13

Using Interfaces

Objectives

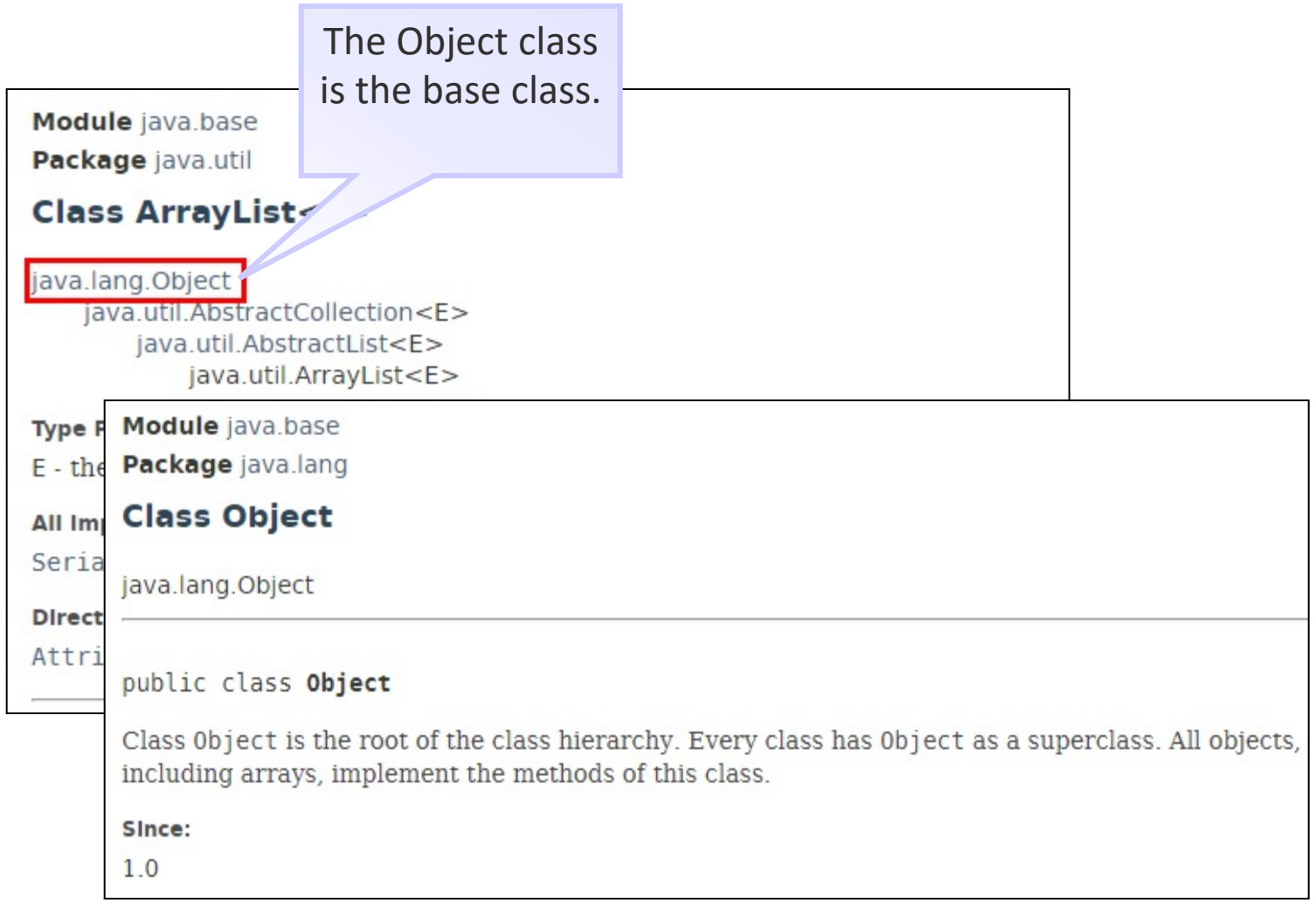
After completing this lesson, you should be able to:

- 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
- Use the local variable type inference feature to declare local variables using `var`
- Write a simple lambda expression that consumes a `Predicate`

Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using local variable type inference
- Using the `List` interface
- Introducing lambda expressions

The `Object` Class



Calling the `toString` Method

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

Overriding `toString` in Your Classes

Shirt class example

```
1  public String toString(){
2     return "This shirt is a " + desc + ";"
3          + " price: " + getPrice() + ","
4          + " color: " + getColor(getColorCode());
5  }
```

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 local variable type inference
- 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.

