5

Collections and Generics

ORACLE

Objectives

After completing this lesson, you should be able to:

- Describe autoboxing and auto-unboxing feature
- Create a generic method and class
- Describe Java Collections framework
- Create different Collection implementations
- Order collections
- Explain wildcards in generics
- Describe Convenience methods for Collections

Type-Wrapper Classes

- Each primitive type has a corresponding type-wrapper class (in `java.lang` package).
  - Enables you to manipulate primitive-type values as objects
  - This is important, because the various implementations of the Collections manipulate and share objects—they cannot manipulate variables of primitive types.
- The following table lists wrapper classes:

Primitive Type	Wrapper Class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`int`	`Integer`
`float`	`Float`
`double`	`Double`
`long`	`Long`
`short`	`Short`

Autoboxing and Auto-Unboxing

- Java provides boxing and unboxing conversions that automatically convert between primitive-type values and type-wrapper objects.
  - **Boxing:** Converts a primitive value to an object of the corresponding type-wrapper class.
  - **Unboxing:** Converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These conversions are performed automatically and are called as autoboxing and auto-unboxing.

```
Integer[] integerArray = new Integer[5]; // create an array of Integer
integerArray[0] = 10; // Autoboxing,assign primitive 10 to Integer array
int value = integerArray[0]; // Auto-unboxing,get int value of Integer
```

Autoboxing and auto-unboxing are Java language features that enable you to make assignments without formal casting syntax. The casting is performed automatically by the compiler.

Note: Be careful when using autoboxing in a loop. There is a performance cost to using this feature.

Generic Methods

Consider a scenario where you have many overloaded methods, i.e., methods performing the same operation but differing in the argument type. For example:

```
public static void main(String args[]){
   Integer[] intArray = {50, 10, 20,100,50};
   Character[] charArray = {'J', 'A', 'V', 'A'};
    public static void displayArray(Integer[] inputArray) {
       for (Integer element : inputArray) {
           System.out.printf("%s ", element);
        }
          System.out.println();
    }
   public static void displayArray(Character[] inputArray) {
       for (Character element : inputArray) {
          System.out.printf("%s ", element);
        }
         System.out.println();
   }
```

A Generic Method

You can instead replace the overloaded methods by a generic method.

- For example, `displayArray` method in which actual type names are replaced with a generic type name (in this case T).

```
public static void displayArray(T [] inputArray) {
    for (T element : inputArray)  {
        System.out.printf("%s ", element);
    }
    System.out.println();
}
```

- The compiler infers the type argument based on the type of the actual arguments passed.

Generic Classes

- Generic classes provide a means to implement a class in a type-independent manner.
  - You can then instantiate type-specific objects of the generic class.
  - The compiler ensures the type safety of your code.

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

Generic `Cache` Class

```
public class CacheAny <T>{
   private 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 can represent any type. As the example in the slide shows, the code has changed to use `t` instead of a specific type of information. This change allows the `CacheAny` class to store any type of object.

`T` was chosen not by accident but by convention. Specific letters are commonly used with generics.

Note: You can use any identifier you want. The following values are merely strongly suggested.

Here are the conventions:

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

Testing the Generic `Cache` Class

```
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 is changed to generic, no changes need to be made to the remaining code.

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

Generics with Type Inference Diamond Notation

- In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) .
- The compiler determines the type arguments from the context.
- The pair of angle brackets <> is called the **diamond** notation.
- For example, you can create an instance of `CacheAny<String>` with the following statement:

```
CacheAny<String> myMessage = new CacheAny<>();
```

The type inference diamond is a new feature in JDK 7. In the generic code, notice how the right-side type definition is always equivalent to the left-side type definition. In JDK 7, you can use the diamond to indicate that the right type definition is equivalent to the left. This helps to avoid typing redundant information over and over again.

Example: `TestCacheAnyDiamond.java`

Note: In a way, it works in an opposite way from a "normal" Java type assignment. For example, `Employee emp = new Manager();` makes `emp` object an instance of `Manager`.

