13

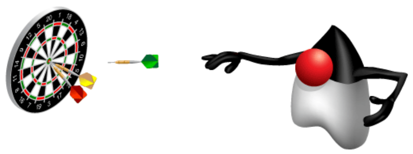
Using Interfaces

Interactive Quizzes

Objectives

After completing this lesson, you should be able to:

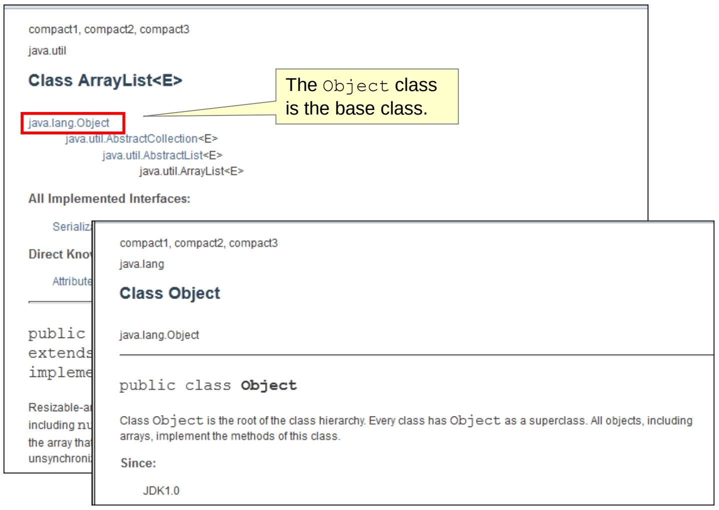
- Override the `toString` method of the `Object` class
- Implement an interface in a class
- Cast to an interface reference to allow access to an object method
- Write a simple lambda expression that consumes a `Predicate`



Topics

- Polymorphism in the JDK foundation classes
- Using interfaces
- Using the `List` interface
- Introducing lambda expressions

The `Object` Class



compact1, compact2, compact3

java.util

Class ArrayList<E>

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

All Implemented Interfaces:

Serializ

Direct Kno

Attribute

```
public
extends
impleme
```

Resizable-a
including nu
the array tha
unsynchroni

compact1, compact2, compact3

java.lang

Class Object

java.lang.Object

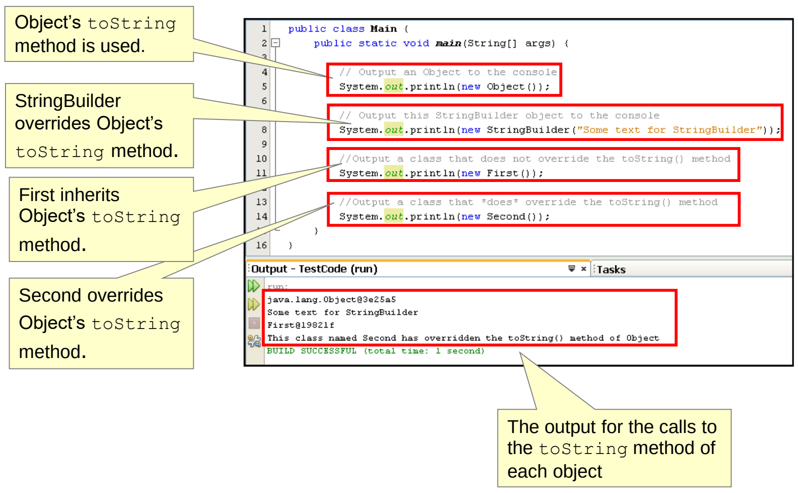
```
public class Object
```

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.

Since:

JDK1.0

Calling the `toString` Method



```java
public class Main {
    public static void main(String[] args) {

        // Output an Object to the console
        System.out.println(new Object());

        // Output this StringBuilder object to the console
        System.out.println(new StringBuilder("Some text for StringBuilder"));

        //Output a class that does not override the toString() method
        System.out.println(new First());

        //Output a class that *does* override the toString() method
        System.out.println(new Second());
    }
}
```

Output - TestCode (run)

```
run:
java.lang.Object@3e25a5
Some text for StringBuilder
First@19821f
This class named Second has overridden the toString() method of Object
BUILD SUCCESSFUL (total time: 1 second)
```

Object's toString method is used.
StringBuilder overrides Object's toString method.
First inherits Object's toString method.
Second overrides Object's toString method.
The output for the calls to the toString method of each object
Tasks