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

```
public 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, static, and private 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.

```
1 public interface Printable {
2     public void print();
3 }
```

Implicitly abstract

```
1 public class Shirt implements Printable {
2       ...
3       public void print(){
4               System.out.println("Shirt description");
5       }
6 }
```

Implements the `print()` method.

No Multiple Inheritance of State

- Multiple Inheritance of methods is not a problem
- Multiple Inheritance of state is a big problem
  - Abstract classes may have instance and static fields.
  - Interface fields must be static final.

Key difference between abstract classes and interfaces

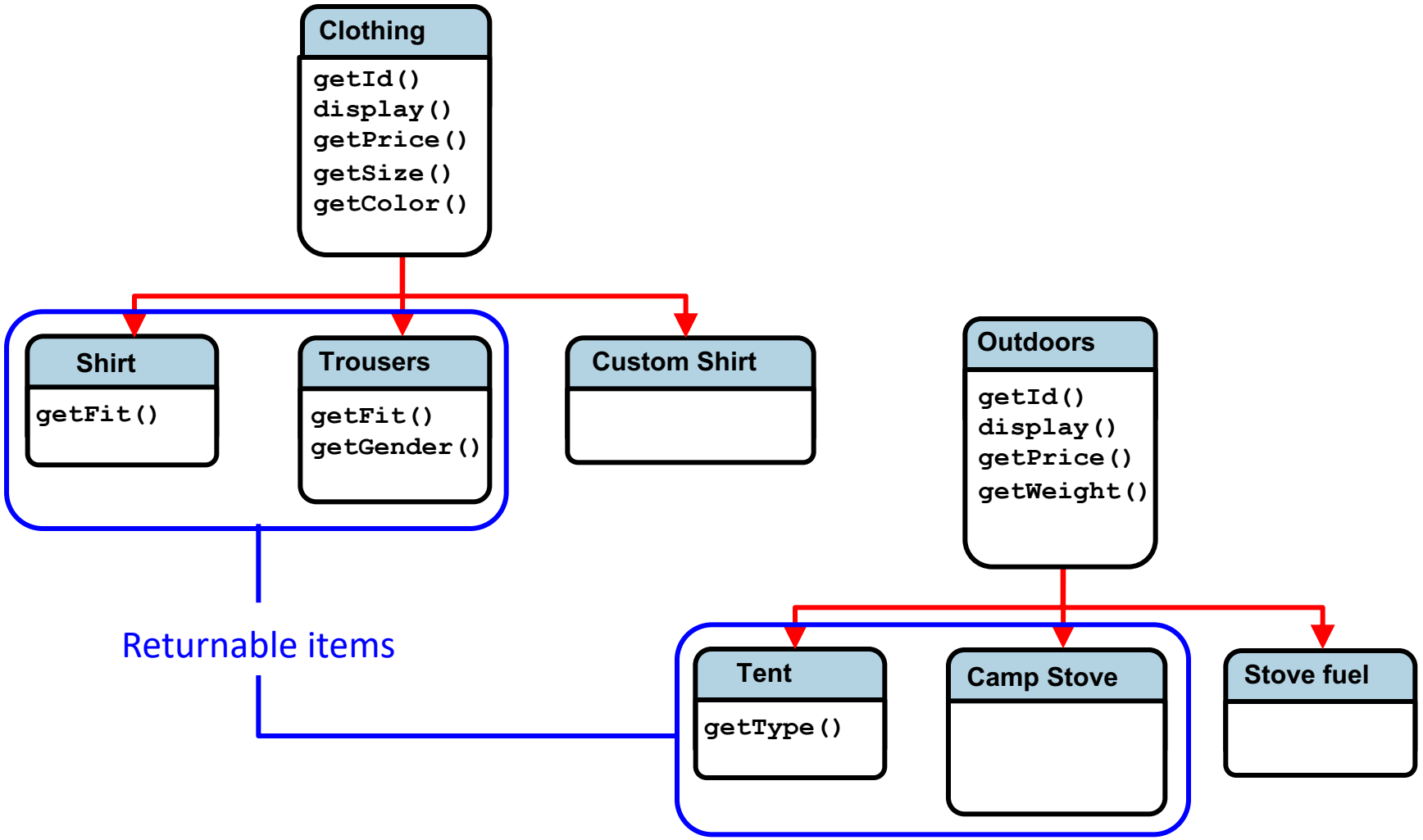
```
public abstract class Red{
    public String color = "Red";
}
```