But in the case of generics:

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

The left side of the expression (rather than the right side) determines the type.

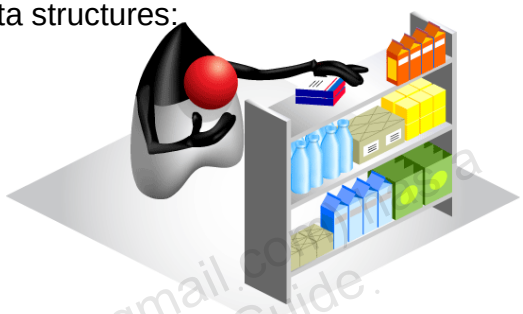
Java SE 9: Diamond Notation with Anonymous Inner Classes

- Prior to Java SE 9, you need to specify class name in angular brackets for inner classes, but in Java SE 9 diamond operator is added for inner class as well.
- For example:

```
public class HelloDemo {

 public static void main(String args[]) {

// Hello<String> hello = new Hello<String>("Prior to Java SE 9") {
     Hello<String> hello = new Hello<>("Java SE 9") {
    @Override
    void hello() {
      System.out.println("Hello " + name);
    }
  };
  hello.hello();
 }
}
```

Collections

- A collection is a single object designed to manage a group of objects.
- Objects in a collection are called *elements*.
  - Used to store, retrieve, and manipulate aggregate data.
  - Represent data items that form a natural group, for example:
    - A mail folder (a collection of letters)
    - A telephone directory (a mapping of names to phone numbers).
- Various collection types implement many common data structures:
  - Stack, queue, tree, and linked list
  - The Collections API relies heavily on generics for its implementation.

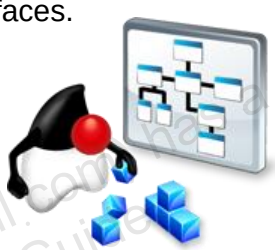


Note: The Collections classes are all stored in the `java.util` package. The `import` statements are not shown in the following examples, but the `import` statements are required for each collection type:

- `import java.util.List;`
- `import java.util.ArrayList;`
- `import java.util.Map;`

Collections Framework in Java

- A collections framework is a unified architecture for representing and manipulating collections.
- All collections frameworks contain the following:
  - Interfaces: Allow collections to be manipulated independently of the details of their representation.
  - Implementations: These are the concrete implementations of the collection interfaces; that is, they are data structures.
  - Algorithms: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.



The Collections classes are all stored in the `java.util` package. The `import` statements are not shown in the following examples, but the `import` statements are required for each collection type:

- `import java.util.List;`
- `import java.util.ArrayList;`
- `import java.util.Map;`

