

Section 6

Collections & Using the Java Library

Learning Outcomes

- After this lecture you should be able to:
 - Give an overview of the Java Collection Framework
 - Describe the purpose of an interface
 - Outline the purpose of type-wrapper classes
 - Describe Autoboxing and Auto-unboxing
 - Outline the relationship between *List*, *ArrayList* & *LinkedList*
 - Implement instances of classes which uses the *List* interface
 - Describe and implement the *enhanced for loop*
 - Describe and implement a *ListIterator*
 - Distinguish between short circuit and non short circuit Boolean operators
 - **Reading and studying** recommended text is essential to improve your understanding of the above

Collections

- The **java.util** standard package contains different types of classes (and interfaces) for maintaining a collection of objects (collectively referred to as the *Java Collection Framework (JCF)*).
- JCF includes classes that maintain collections of objects as sets, lists, or maps.
- An *interface* is a reference data type, but unlike a class, an interface includes constants and *abstract* methods.
 - All methods of an Interface do not contain implementation (method bodies) as of all versions below Java 8
 - An *abstract* method has only a method header (method prototype) and no method body
 - The abstract methods of an interface define a behaviour.



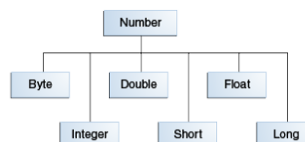
If only I could find an array that could shrink when you remove something. And one that you didn't have to loop through to check each element, but instead you could just ask it if it contains what you're looking for. And it would let you get things out of it, without having to know exactly which slot the things are in. That would be dreamy. But I know it's just a fantasy...

Collections

- Java API provides several predefined data structures, called **collections**, used to store groups of related objects.
 - Each provides efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
 - Reduce application-development time.
- **ArrayList<T>** (package java.util) can dynamically change its size to accommodate more elements.
 - T is a *placeholder* for the type of element stored in the collection.
 - This is similar to specifying the type when declaring an array, except that only non-primitive types can be used with these collection classes.
- Classes with this kind of placeholder that can be used with any type are called **generic classes**.

Type-Wrapper Classes

- Each primitive type has a corresponding **type-wrapper class** (in package java.lang).
 - **Boolean, Byte, Character, Double, Float, Integer, Long** and **Short**.
- These classes "wrap" the primitive in an object.
 - Each type-wrapper class enables you to manipulate primitive-type values as objects.
 - All of the numeric wrapper classes are subclasses of the abstract class **Number**:



- Collections cannot manipulate variables of primitive types.
 - They can manipulate objects of the type-wrapper classes, because every class ultimately derives from Object.

Autoboxing

- *Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.

```
Integer[] integerArray = new Integer[3];

for (int i=0; i < integerArray.length; i++)
    integerArray[i] = i; // Autoboxing

for (int i=0; i < integerArray.length; i++)
    integerArray[i] = Integer.valueOf(i); // Wrapper
```

Autoboxing

- The Java compiler applies *autoboxing* when a primitive value is:
 - Passed as a parameter to a method that expects an object of the corresponding wrapper class.
 - Assigned to a variable of the corresponding wrapper class.

```
List<Integer> myList = new ArrayList<>();

for (int i = 1; i < 10; i += 2)
    myList.add(i); // Autoboxing

for (int i = 1; i < 10; i += 2)
    myList.add(Integer.valueOf(i)); // Wrapper
```

Auto-Unboxing

- Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called *unboxing*.
- Compiler invokes the *intValue* method to convert an Integer to an *int* at runtime.

```
Integer[] integerArray = new Integer[3];  
  
integerArray[0] = 4;  
  
int y = integerArray[0]; // Auto-Unboxing  
  
int y = integerArray[0].intValue();
```

Auto-Unboxing

- The Java compiler applies unboxing when an object of a wrapper class is:
 - Passed as a parameter to a method that expects a value of the corresponding primitive type.
 - Assigned to a variable of the corresponding primitive type.

```
Integer x = 3;  
takeNumber(x);  
  
void takeNumber (int y) { } // Auto-Unboxing  
  
int y = x; // Auto-Unboxing
```

Java Interface

- A Java **interface** defines only the behavior of objects
 - It includes only public methods with no method bodies.
 - It does not include any data members except public constants
 - No instances of a Java interface can be created
 - Need an instance of a class that implements the interface

```
List myList = new List(); // compiler error

//Add members to a list
List<Person> friends = new ArrayList<Person>( );

//Add members to a list
List<Person> friends = new LinkedList<Person>( );
```

Java Interfaces (Cont.)