```
public abstract class Blue{
    public String color = "Blue";
}
```

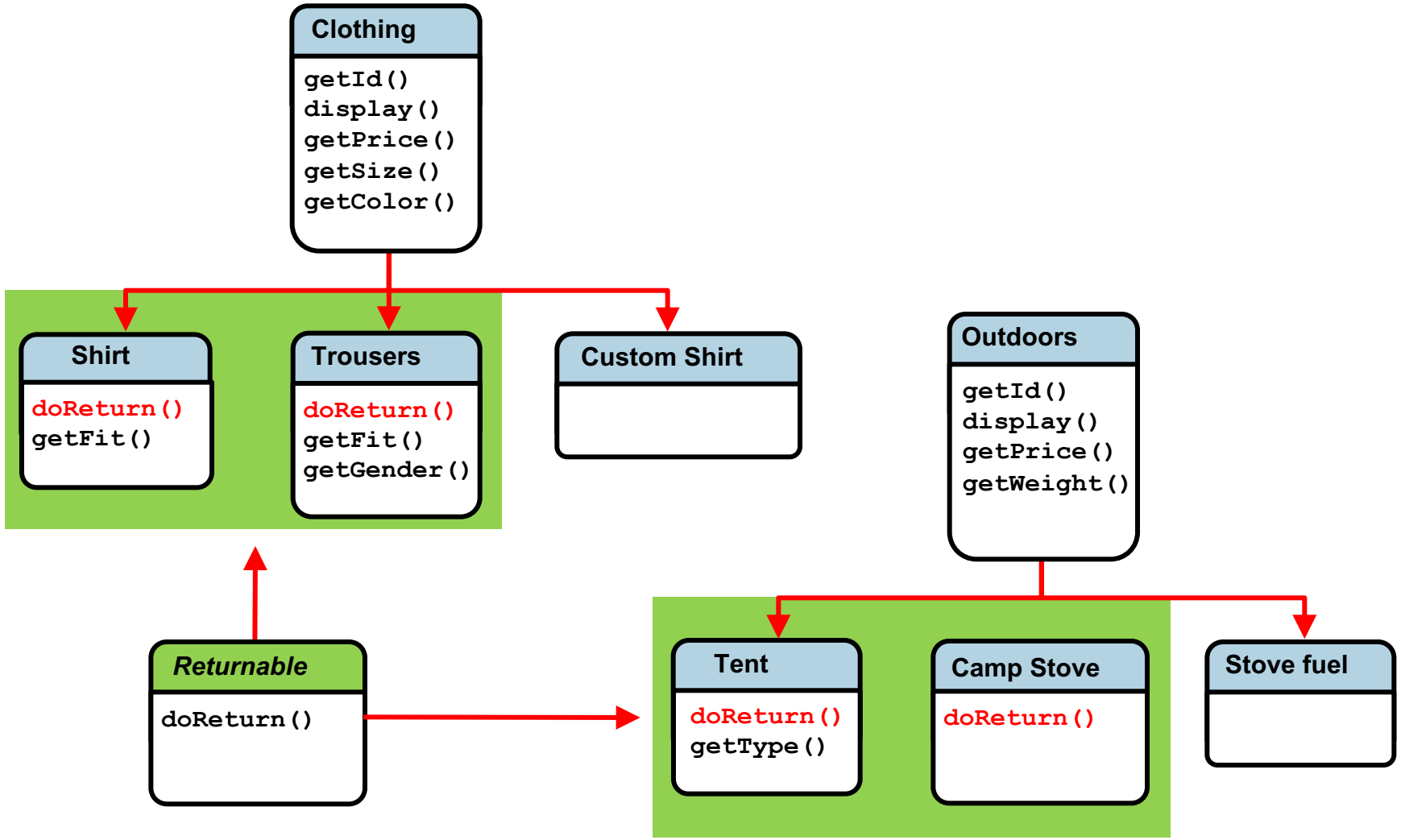
```
public class Purple extends Red, Blue{
      public void printStuff() {
         System.out.println(color);
      }
}
```

Which value of `color` will print?

Multiple Hierarchies with Overlapping Requirements



Using Interfaces in Your Application



Implementing the `Returnable` Interface

`Returnable` interface

