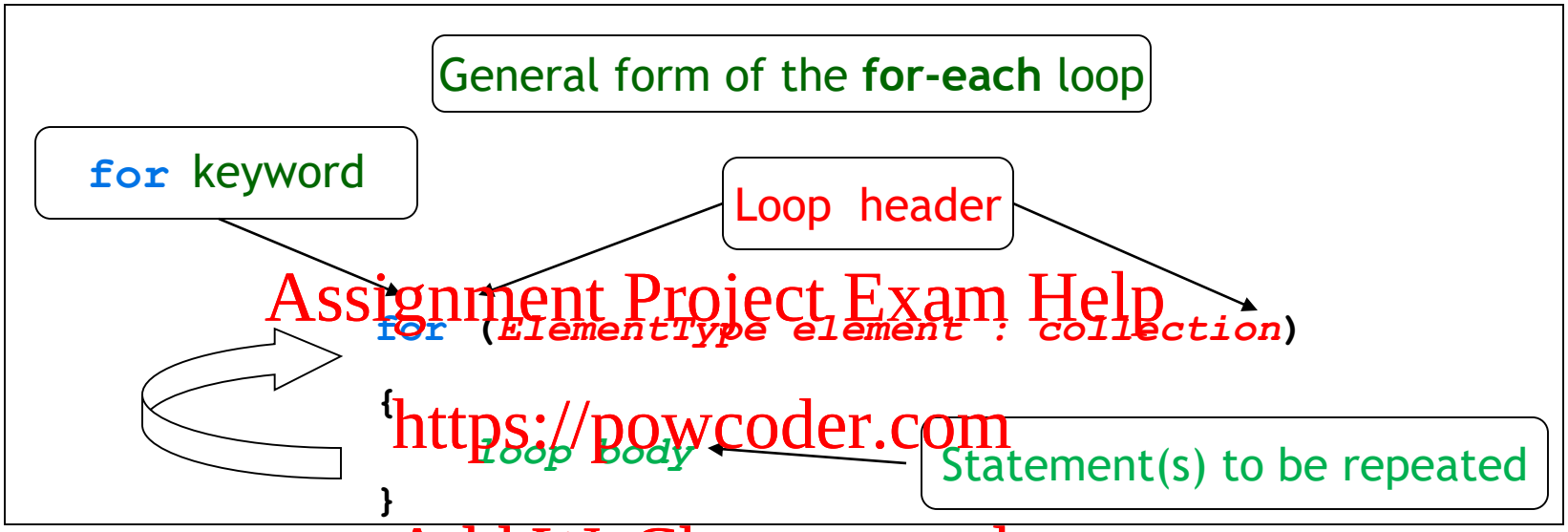
- Eg : printing out the names of all the student objects in a student database collection. Each "print" operation is identical for each student.



Java has several types of loop statements.

We will start with the *Java for-each loop*, which is design to be used with a Java Collection class.

# Syntax : *for-each* loop






The above means : For each *element (of type ElementType)*, in the given *collection*, *perform the statement(s)* 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);
}
```



In English, this “for-each loop” means :

for each *filename* in the collection of *files*, print out *filename*

# An alternative : the `while` loop

A *for-each* loop repeats the loop body for each object in the given collection.

Sometimes we require more variation than this. For example, we may want to read only part of a collection which satisfies certain conditions.

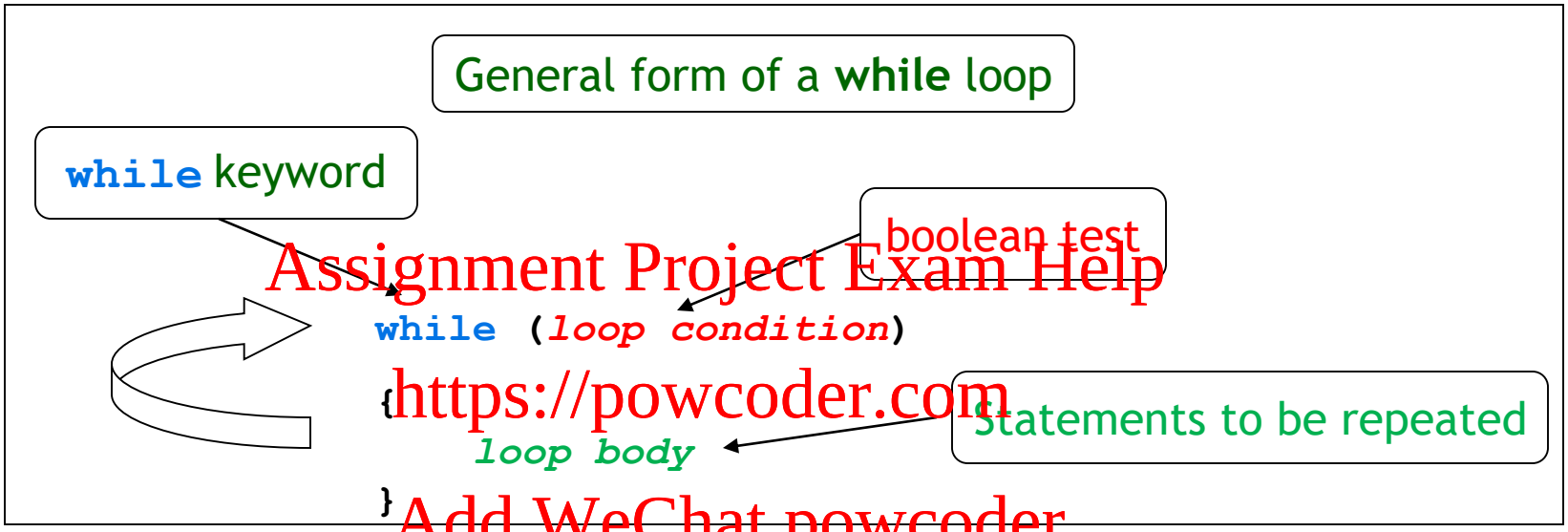


A *while* loop provides this control.



We typically use a *boolean* condition to decide whether or not to keep going with the *while* loop.

# Syntax : `while` loop






The above means : while the loop condition is *true*, perform the statements in the loop body.

# A Java while loop 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++;
    }
}
```