Overriding `toString` in Your Classes

Shirt class example

```
public String toString(){
  return "This shirt is a " + desc + ";"
     + " price: " + getPrice() + ","
     + " color: " + getColor(getColorCode());
}
```

Output of `System.out.println(shirt)`:

- Without overriding `toString`
  ```
  examples.Shirt@73d16e93
  ```
- After overriding `toString` as shown above
  ```
  This shirt is a T Shirt; price: 29.99, color: Green
  ```

Topics

- Polymorphism in the JDK foundation classes
- Using interfaces
- Using the `List` interface
- Introducing lambda expressions

The Multiple Inheritance Dilemma

Can I inherit from *two* different classes? I want to use methods from both classes.

- Class Red:

  `public void print()  {System.out.print("I am Red");}`

- Class Blue:

  `public void print() {System.out.print("I am Blue");}`

```
public class Purple extends Red, Blue{
    public void printStuff() {
        print();   }
}
```

Which implementation of `print()` will occur?

The Java Interface

- An interface is similar to an abstract class, except that:
  - Methods are implicitly abstract (except default methods)
  - A class does not *extend* it, but *implements* it
  - A class may implement more than one interface
- All abstract methods from the interface must be implemented by the class.

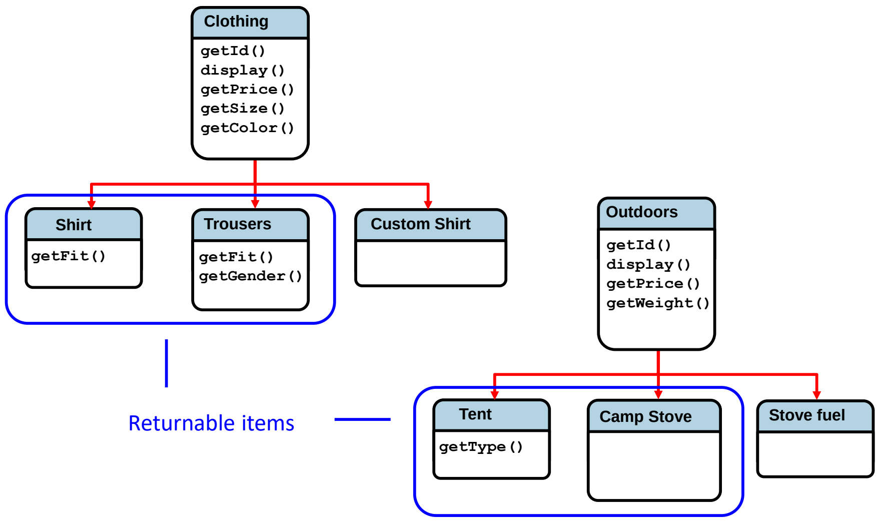
```
public interface Printable {
    public void print();
}
```

Implicitly abstract

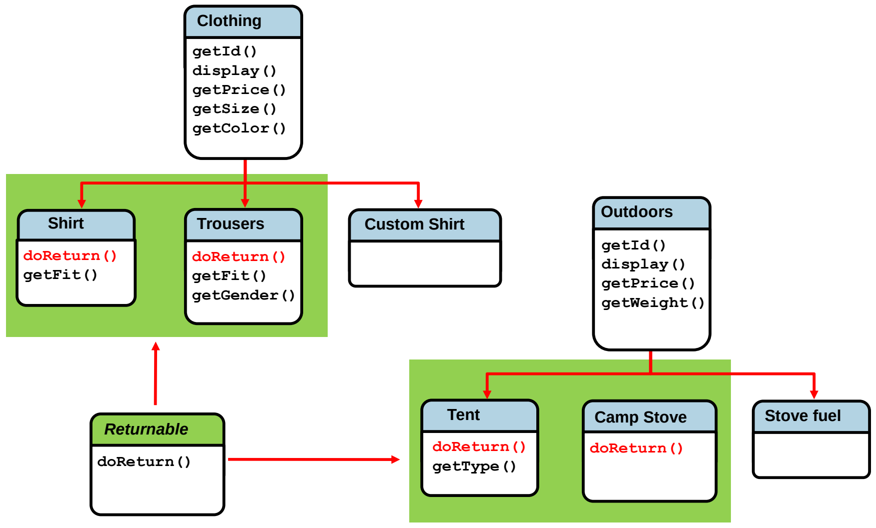
```
public class Shirt implements Printable {
    ...
    public void print(){
        System.out.println("Shirt description");
    }
}
```

Implements the `print()` method.