```
01  public interface Returnable {
02   public String doReturn();
03 }
```

Implicitly abstract method

`Shirt` class

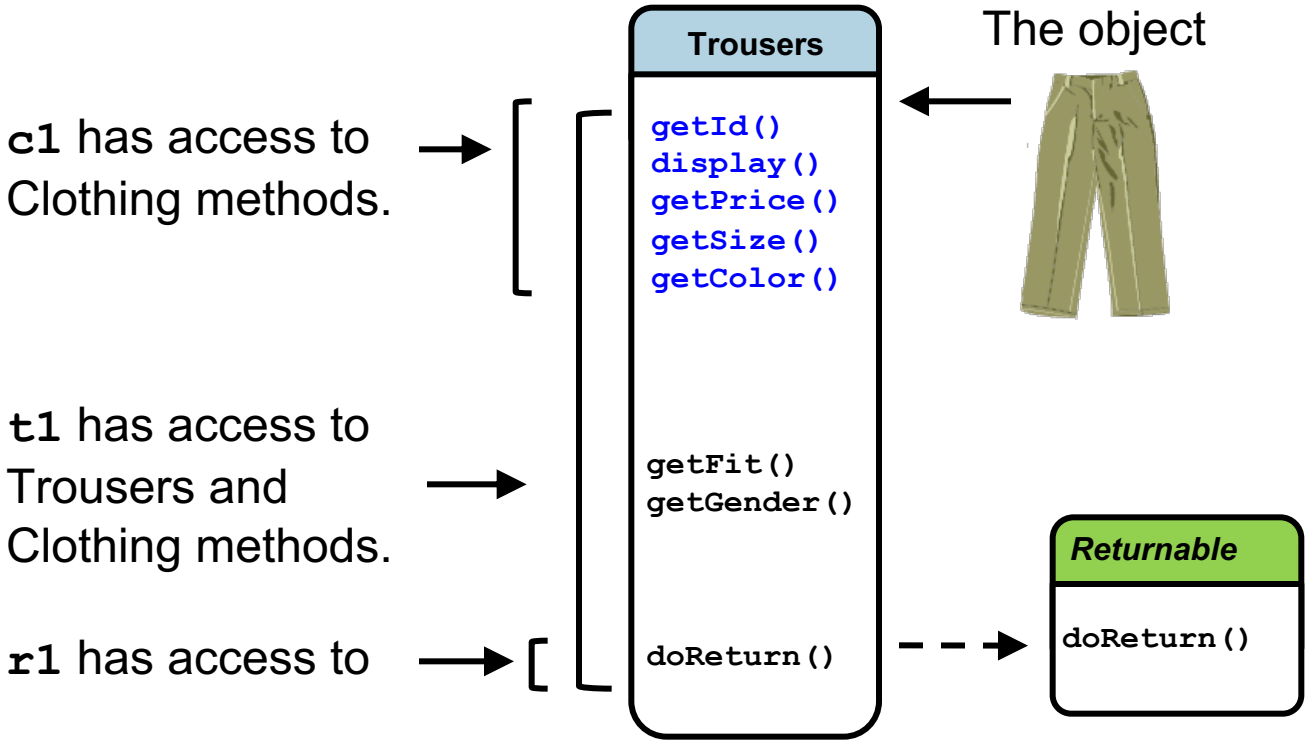
Now, Shirt '**is a**' Returnable.

