4

Generics and JavaFX Collections

ORACLE

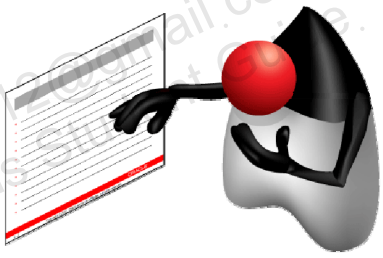
Objectives

After completing this lesson, you should be able to describe:

- Generics
- JavaFX Collections

Topics

- Generics
- JavaFX Collections



Generics

- Provide flexible type safety to your code
- Move many common errors from runtime to compile time
- Provide code that is cleaner and easier to write
- Reduce the need for casting with collections
- Are used heavily in the Java Collections API

Simple Cache Class Without Generics

```
public class CacheString {
    private String message = "";

    public void add(String message){
        this.message = message;
    }

    public String get(){
        return this.message;
    }
}
```

```
public class CacheShirt {
    private Shirt shirt;

    public void add(Shirt shirt){
        this.shirt = shirt;
    }

    public Shirt get(){
        return this.shirt;
    }
}
```


The two examples in the slide show very simple caching classes. Even though each class is very simple, a separate class is required for any object type.

Generic Cache Class

```
public class CacheAny <T>{

    private T t;

    public void add(T t){
        this.t = t;
    }

    public T get(){
        return this.t;
    }
}
```


To create a generic version of the `CacheAny` class, a variable named `T` is added to the class definition surrounded by angle brackets. In this case, `T` stands for "type" and could represent any type. As the example shows, the code has changed to use `t` instead of a specific type information. This change allows the `CacheAny` class to store any type of object.

`T` was chosen not by accident but by convention. A number of letters are commonly used with generics. Here are the conventions and what they stand for.

Note: You could use any identifier you want. These values are merely strongly suggested.

- `T`: Type
- `E`: Element
- `K`: Key
- `V`: Value
- `S`, `U`: Used if there are two, three, or more types

Generics in Action

Compare the type-restricted objects to their generic alternatives:

```
public static void main(String args[]){
    CacheString myMessage = new CacheString(); // Type
    CacheShirt myShirt = new CacheShirt();     // Type

    //Generics
    CacheAny<String> myGenericMessage = new CacheAny<String>();
    CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();

    myMessage.add("Save this for me"); // Type
    myGenericMessage.add("Save this for me"); // Generic

}
```


Note how the one generic version of the class can replace any number of type-specific caching classes. The `add()` and `get()` functions work exactly the same way. In fact, if the `myMessage` declaration were changed to generic, no changes would need to be made to the remaining code.

The example code shown in the slide can be found in the Generics project in the `TestCacheAny.java` file.

Generics with Diamond Operator

- Syntax
  - There is no need to repeat types on the right side of the statement.
  - Angle brackets indicate that type parameters are mirrored.
- Simplifies generic declarations
- Saves typing

```
//Generics
CacheAny<String> myMessage = new CacheAny<>();
}
```


ORACLE

The diamond operator is a new feature in JDK 7. In the generic code, notice how the type definition on the right is always equivalent to the type definition on the left. In JDK 7, you can use the diamond operator to indicate that the right type definition is equivalent to the left type definition. This helps to avoid repeatedly typing redundant information.

Example: `TestCacheAnyDiamond.java`

This example works in an opposite way from a "normal" Java type assignment. In a "normal" assignment, we have the following:

`Employee emp = new Manager();` makes the `emp` object an instance of `Manager`.

But in the case of generics, we have the following:

```
ArrayList<Manager>  managementTeam = new ArrayList<>();
```

The type is determined by the left side of the expression, and not the right side.

`ArrayList` Without Generics

```
1 public class OldStyleArrayList {
2  public static void main(String args[]){
3    List partList = new ArrayList(3);
4
5    partList.add(new Integer(1111));
6    partList.add(new Integer(2222));
7    partList.add(new Integer(3333));
8    partList.add("Oops a string!");
9
10    Iterator elements = partList.iterator();
11    while (elements.hasNext()) {
12      Integer partNumberObject = (Integer) (elements.next()); // error?
13      int partNumber = (int) partNumberObject.intValue();
14
15      System.out.println("Part number: " + partNumber);
16    }
17  }
18 }
```


In the example in the slide, a part number list is created using an `ArrayList`. Using syntax prior to Java version 1.5, there is no type definition. So any type can be added to the list as shown on line 8. It is therefore up to the programmer to know what objects are in the list and what their order is. If an assumption is made that the list is only for `Integer` objects, a runtime error occurs on line 12.