Multiple Hierarchies with Overlapping Requirements



Using Interfaces in Your Application



Implementing the `Returnable` Interface

`Returnable` interface

```
public interface Returnable {
  public String doReturn();
}
```

Implicitly abstract method

`Shirt` class

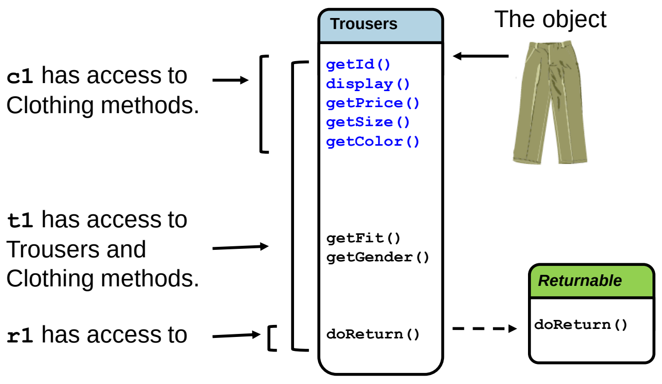
Now, Shirt **'is a'** Returnable.

```
public class Shirt extends Clothing implements Returnable {
 public Shirt(int itemID, String description, char colorCode,
              double price, char fit) {
    super(itemID, description, colorCode, price);
    this.fit = fit;
  }
public String doReturn() {
  // See notes below
   return "Suit returns must be within 3 days";
}
...< other methods not shown > ...        } // end of class
```

Shirt implements the method declared in Returnable.

Access to Object Methods from Interface

```
Clothing c1 = new Trousers();
Trousers t1 = new Trousers();
Returnable r1 = new Trousers();
```



Casting an Interface Reference

```
Clothing c1 = new Trousers();
Trousers t1 = new Trousers();
Returnable r1 = new Trousers();
```

- The `Returnable` interface does not know about `Trousers` methods:

```
r1.getFit()                    //Not allowed
```

- Use **casting** to access methods defined outside the interface.

```
((Trousers)r1).getFit();
```

- Use `instanceof` to avoid inappropriate casts.

```
if(r1 instanceof Trousers) {
       ((Trousers)r1).getFit();
}
```

Quiz

Which methods of an object can be accessed via an interface that it implements?

a. All the methods implemented in the object's class
b. All the methods implemented in the object's superclass
c. The methods declared in the interface

Quiz

How can you change the reference type of an object?

a. By calling `getReference`
b. By casting
c. By declaring a new reference and assigning the object

Topics

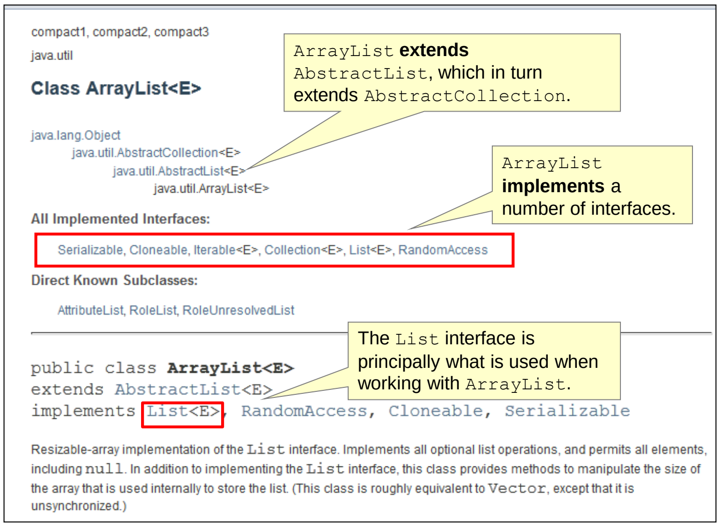
- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using the `List` interface
- Introducing lambda expressions

The Collections Framework

The collections framework is located in the `java.util` package. The framework is helpful when working with lists or collections of objects. It contains:

- Interfaces
- Abstract classes
- Concrete classes (Example: `ArrayList`)