Condition to be tested

Increment *index* by 1

(What would happen if this step is omitted?)



Meaning : while the value of *index* is less than the size of the collection, get and print the next file name, and then increment *index by 1*

# Example : Searching a collection

```
int index = 0;
boolean found = false;
while(index < files.size() && !found)
{
     String file = files.get(index);
     if (file.contains(searchString))
          // item found, stop searching!
          found = true;
     else
        index++;
}
// When we finish the loop, either we have found the
// item at position index, or we have finished searching
// the whole collection and found no matching item.
```

# for-each versus while

- **for-each** loop:
  - easier to write. Works well with Java Collections.
  - safe : it is guaranteed to stop.
- **while** loop:
  - we don't *have* to process the whole collection.
    - doesn't even have to be used with a collection.
  - need to be careful : could be an *infinite loop* or try to process elements outside the collection.

# Example : while loop without using a collection

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

# An example: getting a menu choice

```
public int getMenuChoice()
{
    int choice = 0;
    Scanner input = new Scanner(System.in);
    while (choice < 1 || choice > 4)   // continues while choices are not valid
    {
        System.out.println("\n\nChoose one from the menu below");
        System.out.println("1  for  Yes");
        System.out.println("2  for  No");
        System.out.println("3  for  Maybe");
        System.out.println("4  for  Exit");
        choice = input.nextInt();          // accepts a user choice
        if (choice < 1 || choice > 4)   // rejects an invalid choice
            System.out.println("Choose a number between 1 and 4!!");
    }
    return choice;                         // returns a valid choice
}
```

# Review of 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 also provide special objects called *Iterators* that provide sequential access to a whole collection. *We will learn these later.*

# Choosing between a for-each and a while loop

*for-each* loop:

- use this if we want to process every element.



*while* loop:



- use this if we might want to stop part way through.



- use this for repetition that doesn't involve a collection.

# Another kind of loop: The traditional `for` loop

There are two variations of the for loop, *for-each* and a "normal" *for*.

The *for* loop is often used to iterate a fixed number of times – and not necessarily with a collection.

Often used with a variable that changes a fixed amount on each iteration.

*for* loops offer an alternative to while loops when the number of repetitions is known. For example with an array (more about this later) - a fixed sized collection.

*for* loops are used when an index variable is required.

# for loop syntax

General form of a for loop

```
for (initialization; condition; post-body action)
{
     statements to be repeated
}
```



The equivalent while-loop form



```
initialization;
while (condition)
{
     statements to be repeated
     post-body action
}
```

# A Java example : for versus while

for loop version

```
for (int number = 0; number < 100; number++)
{
     System.out.println(number);
}
```



while loop version



```
int number = 0;
while (number < 100)
{
     System.out.println(number);
     number++;
}
```

# Example : for loop with a bigger increment "step"

Eg :



```
// Print multiples of 3 that are below 40.

for (int number = 3; number < 40; number = number + 3)

{

        System.out.println(number);

}
```

# Pre and post-test loops

The `while` and `for-each` and `for` loops are *pre-test* loops. Pre-test loops test the condition at the start of the loop. This means that the loop may or may not execute.