On lines 10–16, with a nongeneric collection, an `Iterator` is used to iterate through the list of items. Notice that a lot of casting is required to get the objects back out of the list so you can print the data.

In the end, there is a lot of needless “syntactic sugar” (extra code) working with collections in this way.

If the line that adds the `String` to the `ArrayList` is commented out, the program produces the following output:

```
Part number: 1111
Part number: 2222
Part number: 3333
```

Generic `ArrayList`

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class GenericArrayList {
5 public static void main(String args[]){
6
7    List<Integer> partList = new ArrayList<>(3);
8
9    partList.add(new Integer(1111));
10    partList.add(new Integer(2222));
11    partList.add(new Integer(3333));
12    partList.add(new Integer(4444)); // ArrayList auto grows
13
14    System.out.println("First Part: " + partList.get(0)); // First item
15    partList.add(0, new Integer(5555)); // Insert an item by index
16
17    // partList.add("Bad Data");  // compile error now
```

ORACLE

With generics, things are much simpler. When the `ArrayList` is initialized on line 6, any attempt to add an invalid value (line 17) results in a compile-time error.

The example in the slide shows several `ArrayList` features:

- Line 12 demonstrates how an `ArrayList` grows automatically when an item is added beyond its original size.
- Line 14 shows how elements can be accessed by their index.
- Line 15 shows how elements can be inserted in the list based on their index.

Note: On line 7, the `ArrayList` is assigned to a `List` type. Using this style enables you to swap out the `List` implementation without changing other code.

Quiz

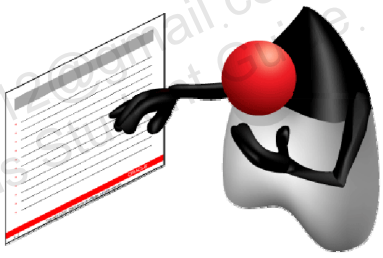
Which of the following is *not* a conventional abbreviation for use with generics?

a. T: Table
b. E: Element
c. K: Key
d. V: Value

Answer: a

Topics

- Generics
- JavaFX Collections



JavaFX Collections and Interfaces

- JavaFX Collections are an extension of the Java Collections Framework.
- JavaFX Collections are defined by the `javafx.collections` package.
- `ObservableList`: A list that enables listeners to track changes when they occur
- `ObservableMap`: A map that enables observers to track changes when they occur
- `ListChangeListener`: An interface that handles list-related events generated by an `ObservableList`

ObservableList

- Extends `javafx.beans.Observable` and `java.util.List`
- Provides a list that supports observability
- Has methods for adding or removing `ListChangeListener`
  - `addListener(ListChangeListener<?super E> listener)`: Adds a listener to this observable list
  - `removeListener(ListChangeListener<? super E> listener)`: Tries to remove a listener from this observable list. If the listener is not attached to this list, nothing happens.
- Has methods for set and retain that add or retain elements of a collection

An `ObservableList` is a list that enables listeners to track changes when they occur.

An observer may be present at most once on a particular list. If an observer is already present and `add` is called a second time, there is no effect. If a `remove` call is made on an observer that is not present, there is no effect.

ObservableMap

- Extends `javafx.beans.Observable` and `java.util.Map`
- Provides a list that supports observability
- Has methods for adding or removing `MapChangeListener`
  - `addListener(MapChangeListener<? super K,? super V> listener)`: Adds a listener to this observable map
  - `removeListener(MapChangeListener<? super K,? super V> listener)`: Tries to remove a listener from this observable map

An `ObservableMap` is a map that enables observers to track changes when they occur.

An observer may be present at most once on a particular list. If an observer is already present and `add` is called a second time, there is no effect. If a `remove` call is made on an observer that is not present, there is no effect.

`FXCollections` Classes

- `FXCollections`: A utility class that consists of static methods that are one-to-one copies of `java.util.Collections` methods
- Wrapper methods return `ObservableList` and are suitable for methods that require `ObservableList` on input.
- Utility methods are used for performance reasons. Methods are optimized to yield only limited numbers of notifications.

ObservableList from Strings