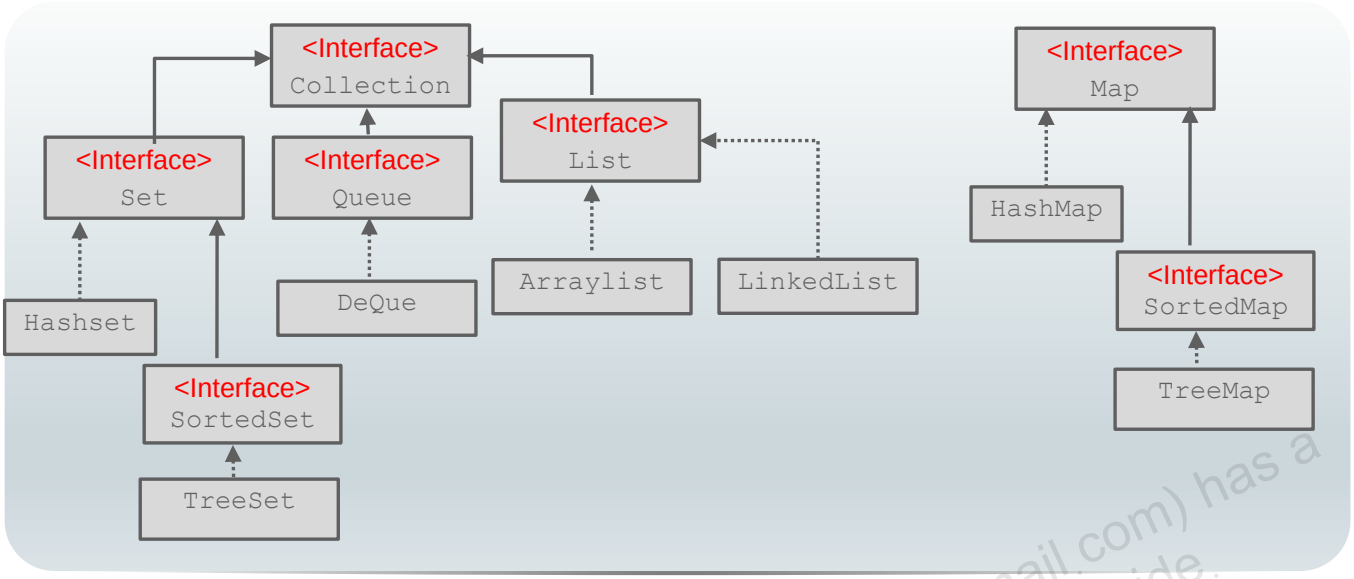
Benefits of the Collections Framework

- The Java Collections Framework provides the following benefits:
  - Reduces programming effort
  - Increases program speed and quality
  - Reduces effort to design new APIs
  - Fosters software reuse

The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Collection Types



The diagram in the slide shows the main interfaces of the Collection framework. The framework is made up of a set of interfaces and some of their implementations for working with a group (collection) of objects.

Characteristics of the Collection Framework

`List`, `Set`, Queue, and `Map` are interfaces in Java, and many concrete implementations of them are available in the Collections API.

The `Map` interface is a separate inheritance because it doesn't extend the `Collection` interface because it represents mappings and not a collection of objects.

Key Collections Interfaces

Interfaces	Implementations
• `List` – Elements are ordered according to how they're added. – Elements are searchable by index. – Duplicates are allowed.	– `ArrayList` – `LinkedList`
• `Set` – Elements cannot be searched by index. – Duplicates are not allowed.	– `HashSet` – `LinkedHashSet` – `TreeSet`
• `Map` – Elements are a key/value pair. – Duplicates are not allowed.	– `HashMap` – `LinkedHashMap` – `HashTable` – `TreeMap`
• `Queue` – First-in, first-out or Last-in, first-out collection that models a waiting line.	– `Deque`

The table in the slide shows the commonly used interfaces and their popular implementation.

ArrayList

- Is an implementation of the `List` interface
  - The list automatically grows if elements exceed initial size.
- Has a numeric index
  - Elements are accessed by index.
  - Elements can be inserted based on index.
  - Elements can be overwritten.
- Allows duplicate items

```
List<Integer> partList = new ArrayList<>(3);
    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add(new Integer(4444)); // ArrayList auto grows
    System.out.println("First Part: " + partList.get(0)); // First item
    partList.add(0, new Integer(5555)); // Insert an item by index
```

`ArrayList` Without Generics

```
public class OldStyleArrayList {
  public static void main(String args[]){
    List partList = new ArrayList(3);

    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add("Oops a string!");

    Iterator elements = partList.iterator();
    while (elements.hasNext()) {
      Integer partNumberObject = (Integer)(elements.next()); //
error?
      int partNumber = partNumberObject.intValue();

      System.out.println("Part number: " + partNumber);
    }
  }
}
```

Runtime error: ClassCastException

In the example in the slide, a part number list is created by using an `ArrayList`. There is no type definition when using syntax prior to Java version 1.5. So any type can be added to the list as shown on line 8. It is up to the programmer to know what objects are in the list and in what order. If the list was only for `Integer` objects, a runtime error would occur 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`

```
public class GenericArrayList {
  public static void main(String args[]) {
    List<Integer> partList = new ArrayList<>(3);

    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add("Bad Data");  // compiler error now

    Iterator<Integer> elements = partList.iterato
    while (elements.hasNext()) {
      Integer partNumberObject = elements.next();
      int partNumber = partNumberObject.intValue();

      System.out.println("Part number: " + partNumber);
    }
  }
}
```

No cast required.