```
01  public class Shirt extends Clothing implements Returnable {
02   public Shirt(int itemID, String description, char colorCode,
03                double price, char fit) {
04      super(itemID, description, colorCode, price);
05      this.fit = fit;
06   }
07  public String doReturn() {
08    // See notes below
09     return "Suit returns must be within 3 days";
10  }
11  ...< 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 local variable type inference
- Using the `List` interface
- Introducing lambda expressions

What is This Feature?

- Local variable type inference is a new language feature in Java 10.
- Use `var` to declare local variables.
- The compiler infers the datatype from the variable initializer.

Before Java 10

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

Datatype declared twice

Now

```
var list = new ArrayList<String>();
```

Datatype declared once

Benefits

- There's less boilerplate typing.
- Code is easier to read with variable names aligned.

```
String desc = "shirt";
ArrayList<String> list = new
ArrayList<String>();
int price = 20;
double tax = 0.05;
```

```
var desc = "shirt";
var list = new ArrayList<String>();
var price = 20;
var tax = 0.05;
```

- It won't break old code.
  - Keywords cannot be variables names.
  - `var` is not a keyword.
  - `var` is a reserved type name.
  - It's only used when the compiler expects a variable type.
  - Otherwise, you can use `var` as a variable name.

But it's a bad name...

Where Can it be Used?

Yes

- Local variables

```
var x = shirt1.toString();
```

- `for` loop

```
for(var i=0; i<10; i++)
```

- `for-each` loop

```
for(var x : shirtArray)
```

No

- Declaration without an initial value

```
var price;
```

- Declaration and initialization with a null value

```
var price = null;
```

- Fields

```
public var price;
```

- Parameters

```
public void setPrice(var price){}
```

- Method return types

```
public var getPrice(){
    return price;
}
```

Why is The Scope So Narrow?

- Larger scopes increase the potential for issues or uncertainty in inferences.
- To prevent issues, Java restricts the usage of `var`.

```
public var getSomething(var something){
      return something;
}
```

How should this compile?
`something` could be anything!

Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using local variable type inference
- 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

`List` Interface

Module java.base
Package java.util

Interface List<E>

Type Parameters:
E - the type of elements in this list

All Superinterfaces:
`Collection<E>, Iterable<E>`

All Known Subinterfaces:
`ObservableList<E>, ObservableListValue<E>, WritableListValue<E>`

All Known Implementing Classes:
`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, FilteredList, LinkedList, ListBinding, ListExpression, ListProperty, ListPropertyBase, ModifiableObservableListBase, ObservableListBase, ReadOnlyListProperty, ReadOnlyListPropertyBase, ReadOnlyListWrapper, RoleList, RoleUnresolvedList, SimpleListProperty, SortedList, Stack, TransformationList, Vector`

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

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

```
1      var words = new ArrayList();
2      var mylist = words;
```

Using local variable type inference

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`:

```
1    String[] nums = {"one","two","three"};
2    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`:

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

Shortcut:

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

Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using local variable type inference
- Using the `List` interface
- Introducing lambda expressions

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`?

```
1  String[] names = {"Ned","Fred","Jessie","Alice","Rick"};
2  List<String> mylist = new ArrayList(Arrays.asList(names));
3
4  // Display all names in upper case
5  for( var s: mylist){
6     System.out.print(s.toUpperCase()+", ");
7  }
8  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.

