University of Southampton

The Java Library

(slides adapted from D. Millard)

Thai Son Hoang

ECS, University of Southampton, U.K.

COMP1202
30th October 2023

Outline

Readings

- Chapter 4.12 and 4.14.7 of Barnes and Kölling [2016]

Encapsulation (1/3)

Take this example class where age is modelled as an **int**

```
public class Student {

    int age = 20;

    // code omitted

    public static void main(String[] args){
        Student s1 = new Student();
        System.out.println(s1.getAge());
    }

    public int getAge(){
        return age;
    }

}
```

Encapsulation (2/3)

```
public class Student {

    //int age = 20;
    Calendar dateOfBirth;

    //code omitted

    public static void main(String[] args){
        Student s1 = new Student();
        System.out.println(s1.getAge());
    }

    //public int getAge(){
    //  return age;
    //}

    public int getAge(){
        Calendar rightNow = Calendar.getInstance();
        int a = calculateAge(rightNow, dateofBirth);
        return a;
    }
}
```

Take this example class where age is modelled as an **int**

We might change the way that age is implemented – e.g. to make it based on the current date. Because we used an Accessor we do not need to alter main.

Encapsulation (3/3)

```
public class Student {

    //int age = 20;
    protected Calendar dateOfBirth;

    //code omitted

    public static void main(String[] args){
        Student s1 = new Student();
        System.out.println(s1.getAge());
    }

    //public int getAge(){
    //  return age;
    //}

    public int getAge(){
        Calendar rightNow = Calendar.getInstar
        int a = calculateAge(rightNow, dateofB
        return a;
    }
}
```

The **protected** keyword tells Java that only methods in this class* can access this variable.

*and its sub-classes, but we'll come to that later in the course...

And yes, **public** means the opposite – that all other methods can access it!

Constructors

```
public class Student {

    protected age;

    public Student() {
        age = 20;
    }

    public Student(int a) {
        age = a;
    }

    public static void main(String[] args){
        Student s1 = new Student(19);
        System.out.println(s1.getAge());
    }

    //code omitted
}
```

Constructor Rules:

- Must have the same name as the class
- Does not need a return type
- Can take parameters
- Can be overloaded
- Are invoked at the point of creation using the new keyword

Varieties of Loops

```
int i = 0;
while(i < 10){
        System.out.println(i);
        i++;
}

int i = 0;
do {
        System.out.println(i);
        i++;
} while (i < 10);

for ( int i = 0;      i < 10;      i++)
{
        System.out.println(i);
}
```

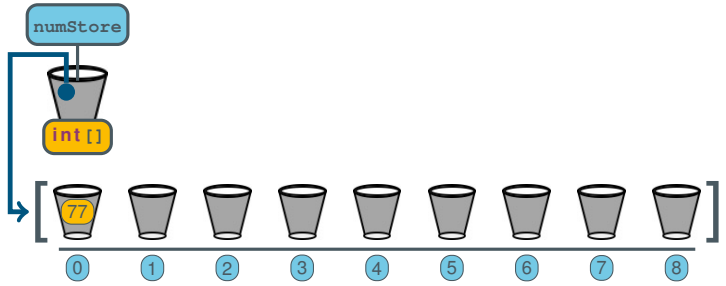
Initialisation | Condition | State Change

Condition is checked at start. Loops zero or more times.

Condition is checked at end. Loops one or more times.

A convenience loop for when we know it advance how many times we want to iterate. Loops zero or more times.

Arrays




```
int[] numStore;
numStore = new int[9];
numStore[0] = 77;
System.out.println(numStore[0]);
```

Declaration
Instantiation
Assignment
Retrieval

Iterating Over an Array

```
int numStore = new int[9];
//some missing code to fill the array with v

for (int i = 0; i < 9; i++){
    System.out.print("Number at position " + i);
    System.out.println(" is " + numStore[i]);
}
```

```
int numStore = new int[9];
    //some missing code to fill the array wi

    for (int n : numStore){
        System.out.println("Number is " + n)
    }
```

Iterating over an array is so common that Java now includes a loop specifically to do it.