`ArrayList` Example



compact1, compact2, compact3
java.util

Class ArrayList<E>

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

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
```

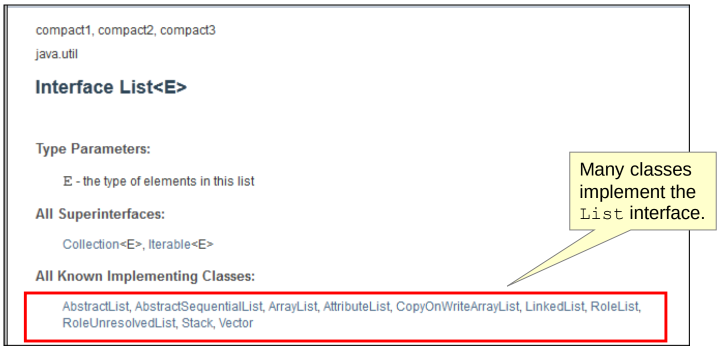
Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

`ArrayList` **extends** `AbstractList`, which in turn extends `AbstractCollection`.

`ArrayList` **implements** a number of interfaces.

The `List` interface is principally what is used when working with `ArrayList`.

`List` Interface



compact1, compact2, compact3

java.util

Interface List<E>

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

All of these object types can be assigned to a `List` variable:

```
ArrayList<String> words = new ArrayList();
List<String> mylist = words;
```

Example: `Arrays.asList`

The `java.util.Arrays` class has many static utility methods that are helpful in working with arrays.

- Converting an array to a `List`:

```
String[] nums = {"one","two","three"};
List<String> myList = Arrays.asList(nums);
```

List objects can be of many different types. What if you need to invoke a method belonging to `ArrayList`?

```
mylist.replaceAll()
```

This works! `replaceAll` comes from List.

```
mylist.removeIf()
```

Error! `removeIf` comes from Collection (superclass of ArrayList).

Example: `Arrays.asList`

Converting an array to an `ArrayList`:

```
String[] nums = {"one","two","three"};
List<String> myList = Arrays.asList(nums);
ArrayList<String> myArrayList = new ArrayList(myList);
```

Shortcut:

```
String[] nums = {"one","two","three"};
ArrayList<String> myArrayList =
     new ArrayList( Arrays.asList(nums));
```

Exercise 13-1: Converting an Array to an ArrayList

In this exercise, you:

- Convert a `String` array to an `ArrayList`
- Work with the `ArrayList` reference to manipulate list values

Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using the `List` interface
- Introducing lambda expressions

New SE 8 Feature!

Example: Modifying a `List` of Names

Suppose you want to modify a `List` of names, changing them all to uppercase. Does this code change the elements of the `List`?

```
String[] names = {"Ned","Fred","Jessie","Alice","Rick"};
List<String> mylist = new ArrayList(Arrays.asList(names));

// Display all names in upper case
for(String s: mylist){
   System.out.print(s.toUpperCase()+", ");
}
System.out.println("After for loop: " + mylist);
```

Returns a new String to print

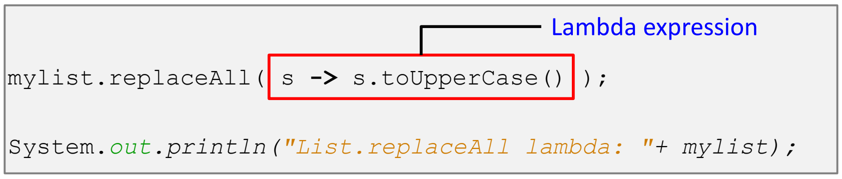
Output:

```
NED, FRED, JESSIE, ALICE, RICK,
After for loop: [Ned, Fred, Jessie, Alice, Rick]
```

The list elements are unchanged.

Using a Lambda Expression with `replaceAll`

`replaceAll` is a default method of the `List` interface. It takes a lambda expression as an argument.



```
mylist.replaceAll( s -> s.toUpperCase() );

System.out.println("List.replaceAll lambda: "+ mylist);
```

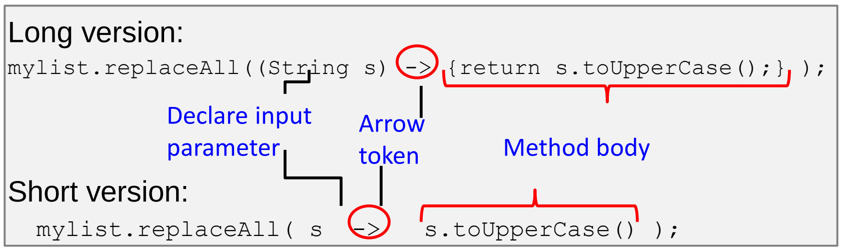
Output:

```
List.replaceAll lambda: [NED, FRED, JESSIE, ALICE, RICK]
```

Lambda Expressions

Lambda expressions are like methods used as the argument for another method. They have:

- Input parameters
- A method body
- A return value



Long version:

```
mylist.replaceAll((String s) -> {return s.toUpperCase();} );
```

Short version:

```
  mylist.replaceAll( s -> s.toUpperCase() );
```

The Enhanced APIs That Use Lambda

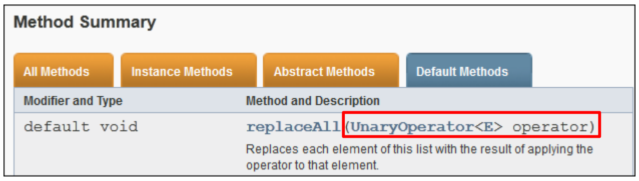
There are three enhanced APIs that take advantage of lambda expressions:

- `java.util.functions` – *New*
  – Provides target types for lambda expressions
- `java.util.stream` – *New*
  – Provides classes that support operations on streams of values
- `java.util` – *Enhanced*
  – Interfaces and classes that make up the collections framework
    – Enhanced to use lambda expressions
    – Includes List and ArrayList

Lambda Types

A lambda *type* specifies the type of expression a method is expecting.

- `replaceAll` takes a `UnaryOperator` type expression.



Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

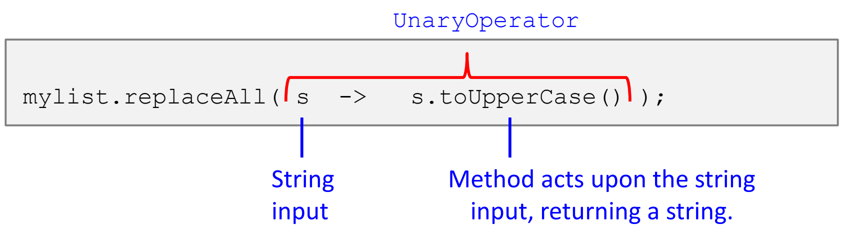
Modifier and Type	Method and Description
`default void`	`replaceAll(UnaryOperator<E> operator)` Replaces each element of this list with the result of applying the operator to that element.

- All of the types do similar things, but have different inputs, statements, and outputs.

The `UnaryOperator` Lambda Type

A `UnaryOperator` has a single input and returns a value of the same type as the input.

- Example: `String` *in* – `String` *out*
- The method body acts upon the input in some way, returning a value of the same type as the input value.
- `replaceAll` example:

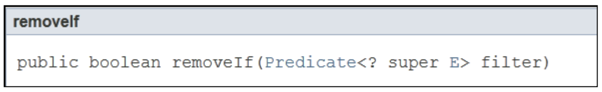


```
mylist.replaceAll( s  ->   s.toUpperCase() );
```

The `Predicate` Lambda Type

A `Predicate` type takes a single input argument and returns a boolean.

- Example: `String` *in* – `boolean` *out*
- `removeIf` takes a `Predicate` type expression.
  - Removes all elements of the `ArrayList` that satisfy the `Predicate` expression



removeIf

```
public boolean removeIf(Predicate<? super E> filter)
```

- Examples:

```
    mylist.removeIf (s -> s.equals(“Rick”));
    mylist.removeIf (s -> s.length() < 5);
```

Exercise 13-2: Using a `Predicate` Lambda Expression

In this exercise, you use the `removeIf()` method to remove all items of the shopping cart whose description matches some value.

- Code the `removeItemFromCart()` method of `ShoppingCart`.
- Create a `Predicate` lambda expression that takes an Item object as input to the expression.

ORACLE

13 - 33

33

Summary

In this lesson, you should have learned the following:

- Polymorphism provides the following benefits:
  - Different classes have the same methods.
  - Method implementations can be unique for each class.
- Interfaces provide the following benefits:
  - You can link classes in different object hierarchies by their common behavior.
  - An object that implements an interface can be assigned to a reference of the interface type.
- Lambda expressions allow you to pass a method call as the argument to another method.

ORACLE

13 - 34

34

Practice 13-1 Overview: Overriding the `toString` Method

This practice covers overriding the `toString` method in `Goal` and `Possession`.

ORACLE

Practice 13-2 Overview: Implementing an Interface

This practice covers implementing the `Comparable` interface so that you can order the elements in an array.

ORACLE

Practice 13-3 (Optional) Overview: Using a Lambda Expression for Sorting

This practice covers using a lambda expression to sort the players.