A *post-test* loop tests the condition at the end of the loop, so the loop is always executed at least once.



A post-test loop is usually used when you know that the loop has to be executed at least once.



Any one of the loop constructs can be used in any repetition situation, but there are times when one is more sensible than the others.

In Java, the post-test loop is a `do ... while` loop.

# Example : Adding numbers keyed in

```
public int addNumbers()
{
    Scanner input = new Scanner(System.in);
    int total = 0;
    int number = 0;
    do
    {
        System.out.println("Enter a number (-99 to stop):");
        number = input.nextInt();
        if (number != -99)
            total = total + number;
    } while (number != -99);
    return total;
}
```

# An equivalent pre-test loop

```
public int addNumbers()
{
    Scanner input = new Scanner(System.in);
    int total = 0;
    int number = 0;
    while (number != -99)
    {
        System.out.println("Enter a number (-99 to stop):");
        number = input.nextInt();
        if (number != -99)
            total = total + number;
    }
    return total;
}
```

# Fixed-size collections

Sometimes the maximum collection size is known in advance.

Modern programming languages often offer a special **fixed-size** collection type: an *array*.



The elements in an array are held in a *fixed number of contiguous memory locations*.



Arrays behave in a similar way to ArrayLists, but use a different syntax.

# Arrays vs. ArrayLists

Access to items in arrays is often more efficient than access to items in comparable flexible-size collections.





Arrays can store objects OR primitive type values, whereas `ArrayLists` can store only objects.



Arrays have a special syntax (using `[ ]`) which is different from the usual Java expressions (including the `ArrayList`).

# Syntax : Creating an array object

**Supposing we want to create an attribute, of type array, to hold up to 5 integers :**



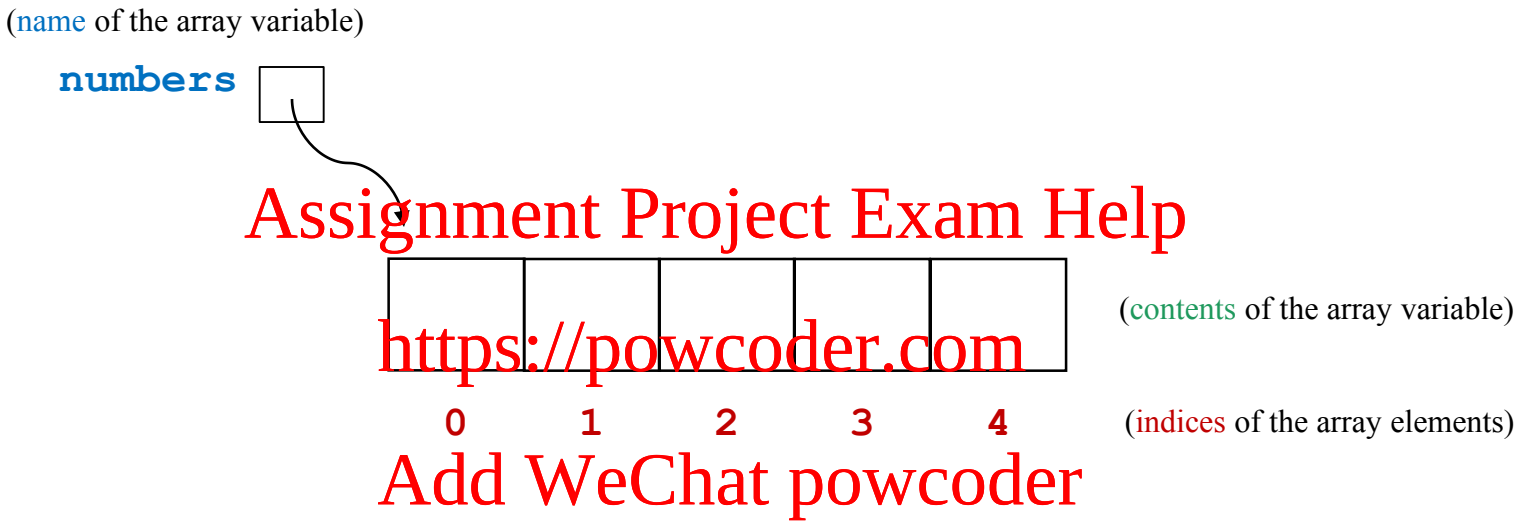
```
public class ArrayDemo
{
    private int[] numbers;
    ...;

    public ArrayDemo()
    {
       numbers = new int[5];
         ...
    }
    ...
}
```

Array object *declaration*

Array object *creation*

Which of these 2 statements do you think allocates the actual memory space for the array?

# The numbers array (logical view)






English explanation of the above:

`numbers` is an ***array of integers***, which can store a maximum of 5 ***elements***. Each individual element can be accessed using an index, which ranges from `0` to `4`. The indices must be integers.