Like the for loop the 'for each' loop is a shortcut, that is a bit neater than writing the code the long way.
But it can only be used for access (e.g. n++ would not increment the value in the array)
And it hides the current index

Arrays vs. ArrayLists

Arrays		ArrayLists
`Cat[] catArray;`	Declaration	`ArrayList catAList;`
`catArray = new Cat[10];`		`catAList = new ArrayList();`
`catArray[0] = moggy1;`	Insertion	`catAList.add(moggy1);`
`catArray[1] = moggy2;`		`catAList.add(moggy2);`
`callMethodOn(catArray[1]);`	Access	`callMethodOn(catAList.get(1));`
`catArray[0] = null;`	Removal	`catAList.remove(moggy1);`

Generics (1/2)

```
ArrayList kennel = new ArrayList();

kennel.add(new Dog("Rover"));
kennel.add(new Dog("Fido"));
kennel.add(new Dog("Patch"));
kennel.add(new Cat("Mr Tiddles"));


for(int i = 0; i < kennel.size(); i++)
    kennel.get(i).bark();
}
```

ArrayLists store objects of any type

Which means we can mix up the types of objects in the ArrayList

Which may cause problems later if we make assumptions about what is in there!

In fact this code will not compile, because Java does not know what is in the ArrayList, and therefore will not let you call **bark** on it

Generics (2/2)

```
ArrayList<Dog> kennel = new ArrayList<Dog>();

kennel.add(new Dog("Rover"));
kennel.add(new Dog("Fido"));
kennel.add(new Dog("Patch"));
kennel.add(new Cat("Mr Tiddles"));


for(int i = 0; i < kennel.size(); i++) {
    kennel.get(i).bark();
}
```

It would be better if we could ensure that the ArrayList **only** contained Dogs in the first place

This is easily done because ArrayList uses a mechanism called **generics**.
We can specify the type allowed when we create the ArrayList.

Now Java will only allow us to add things of type **Dog**. So this line will force a **compile time** error

Iterator (1/2)

University of Southampton

```
ArrayList<Dog> kennel = new ArrayList<Dog

kennel.add(new Dog("Rover"));
kennel.add(new Dog("Fido"));
kennel.add(new Dog("Patch"));


for(int i = 0; i < kennel.size(); i++) {
    kennel.get(i).bark();
}

Iterator<Dog> it = kennel.iterator();

while(it.hasNext()) {
    it.next().bark();
}
```

1) They are neater, and neat code is easier to read and understand

2) They decouple the loop from the collection (notice that in the loop we do not reference the ArrayList at all)
This means we could pass the iterator to a method – and that method does not even need to know what the collection is!

Iterators (2/2)

University of Southampton

```
public void makeThemBark( Iterator<Dog> it) {

    while (it.hasNext()) {
        it.next().bark();
    }

}
```

1) They are neater, and neat code is easier to read and understand

2) They decouple the loop from the collection (notice that in the loop we do not reference the Arraylist at all)
This means we could pass the iterator to a method – and that method does not even need to know what the collection is!

Library

University of Southampton

- The java library is full of helpful classes
- Like ArrayList
  - What does the inside of an ArrayList look like?
  - How does it handle the resizing?
  - How does it know when the throw an error?
  - How does it handle renumbering when removing elements?
  - But they are implementation details.

Implementation vs. Interface

- Because of encapsulation all we need to know to use the ArrayList class, and the other library classes is what their interface is
- A Class' interface is how we interact with the class
  - It's public variables and methods
  - what methods we can call
  - what they do
  - what they will return

Importing

- Library classes must be imported using an **import** statement

```
import java.util.ArrayList;

public class myClass{

    ArrayList<String> arrl;
    arrl = new ArrayList<String>;

    public static void main(String[] args){
        //code omitted
    }
}
```


Importing packages

- Classes are organised in packages.
- Single class may be imported:

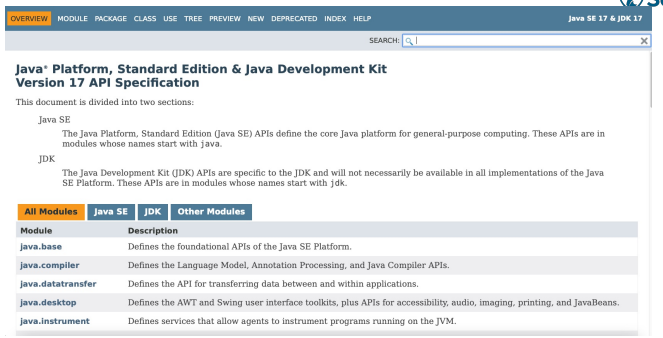
```
import java.util.ArrayList;
```

- Whole package can be imported:

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


Where is the Library?