Lambda expression

```
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();} );
```

Declare input parameter

Arrow token

Method body

Short version:

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

The Enhanced APIs That Use Lambda

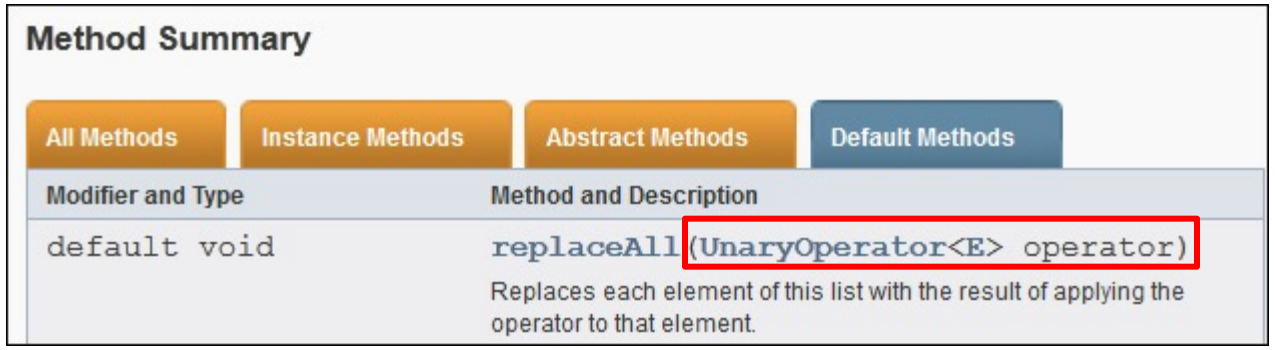
There are three enhanced APIs that take advantage of lambda expressions:

- `java.util.functions`
  - Provides target types for lambda expressions
- `java.util.stream`
  - Provides classes that support operations on streams of values
- `java.util`
  - 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.

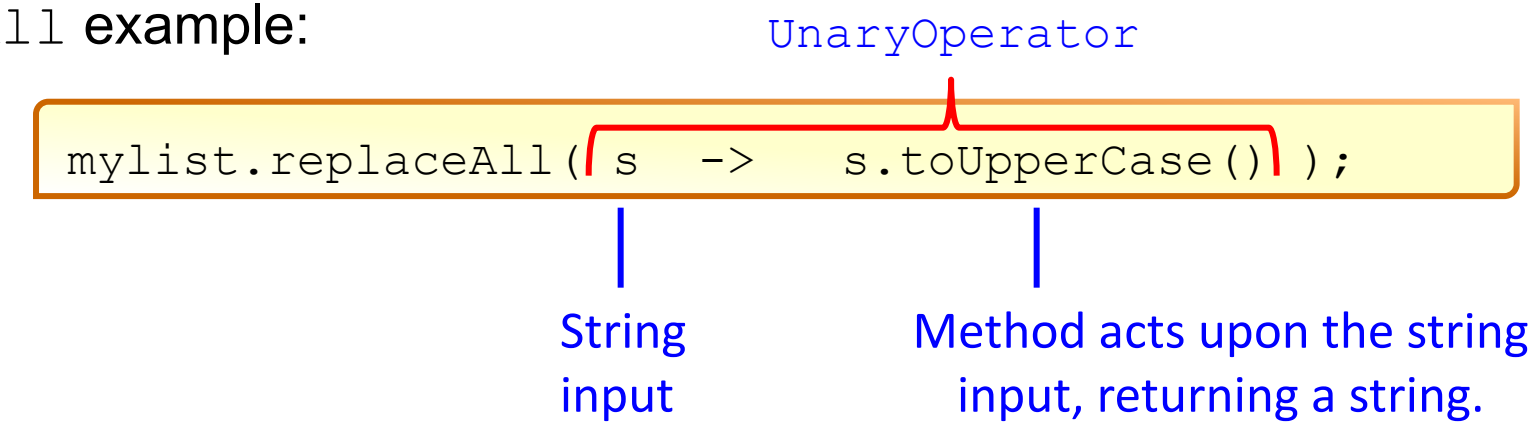


- All of the types do similar things, but have different inputs, statements, and outputs.

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:



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);
```

Summary

In this lesson, you should have learned the following:

- 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
- Use local variable type inference feature to declare local variables using `var`
- Write a simple lambda expression that consumes a `Predicate`