With generics, things are much simpler. When the `ArrayList` is initialized on line 3, any attempt to add an invalid value (line 8) results in a compile-time error.

Note: On line 3, the `ArrayList` is assigned to a `List` type. Using this style enables you to swap out the `List` implementation without changing other code.

TreeSet: Implementation of Set

```
public class SetExample {
   public static void main(String[] args){
       Set<String> set = new TreeSet<>();

       set.add("one");
       set.add("two");
       set.add("three");
       set.add("three"); // not added, only unique

       for (String item:set){
           System.out.println("Item: " + item);
       }
   }
}
```

Set Interface :

A `Set` is an interface that contains only unique elements.

A `Set` has no index.

Duplicate elements are not allowed.

You can iterate through elements to access them.

`TreeSet` provides sorted implementation.

The example in the slide uses a `TreeSet`, which sorts the items in the `set`. If the program is run, the output is as follows:

```
Item: one
Item: three
Item: two
```

`Map` Interface

- A collection that stores multiple key-value pairs
  - Key: Unique identifier for each element in a collection
  - Value: A value stored in the element associated with the key
- Called “associative arrays” in other languages

Key	Value
101	Blue Shirt
102	Black Shirt
103	Gray Shirt

A `Map` is good for tracking things such as part lists and their descriptions (as shown in the slide).

TreeMap: Implementation of Map

```
public class MapExample {

    public static void main(String[] args){
        Map <String, String> partList = new TreeMap<>();
        partList.put("S001", "Blue Polo Shirt");
        partList.put("S002", "Black Polo Shirt");
        partList.put("H001", "Duke Hat");

        partList.put("S002", "Black T-Shirt"); // Overwrite value
        Set<String> keys = partList.keySet();

        System.out.println("=== Part List ===");
        for (String key:keys){
            System.out.println("Part#: " + key + " " +
                               partList.get(key));
        }
    }
 }
```

Some of the key implementation classes include:

- **`TreeMap`:** A map where the keys are automatically sorted
- **`Hashtable`:** A classic associative array implementation with keys and values. `Hashtable` is synchronized.
- **`HashMap`:** An implementation just like `Hashtable` except that it accepts null keys and values. Also, it is not synchronized.

The example in the slide shows how to create a `Map` and perform standard operations on it. The output from the program is:

```
=== Part List ===
Part#: H002 Duke Hat
Part#: S001 Blue Polo Shirt
Part#: S002 Black T-Shirt
```

Stack with `Deque`: Example

```
public class MapExample {

    public static void main(String[] args){
        Map <String, String> partList = new TreeMap<>();
        partList.put("S001", "Blue Polo Shirt");
        partList.put("S002", "Black Polo Shirt");
        partList.put("H001", "Duke Hat");

        partList.put("S002", "Black T-Shirt"); // Overwrite value
        Set<String> keys = partList.keySet();

        System.out.println("=== Part List ===");
        for (String key:keys){
            System.out.println("Part#: " + key + " " +
                               partList.get(key));
        }
    }
}
```

`Deque` Interface

A collection that can be used as a stack or a queue

- It means a “double-ended queue” (and is pronounced “deck”).
- A queue provides FIFO (first in, first out) operations:
  - `add(e)` and `remove()` methods
- A stack provides LIFO (last in, first out) operations:
  - `push(e)` and `pop()` methods

Ordering Collections

- The `Comparable` and `Comparator` interfaces are used to sort collections.
  - Both are implemented by using generics.
- Using the `Comparable` interface:
  - Overrides the `compareTo` method
  - Provides only one sort option
- The `Comparator` interface:
  - Is implemented by using the `compare` method
  - Enables you to create multiple `Comparator` classes
  - Enables you to create and use numerous sorting options

The Collections API provides two interfaces for ordering elements: `Comparable` and `Comparator`.