- All library classes are included in the Java runtime and development environments
- All the documentation is available online:
  - https://docs.oracle.com/en/java/javase/17/docs/api/index.html



OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP — Java SE 17 & JDK 17

SEARCH:

Java® Platform, Standard Edition & Java Development Kit
Version 17 API Specification

This document is divided into two sections:

Java SE

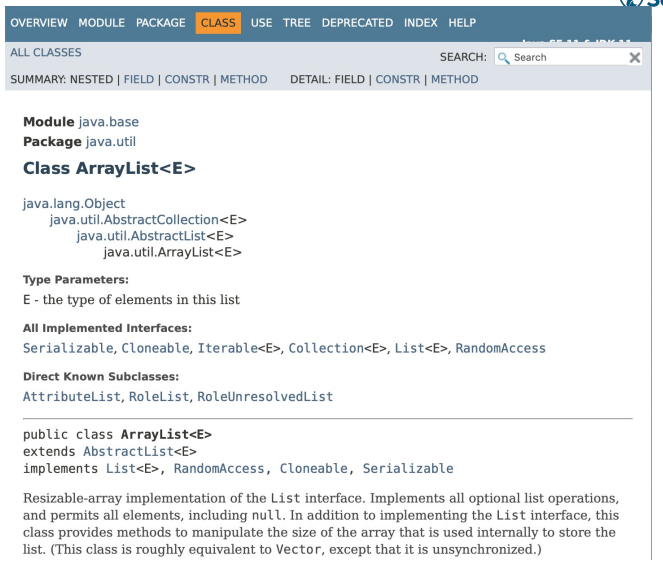
The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

All Modules | Java SE | JDK | Other Modules

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

ALL CLASSES — SEARCH: Search

SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.util

Class ArrayList<E>

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

Type Parameters:

E - the type of elements in this list

All Implemented Interfaces:

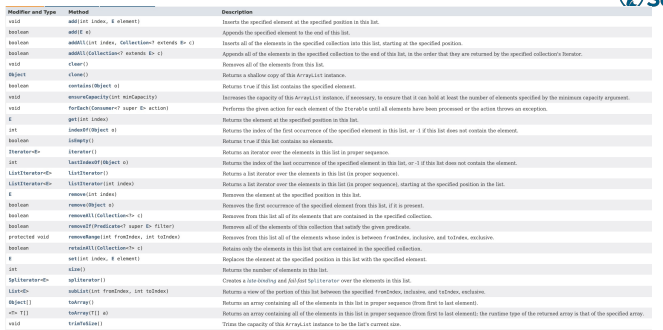
`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess`

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.)



Modifier and Type	Method	Description
void	add(int index, E element)	Inserts the specified element at the specified position in this list.
boolean	add(E e)	Appends the specified element to the end of this list.
boolean	addAll(int index, Collection<? extends E> c)	Inserts all of the elements in the specified collection into this list, starting at the specified position.
boolean	addAll(Collection<? extends E> c)	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
void	clear()	Removes all of the elements from this list.
Object	clone()	Returns a shallow copy of this ArrayList instance.
boolean	contains(Object o)	Returns true if this list contains the specified element.
void	ensureCapacity(int minCapacity)	Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
void	forEach(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
E	get(int index)	Returns the element at the specified position in this list.
int	indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty()	Returns true if this list contains no elements.
Iterator<E>	iterator()	Returns an iterator over the elements in this list in proper sequence.
int	lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	listIterator()	Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	listIterator(int index)	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	remove(int index)	Removes the element at the specified position in this list.
boolean	remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present.
boolean	removeAll(Collection<?> c)	Removes from this list all of its elements that are contained in the specified collection.
boolean	removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
protected void	removeRange(int fromIndex, int toIndex)	Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
boolean	retainAll(Collection<?> c)	Retains only the elements in this list that are contained in the specified collection.
E	set(int index, E element)	Replaces the element at the specified position in this list with the specified element.
int	size()	Returns the number of elements in this list.
Spliterator<E>	spliterator()	Creates a late-binding and fail-fast Spliterator over the elements in this list.
List<E>	subList(int fromIndex, int toIndex)	Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	toArray()	Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	toArray(T[] a)	Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
void	trimToSize()	Trims the capacity of this ArrayList instance to be the list's current size.