- **Interfaces** offer a capability requiring that unrelated classes implement a set of common methods.
- Interfaces define and standardise the ways in which things such as people and systems can interact with one another.
- The interface specifies *what* operations to perform but does not specify *how* the operations are performed.
- A Java interface describes a set of methods that can be called on an object.

Java Interfaces (Cont.)

- An **interface declaration** begins with the keyword **interface** and contains only constants and abstract methods.
 - All interface members must be public.
 - Interfaces may not specify any implementation details, such as concrete method declarations and instance variables.
 - All methods declared in an interface are implicitly public abstract methods.
 - All fields are implicitly public, static and final.

List Interface

- An abbreviated *List* interface (that can contain elements of any type E) is defined as follows:

```
public interface List<E> extends Collection<E> {  
  
    int size();  
    boolean isEmpty();  
    boolean contains(Object o);  
    boolean add(E e);  
    boolean remove(Object o);  
    E get(int index);  
    void clear();  
    ...  
}
```

Lists

- A List (sometimes called a **sequence**) is a Collection that can contain duplicate elements.
- In addition to the methods inherited from Collection, List provides methods for manipulating elements via their indices (List indices are zero based), manipulating a specified range of elements, searching for elements and obtaining a **ListIterator** to access the elements.
- Interface List is implemented by several classes, including **ArrayList**, **LinkedList** and **Vector**.
- Autoboxing occurs when you add primitive-type values to objects of these classes, because they store only references to objects.

Lists (cont.)

- Class *ArrayList* and *Vector* are resizable-array **implementations** of List.
- Inserting an element between existing elements of an *ArrayList* or *Vector* is an inefficient operation.
- A *LinkedList* enables efficient insertion (or removal) of elements in the middle of a collection.
- The primary difference between *ArrayList* and *Vector* is that *Vectors* are synchronized by default, whereas *ArrayLists* are not.
- Unsynchronized collections provide better performance than synchronized ones.
- For this reason, *ArrayList* is typically preferred over *Vector* in programs that do not share a collection among threads.

Lists (cont.)

- The **List** interface that supports methods to maintain a collection of objects as a linear list

$L = (l_0, l_1, l_2, \dots, l_N)$

- We can add to, remove from, and retrieve objects in a given list.
- A list does not have a set limit to the number of objects we can add to it.

```
import java.util.List;  
import java.util.ArrayList;  
import java.util.LinkedList;
```

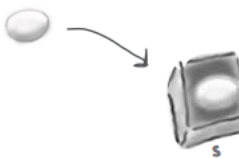
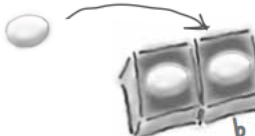
List Methods

- Here are five of the 25 list methods:


<code>boolean add (Object o)</code>
Adds an object o to the list
<code>void clear ()</code>
Clears this list, i.e., make the list empty
<code>Object get (int idx)</code>
Returns the element at position idx
<code>boolean remove (int idx)</code>
Removes the element at position idx
<code>int size ()</code>
Returns the number of elements in the list

ArrayList

Some things you can do with ArrayList

- ① Make one
`ArrayList<Egg> myList = new ArrayList<Egg>();`
Don't worry about this new <Egg> angle-bracket syntax right now; it just means "make this a list of Egg objects".
A new ArrayList object is created on the heap. It's little because it's empty.
- ② Put something in it
`Egg s = new Egg();`
`myList.add(s);`

Now the ArrayList grows a "box" to hold the Egg object.
- ③ Put another thing in it
`Egg b = new Egg();`
`myList.add(b);`

The ArrayList grows again to hold the second Egg object.