- **`Comparable`:** Is implemented in a class and provides a single sorting option for the class
- **`Comparator`:** Enables you to create multiple sorting options. You plug in the designed option whenever you want

Both interfaces can be used with sorted collections, such as `TreeSet` and `TreeMap`.

Comparable: Example

```
public class ComparableStudent implements Comparable<ComparableStudent>{
   private String name; private long id = 0; private double gpa = 0.0;

   public ComparableStudent(String name, long id, double gpa){
      // Additional code here
   }
   public String getName(){ return this.name; }
      // Additional code here

   public int compareTo(ComparableStudent s){
       int result = this.name.compareTo(s.getName());
       if (result > 0) { return 1; }
       else if (result < 0){ return -1; }
       else { return 0; }
   }
}
```

The example in the slide implements the `Comparable` interface and its `compareTo` method. Notice that because the interface is designed by using generics, the angle brackets define the class type that is passed into the `compareTo` method. The `if` statements are included to demonstrate the comparisons that take place. You can also merely return a result.

The returned numbers have the following meaning.

- **Negative number:** `s` comes before the current element.
- **Positive number:** `s` comes after the current element.
- **Zero:** `s` is equal to the current element.

In cases where the collection contains equivalent values, replace the code that returns zero with an additional code that returns a negative or positive number.

Comparable Test: Example

```
public class TestComparable {
  public static void main(String[] args){
    Set<ComparableStudent> studentList = new TreeSet<>();

    studentList.add(new ComparableStudent("Thomas Jefferson",
   1111, 3.8));
    studentList.add(new ComparableStudent("John Adams", 2222,
   3.9));
    studentList.add(new ComparableStudent("George Washington",
   3333, 3.4));

    for(ComparableStudent student:studentList){
      System.out.println(student);
    }
  }
}
```

In the example in the slide, an `ArrayList` of `ComparableStudent` elements is created. After the list is initialized, it is sorted by using the `Comparable` interface. The output of the program is as follows:

```
Name: George Washington  ID: 3333  GPA:3.4
Name: John Adams  ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
```

Note: The `ComparableStudent` class has overridden the `toString()` method.

`Comparator` Interface

- Is implemented by using the `compare` method
- Enables you to create multiple `Comparator` classes
- Enables you to create and use numerous sorting options

The example in the next slide shows how to use `Comparator` with an unsorted interface such as `ArrayList` by using the `Collections` utility class.

Comparator: Example

```
public class StudentSortName implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        int result = s1.getName().compareTo(s2.getName());
        if (result != 0) { return result; }
        else {
            return 0;  // Or do more comparing
        }
    }
}
```

```
public class StudentSortGpa implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        if (s1.getGpa() < s2.getGpa()) { return 1; }
        else if (s1.getGpa() > s2.getGpa()) { return -1; }
        else { return 0; }
    }
}
```

Here the compare logic is reversed and results in descending order.

The example in the slide shows the `Comparator` classes that are created to sort based on Name and GPA. For the name comparison, the `if` statements have been simplified.

Comparator Test: Example

```
public class TestComparator {
  public static void main(String[] args){
      List<Student> studentList = new ArrayList<>(3);
      Comparator<Student> sortName = new StudentSortName();
      Comparator<Student> sortGpa = new StudentSortGpa();

      // Initialize list here

      Collections.sort(studentList, sortName);
      for(Student student:studentList){
          System.out.println(student);
      }

      Collections.sort(studentList, sortGpa);
      for(Student student:studentList){
          System.out.println(student);
      }
  }
}
```

The example in the slide shows how the two `Comparator` objects are used with a collection.

Note: Some code has been commented out to save space.

Notice how the `Comparator` objects are initialized on lines 4 and 5. After the `sortName` and `sortGpa` variables are created, they can be passed to the `sort()` method by name. Running the program produces the following output.

```
Name: George Washington  ID: 3333  GPA:3.4
Name: John Adams  ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
Name: John Adams  ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
Name: George Washington  ID: 3333  GPA:3.4
```

Note

- The `Collections` utility class provides a number of useful methods for various collections. Methods include `min()`, `max()`, `copy()`, and `sort()`.
- The `Student` class has overridden the `toString()` method.