# Syntax for using an array

A special "square-bracket" notation is used to access an array element. This applies for both assigning values to the array and retrieving values from the array.



Eg : `numbers[2]` means: "the third element in the array whose name is `numbers`"



(NB. the index *starts from 0*)

# Assigning values to an array

Array elements are accessed just like ordinary variables. Eg :

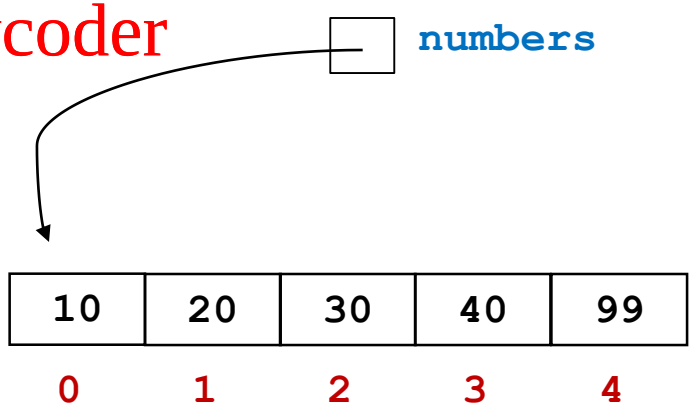
- to assign a value to each of the elements :



```
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 99;
```

note the use of the square brackets to indicate the use of an array

note the use of the indices to refer to the elements



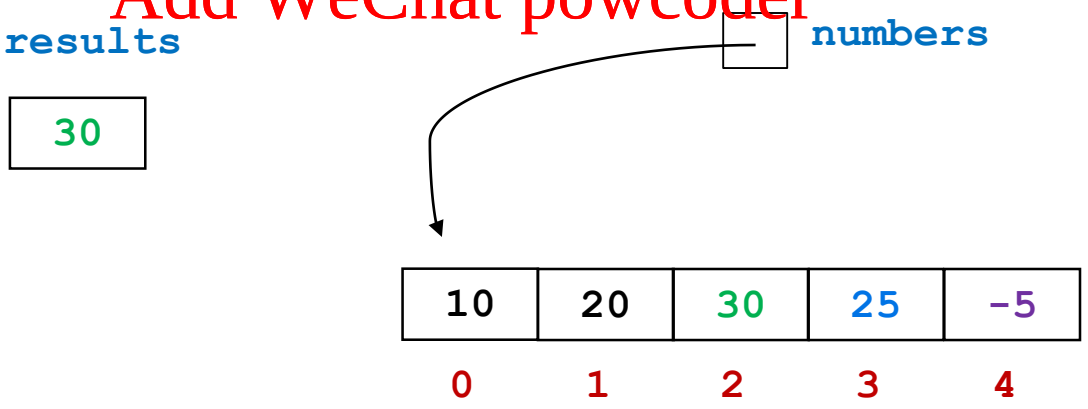
# Retrieving values from an array

Eg :

- to access the individual array elements:

```
numbers[3] = numbers[1] + 5;
numbers[4] = -5;
result = numbers[2];
```





# Arrays and Loops

**Loops are often used to manipulate arrays.**

**For instance, to initialize an array :**

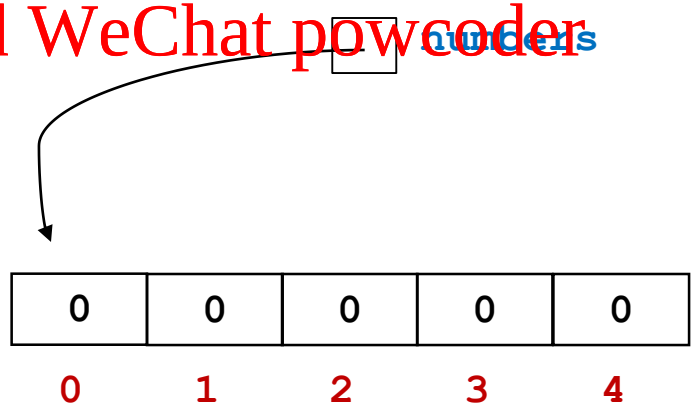
this is a special value which represents the **size** of an array



```
for (int index = 0; index < numbers.length; index++)
    numbers[index] = 0; // sets all elements to 0
```







# *More Array Example : The* weblog-analyzer *project*

A Web server records details of each access.

We can analyse this data to determine information such as:



- Most popular pages.
- Busiest periods.



- How much data is being delivered.



- Broken references.

The *weblog-analyzer* project (textbook Chapter04) counts accesses made in each one-hour period over 24 hours during the duration covered by the log.