Outline

Strings

- Strings are actually objects
  - Did you notice we always use a capital S like other classes?
- You don't need to import them
  - they are from the automatically imported from `java.lang.*`;
- As is `System` as in `System.out.println()`

A word about comparing Strings

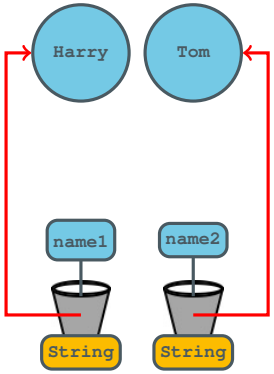
```
if (input == "hello") { // Tests identity
    //code here
}

if (input.equals("hello")) { // Tests Equality
    //code here
}
```

- You probably mean to compare strings using `.equals`.
- And you should always use `.equals`.

identity vs equality (1/4)

```
String name1 = "Harry";
String name2 = "Tom";
```

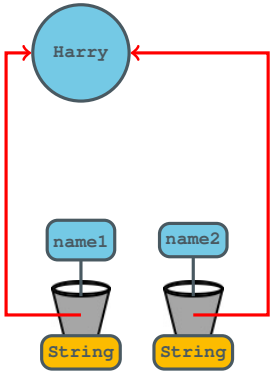
- `name1 == name2`: false
  (different addresses)
- `name1.equals(name2)`: false
  (different values)



identity vs equality (2/4)

```
String name1 = "Harry";
String name2 = "Harry";
```

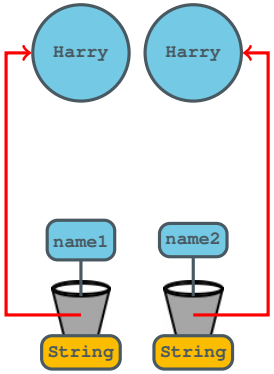
- `name1 == name2`: true
  (same address)
- `name1.equals(name2)`: true
  (same value)



identity vs equality (3/4)

```
String name1 = "Harry";
String name2 = new
    String("Harry");
```

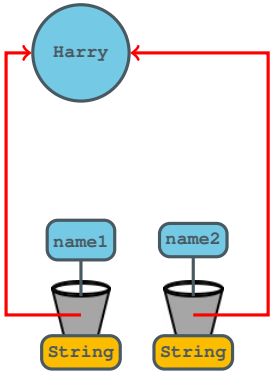
- `name1 == name2`: false (different addresses)
- `name1.equals(name2)`: true (same value)



identity vs equality (4/4)

```
String name1 = "Harry";
String name2 = name1;
```

- `name1 == name2`: true (same address)
- `name1.equals(name2)`: true (same value)



Outline

Maps

Name	Number
Alfie	407351
Jenny	763412
...	...

- Maps are a collection type that map a key to a value
- `put("Alfie", "407351");`
- `put("Jenny", "763412");`
- and so on ...

Lookup

- Lookup: supplying the key and having the value returned

```
String num = myHashMap.get("Alfie");
```

Name	Number
Alfie	407351
Jenny	763412
...	...

Bringing it together . . .

```
import java.util.HashMap;

//code omitted

HashMap<String, Integer> marks;
marks = new HashMap<String, Integer>();

marks.put("Alice", 75);
marks.put("Bob", 62);
marks.put("Colin", 68);

System.out.println("Bob got " +
    marks.get("Bob"));
```

What is this?

HashMap is a **generic class**, this means we should tell it what two types it maps together

What is happening here?

This is **autoboxing** – the ints are automatically turned into Integers for us

Which type of String comparison is being used?

Equality (not Identity). It is using the **.equals** method

Summary

- Implementation vs. Interface
- Strings
- HashMaps

Self-Tests

Blackboard tests on Strings

Blackboard tests on HashMaps

YOUR QUESTIONS

References I

- David J. Barnes and Michael Kölling. *Objects First with Java: A Practical Introduction using BlueJ*.
  Pearson, sixth edition edition, 2016 (Chapter 4.12 and 4.14.7)