Wildcards

- In generics code, the question mark (?) is called the wildcard and represents an unknown type.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable.
- For example, the `sum` method totals the elements in a `List` using a wildcard in the `List` parameter.

```
public static double sum(List<? extends Number> list) {
  double total = 0; //
     for (Number element : list) {
       total += element.doubleValue();
     }
   return total;
 }
```

Wildcards: Upper Bound

- In the `sum` method, `List<? extends Number>` indicates the wildcard extends class `Number`, which means that the wildcard has an **upper bound** of `Number`.
- Thus, the unknown-type argument, `?` must be either `Number` or a subclass of `Number`.
- With the wildcard type argument, method sum can receive an argument a `List` containing any type of `Number`, such as a `List<Integer>`, `List<Double>`, or `List<Number>`.

```
public static void main(String[] args) {
      Integer[] integers = {1, 2, 3, 4, 5};
      List<Integer> integerList = new ArrayList<>();
      //Insert elements into the integerList and then invoke sum method
      sum(integerList));

     Double[] doubles = {1.1, 3.3, 5.5};
      List<Double> doubleList = new ArrayList<>();
      //Insert elements into the doubleList and then invoke sum method
      sum(doubleList));
}
```

Why Use Generics?

In a nutshell, code that uses generics has many benefits over nongeneric code:

- Stronger type checks at compile time.
  - A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.
  - Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
- Elimination of casts.
  - The following code snippet with generics doesn't require casting:

```
List<String> list = new ArrayList<String>();
    list.add("hello");
    String s = list.get(0);    // no cast
```

- Enabling programmers to implement generic algorithms.

  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Java SE 9: Convenience Methods for Collections

Java SE 9 adds new static convenience factory methods to interfaces `List`, `Set`, and `Map` that enables you to create small immutable collections; that is, they cannot be modified once they are created.

- Their goals are to:
  - Reduce boilerplate code
  - Improve readability
  - Improve performance
  - Get the same amount of work done by typing less code

Please refer to JEP 269 on convenience methods for collections.

The word *factory* indicates that these methods create objects. These methods are ***convenient*** because you simply pass the elements as arguments to a convenience factory method, which creates the collection and adds the elements to the collection for you.

In lessons on Advanced and Parallel streams, you'll see how using lambdas and streams with immutable entities can help you create "parallelizable" code that will run more efficiently on today's multi-core architectures.

# of Convenience Method

- Java SE 8: Collections require one line of code to add each element. For example:

```
List<String> testList = new ArrayList<>();
testList.add("A");
testList.add("B");
testList.add("C");
testList.add("D");
testList.add("E");
```

- Java SE 9: The same work is done in one line with the of method.

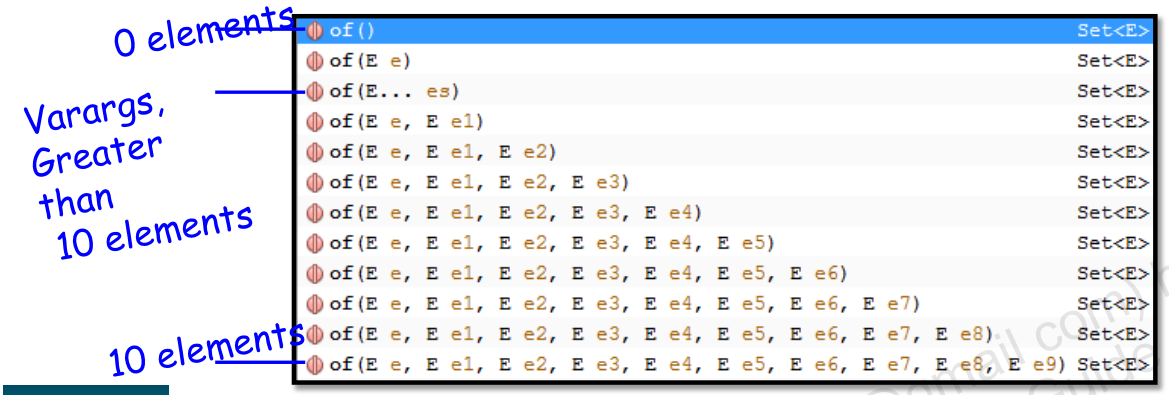
```
List<String> testList = List.of("A", "B", "C", "D", "E");                  // List

Set<String> testSet = Set.of("A", "B", "C", "D", "E");                     //Set

Map<String, Integer> testMap = Map.of("A", 1, "B", 2, "C", 3, "D", 4, "E", 5);  //Map
```