```
public class ListExample {

    public static void main(String[] args) {
        final List<String> list = new ArrayList<String>();
        final ObservableList<String> observableList =
     FXCollections.observableList(list);

        observableList.addListener(new ListChangeListener() {

            @Override
            public void onChanged(ListChangeListener.Change change) {
                System.out.println("List changed");
                System.out.println("Observable list length: " +
                     observableList.size() + "  List length: " + list.size());
            }
        });

        System.out.println("--Adding items--");
        observableList.add("First item");
        observableList.add("Second item");
        observableList.add("Third item");
        observableList.add("Fourth item");

        System.out.println("List contents: " + list.toString());
    }
}
```

The `javafx.collections.FXCollections` class is used to create and return the `ObservableList` objects. `ListChangeListener.Change` represents a change made to an `ObservableList`. The program produces the following output:

```
--Adding items--
List changed
Observable list length: 1  List length: 1
List changed
Observable list length: 2  List length: 2
List changed
Observable list length: 3  List length: 3
List changed
Observable list length: 4  List length: 4
List contents: [First item, Second item, Third item, Fourth item]
```

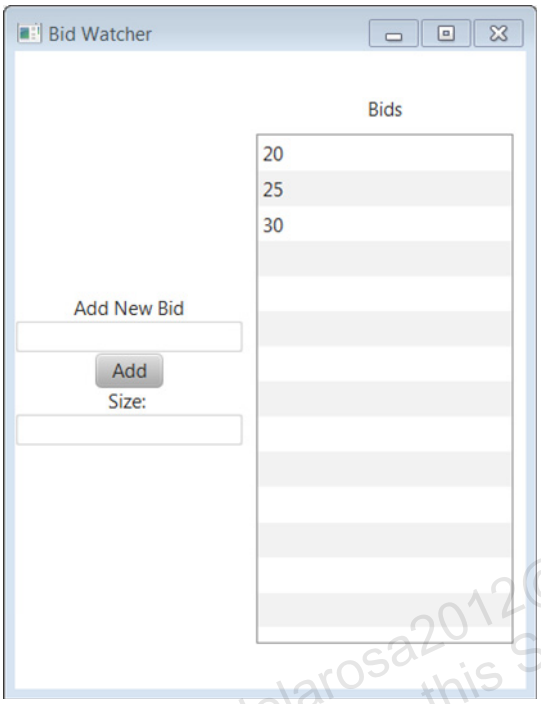
ObservableMap from Strings

```
public class MapExample {

    public static void main(String[] args) {
        final Map<String, String> map = new HashMap<>();
        final ObservableMap<String, String> observableMap =
     FXCollections.observableMap(map);

        observableMap.addListener(new MapChangeListener() {

            @Override
            public void onChanged(MapChangeListener.Change change) {
                System.out.println("Map changed!");
                System.out.println("Observable map length: " +
                    observableMap.size() + "  Map length: " + map.size());
            }
        });

        System.out.println("--Adding map items--");
        observableMap.put("1111", "Couch");
        observableMap.put("1112", "Chair");
        observableMap.put("1113", "Lamp");
        observableMap.put("1114", "Table");

        System.out.println("Map contents: " + map.toString());
    }
}
```


The `javafx.collections.FXCollections` class is used to create and return the `ObservableMap` objects. `MapChangeListener.Change` represents a change made to an `ObservableMap`. The program produces the following output:

```
--Adding map items--
Map changed!
Observable map length: 1  Map length: 1
Map changed!
Observable map length: 2  Map length: 2
Map changed!
Observable map length: 3  Map length: 3
Map changed!
Observable map length: 4  Map length: 4
Map contents: {1111=Couch, 1112=Chair, 1113=Lamp, 1114=Table}
```

Bid Example: Demo



The `BidExample` program in the `ObserveExample` project demonstrates how a `ListView` and `ListChangeListener` can be used to observe the same `ObservableList`.

Quiz

Which of the following is an interface that receives notifications of changes to an `ObservableMap`?

a. `ObservableMap`
b. `javafx.collections`
c. `MapChangeListener`
d. `ObservableList`

Answer: c

Summary

In this lesson, you should have learned how to describe:

- Generics
- JavaFX Collections

Practice 4: Overview

4-1: Adding a Second Listener to Your List