ArrayList

- ④ Find out how many things are in it
`int theSize = myList.size();`
The ArrayList is holding 2 objects so the size() method returns 2
- ⑤ Find out if it contains something
`boolean isIn = myList.contains(s);`
The ArrayList DOES contain the Egg object referenced by 's', so contains() returns true
- ⑥ Find out where something is (i.e. its index)
`int idx = myList.indexOf(b);`
ArrayList is zero-based (means first index is 0) and since the object referenced by 'b' was the second thing in the list, indexOf() returns 1
- ⑦ Find out if it's empty
`boolean empty = myList.isEmpty();`
it's definitely NOT empty, so isEmpty() returns false
- ⑧ Remove something from it
`myList.remove(s);`

Hey look — it shrank!

Class Object

- Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class i.e. equals(), toString()

```
public String toString()
```

- Returns a string representation of the object. In general, the toString method returns a string that "textually represents" this object
 - getClass().getName() + '@' + Integer.toHexString(hashCode())
- The result should be a concise but informative representation that is easy for a person to read.
- It is recommended that *all* subclasses **override** this method.

Person Class

```
class Person {  
    // Data Members  
    private String name; // The name of this person  
    private int age; // The age of this person  
    private char gender; // The gender of this person  
  
    // Default constructor  
    public Person() {  
        this("Not Given", 0, 'U');  
    }  
  
    // Constructs a new Person with passed name, age, and gender.  
    // Illustrates explicit use of this  
    public Person(String name, int age, char gender) {  
        this.age = age;  
        this.name = name;  
        this.gender = gender;  
    }  
}
```

Person Class (cont.)

```
public int getAge( ) {
    return age;
}
public char getGender( ) {
    return gender;
}
public String getName( ) {
    return name;
}
public void setAge( int age ) {
    this.age = age;
}
public void setGender( char gender ) {
    this.gender = gender;
}
public void setName( String name ) {
    this.name = name;
}
// OVERRIDE toString()
public String toString() {
    return getName() + " " + getAge() + " " + getGender();
}

} // end class
```

Using the List Interface

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

class FriendsList {
    public static void main(String[] arg) {

        List<Person> friends = new ArrayList<>( );
        // List<Person> friends = new LinkedList<>( );

        friends.add(new Person("jane", 10, 'F'));
        friends.add(new Person("jack", 6, 'M'));
        friends.add(new Person("jill", 8, 'F'));

        for (int i=0; i < friends.size(); i++){
            System.out.println(friends.get(i).getName() + " " +
            friends.get(i).getAge() + " " + friends.get(i).getGender());
        }

        friends.remove(1); //Remove the second person

        for (Person p : friends) {
            System.out.println(p.toString()); // Overridden toString()
        }
    }
}
```

Enhanced for Statement

- Iterates through the elements of an array without using a counter.
- Avoids the possibility of “stepping outside” the array.
- Also works with the Java API’s prebuilt collections

Syntax:

```
for ( parameter : arrayName )  
    statement
```

- where *parameter* has a type and an identifier and *arrayName* is the array through which to iterate.
- Parameter type must be consistent with the array’s element type.
- The enhanced for statement simplifies the code for iterating through an array.

Enhanced for Statement (cont.)

- The enhanced for statement can be used only to obtain array elements
 - It cannot be used to modify elements.
 - To modify elements, use the traditional counter-controlled for statement.

Read this loop declaration as “repeat for each element in the `friends` ArrayList ; take the next element in the ArrayList and assign it to the Person variable `p`”.

The colon (:) means “in” so the whole thing means “for each Person value ***IN*** `friends`”.