Key Value

This slide demonstrates the convenience factory method, `of`, for a `List`, a `Set`, and a `Map`.

Overloading `of` Method

- Most variants return a collection of a specific size.
- The smallest returns a collection of 0 elements.
- The largest returns a collection of 10 elements.
- The varargs variant is used for all other sizes or greater than 10 elements.



Why Overload the `of` Method for zero to 10 elements?

The of method is overloaded for zero to 10 elements because research showed that these handle the vast majority of cases in which immutable collections are needed. *They eliminate the extra overhead of processing variable-length argument lists. This improves the performance of applications that create small immutable collections.*

For performance and space efficiency.

Varargs cause performance overhead relating to:

- Temporary array allocation
- Initialization
- Garbage collection

Specifying variants with 10 parameters or less covers the majority of use cases. This avoids the need to use the vararg variant and the associated performances overhead. This is the case for `Lists` and `Sets`. `Maps` are slightly different.

ofEntries Method for Maps

```
Map<String, Integer> testMap = Map.ofEntries(
    entry("A", 1),
    entry("B", 2),
    entry("C", 3),
    entry("D", 4),
    entry("E", 5),
    entry("F", 6),
    entry("G", 7),
    entry("H", 8),
    entry("I", 9),
    entry("J", 10),
    entry("K", 11));
```

This slide demonstrates creating a Map using method `ofEntries` for more than 10 key and value pairs.

If a `Map` needs more than 10 elements, key and value pairs must be boxed as an `entry`.

There is no varargs variant for the `of` method in the `Map` interface. Instead, `ofEntries` accommodates returning a `Map` of indeterminate size. `ofEntries` is a static factory method of the `Map` interface.

Features of Convenience Methods

- Immutability:
  - `of` and `ofEntries` return immutable collections.
  - Methods like `add`, `set`, and `remove` throw `UnsupportedOperationException`.
  - It's thread-safe.
- No null values:
  - Null values are disallowed as `List` or `Set` elements, `Map` keys or values. This avoids later `NullPointerException`.

```
Set<String> testSet = Set.of("A", "B", null, "D", "E");
```

- No duplicates:
  - Duplicates in Sets and Maps cause an `IllegalArgumentException`.

```
Set<String> testSet = Set.of("A", "B", "A", "D", "E");
```

Summary

In this lesson, you should have learned how to:

- Describe autoboxing and auto-unboxing feature
- Create a generic method and class
- Describe Java Collections framework
- Create different Collection implementations
- Order collections
- Explain wildcards in generics
- Describe Convenience methods for Collections

Practice 5: Overview

This practice covers the following topics:

- 5-1: Creating a map to store a part number and count
- 5-2: Using convenience method `of`
- 5-3: Using convenience method `ofEntries`

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

Quiz

Which interface would you use to create multiple sort options for a collection?

A. `Comparable`
B. `Comparison`
C. `Comparator`
D. `Comparinator`

Quiz

The `of` convenience method is overloaded. How many variants of the `of` method exist in the `Set` interface?

A. 2
B. 3
C. 4
D. 8
E. 12

Quiz

Which two are important consideration when deciding to use an `of` factory method to create a `List`?

A. Whether elements need to be added to the `List` later

B. Whether you will have more than 10 elements

C. Whether the `List` may contain a null value

D. Whether the `List` should contain duplicates