```
for (Person p : friends)
```

Declare a variable that will hold one element from the ArrayList. With each iteration, this variable will hold a different element from the ArrayList until there are no more elements left.

The ArrayList to iterate over in the loop. With each iteration, the next element in the ArrayList will be assigned to the Variable “p”.

Iterator

- To cycle through the elements in a collection.
- You can use an iterator, which is an object that implements either the *Iterator* or the *ListIterator* interface.
 - Iterator enables you to cycle through a collection, obtaining or removing elements.
 - ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
- Before you can access a collection through an iterator, you must obtain one.
 - Each of the collection classes provides an `iterator()` method that returns an iterator to the start of the collection.
 - By using this iterator object, you can access each element in the collection, one element at a time.

Iterator (cont.)

- In general, to use an iterator to cycle through the contents of a collection, follow these steps:
 - Obtain an iterator to the start of the collection by calling the collection's `iterator()` method.
 - Set up a loop that makes a call to `hasNext()`. Have the loop iterate as long as `hasNext()` returns true.
 - Within the loop, obtain each element by calling `next()`.
- For collections that implement `List`, you can also obtain an iterator by calling `ListIterator`.

ListIterator

- An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
- Interface ListIterator **extends** Iterator
- ListIterator traverses Lists only

```
import java.util.ListIterator;

//Traverse the list with a ListIterator
ListIterator<Person> itr = friends.listIterator( );
while ( itr.hasNext( ) ) {
    System.out.println(p.getName() + " " + p.getAge() + " " +
        p.getGender());
    //System.out.println(itr.next( )); //if toString() is Overridden
}
```

ArrayList VS Array

① **A plain old array has to know its size at the time it's created.**

But for ArrayList, you just make an object of type ArrayList. Every time. It never needs to know how big it should be, because it grows and shrinks as objects are added or removed.

`new String[2]` Needs a size.

`new ArrayList<String>()`

No size required (although you can give it a size if you want to).

② **To put an object in a regular array, you must assign it to a specific location.**

(An index from 0 to one less than the length of the array.)

`myList[1] = b;`

Needs an index.

If that index is outside the boundaries of the array (like, the array was declared with a size of 2, and now you're trying to assign something to index 3), it blows up at runtime.

With ArrayList, you can specify an index using the `add(anInt, anObject)` method, or you can just keep saying `add(anObject)` and the ArrayList will keep growing to make room for the new thing.

`myList.add(b);`

No index.

ArrayList VS Array

③ Arrays use array syntax that's not used anywhere else in Java.

But ArrayLists are plain old Java objects, so they have no special syntax.

```
myList[1]
```

The array brackets `[]` are special syntax used only for arrays.

④ ArrayLists in Java 5.0 are parameterized.

We just said that unlike arrays, ArrayLists have no special syntax. But they *do* use something special that was added to Java 5.0 Tiger—*parameterized types*.

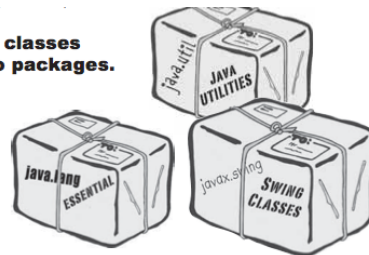
```
ArrayList<String>
```

The `<String>` in angle brackets is a "type parameter". `ArrayList<String>` means simply "a list of Strings", as opposed to `ArrayList<Dog>` which means, "a list of Dogs".

Using the Java API

- Every class in the Java library belongs to a package. ArrayList is in a package called java.util.
- To use a class from the API in your code, you treat it as though you wrote and compiled it
- You have to indicate in your code the FULL name of the library class you want to use i.e. `import java.util.List;`
- You have already used classes from a package. System, String and Math all belong to the java.lang package.

In the Java API, classes are grouped into packages.



To use a class in the API, you have to know which package the class is in.

`java.util.ArrayList`
package name class name

Boolean Operators Revisited

- AND (&&), OR (||), Exclusive OR (^), Not (!)
 - && returns *true* if both conditions are true.
 - || returns *true* if either condition is true.
 - ^ returns *true* if and *ONLY* if, one of its operands are true and the other is false.
 - ! Reverses the meaning of a condition.
- Use Parenthesis rather than learning precedence
- Short Circuit Operators (&&, ||)
 - &&, if first condition is false JVM stops evaluation.
 - ||, if first condition is true, JVM stops evaluation and returns true.
- Non Short Circuit Operators (&, |)
 - Force JVM to check both sides of the expression.