Introduction to Programming I

Lab 12

Munir Makhmutov, Mansur Khazeev, Sami Sellami and Furqan Haider

Agenda

- Revision of generics in Java: from different angle
- Warm-up exercises
- Solving Problems
- Q&A

Learning outcome:

- Strengen the knowledge generic in Java
- Application of generics in real-world problems
- Code-review & improving code quality

Java generics

Data Flair
Java
Generics
Class
01
Multiple Type parameters
02
03
Functions
04
Advantages

https://data-flair.training/blogs/java-generics/

What are Java Generics?

Generics in Java are like templates in C++. The thought is to permit Integer, String and so on and user-defined types, to be a parameter to methods, classes, and interfaces.

For instance, classes like HashSet, ArrayList, HashMap, and so forth utilize generics exceptionally well. We can utilize them for any kind.

What are Java Generics?

JDK 5.0 introduced **Java Generics** with the aim of reducing bugs and adding an extra layer of abstraction over types.

In a nutshell, **generics** enable ***types*** (integer, string...etc) to be parameters when defining classes, interfaces and methods.

A **generic class** is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

Generic class example

A Simple Box Class

```
public class Box {
   private Object object;

   public void set(Object object) {
     this.object = object;
   }
   public Object get() {
     return object;
   }
}
```

A Generic Version of the Box Class

```
/**
* Generic version of the Box class.
* @param <T> the type of the value being boxed
*/
public class Box<T> {
   // T stands for "Type"
   private T t;

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

Java Generics in Class

```java
class Test<T> {
// An object of type T is declared
       T obj;
       Test(T obj) {
           this.obj = obj;  // constructor
       }
       public T getObject() {
           return this.obj;
       }
    }

class Main {
   public static void main (String[] args) {
       Test<Integer> iObj = new Test<Integer>(15);
       System.out.println(iObj.getObject());
       Test<String> sObj = new Test<String>("DataFlair");
       System.out.println(sObj.getObject());
   }
}
```

Type parameter naming conventions

A **type parameter**, also known as a type variable, is **an identifier that specifies a generic type name.**

By convention, type parameter names are **single**, **uppercase letters**. Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

Generic class instantiation

```
Box<Integer> integerBox = new Box<Integer>();
```

Or in Java SE 7 and later:

```
Box<Integer> integerBox = new Box<>();
```

This pair of angle brackets, **<>**, is informally called *the diamond*.

Multiple type parameters

A generic class can have **multiple** type parameters. Here, the generic *OrderedPair* class implements the generic *Pair* interface.

Note that a generic interface is created following the same conventions as a generic class.

```java
public interface Pair<K, V> {
   public K getKey();
   public V getValue();
}
```

```java
public class OrderedPair<K, V> implements Pair<K, V>
{

   private K key;
   private V value;

   public OrderedPair(K key, V value) {
      this.key = key;
      this.value = value;
   }

   public K getKey()   { return key; }
   public V getValue() { return value; }
}
```

Multiple type parameters: instantiation

```java
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

The code, new **OrderedPair<String, Integer>**, instantiates **K** as a **String** and **V** as an **Integer**. Therefore, the parameter types of *OrderedPair*'s constructor are *String* and *Integer*, respectively.

Using the diamond operator **<>** :

```java
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

Parameterized types

A type parameter (that is, **K** or **V**) can be substituted with a **parameterized type**. For example, using the *OrderedPair<K, V>* example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

Generic methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared.

Static and non-static generic methods are allowed, as well as generic class constructors.

Java Generic Methods

We can also write generic methods in Java as they can be called by generic arguments which are handled by the compiler in Java.

```
class Test {
     static <T> void genericDisplay (T element) {
           System.out.println(element.getClass().getName() + " = " + element);
     }

     public static void main(String[] args) {
              genericDisplay(11);
              genericDisplay("data flair");
              genericDisplay(1.0);
     }
}
```

Generic method declaration

Here, the ***Util*** class includes a **generic method**, compare, which compares two ***Pair*** objects:

```java
public class Util {
   public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2)
{
       return p1.getKey().equals(p2.getKey()) &&
              p1.getValue().equals(p2.getValue());
   }
}
```

```java
public class Pair<K, V> {

   private K key;
   private V value;

   public Pair(K key, V value) {
       this.key = key;
       this.value = value;
   }

   public void setKey(K key) { this.key = key; }
   public void setValue(V value) { this.value = value; }
   public K getKey()   { return key; }
   public V getValue() { return value; }
}
```

Generic method invoking

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

Bounded generics

Type parameters can be bounded (restricted) and we can restrict the types that a method accepts.

```
public <T extends Number> List<T> fromArrayToList(T[] a) {
   // ...
}
```

Wildcards

Wildcards are represented by the question mark ***?*** in Java, and we use them to refer to an unknown type.

Non-generic version

```java
void printCollection(Collection c) {
   Iterator i = c.iterator();
   for (k = 0; k < c.size(); k++) {
       System.out.println(i.next());
   }
}
```

Naive generic version

```java
void printCollection(Collection<Object> c) {
   for (Object e : c) {
       System.out.println(e);
   }
}
```

Generic version with wildcards

```java
void printCollection(Collection<?> c) {
   for (Object e : c) {
       System.out.println(e);
   }
}
```

The naive generic code only takes **Collection<Object>**, which is **not** a supertype of all kinds of collections.

Collection<?> ("collection of unknown"), is a collection whose element type matches anything, this solves the issue.

Upper bounded wildcards

An upper bounded wildcard **restricts** the unknown type to be a specific type or a **subtype** of that type and is represented using the *extends* keyword.

```
public static double sumOfList(List<? extends Number> list) {
   double s = 0.0;
   for (Number n : list)
       s += n.doubleValue();
   return s;
}
```

The upper bounded wildcard, **`<? extends Number>`**, matches **`Number`** and any subtype of **`Number`**.

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));
// will output sum = 6.0
```

```
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
// will output sum = 7.0
```

Lower bounded wildcards

An lower bounded wildcard **restricts** the unknown type to be a specific type or a ***super type*** of that type.

```
public static void addNumbers(List<? super Integer> list) {
   for (int i = 1; i <= 10; i++) {
       list.add(i);
   }
}
```

Primitive data types

One **restriction** of generics in Java is that the **type parameter cannot be a primitive type.**

For example, the following **does not** compile:

```java
List<int> list = new ArrayList<>();
list.add(17);
```

Therefore, type parameters must be **convertible to *Object***. Since **primitive types don't extend *Object***, we can't use them as type parameters. However, Java provides boxed types for primitives:

```java
List<Integer> list = new ArrayList<>();
list.add(17);
int first = list.get(0);
```

Primitive data types

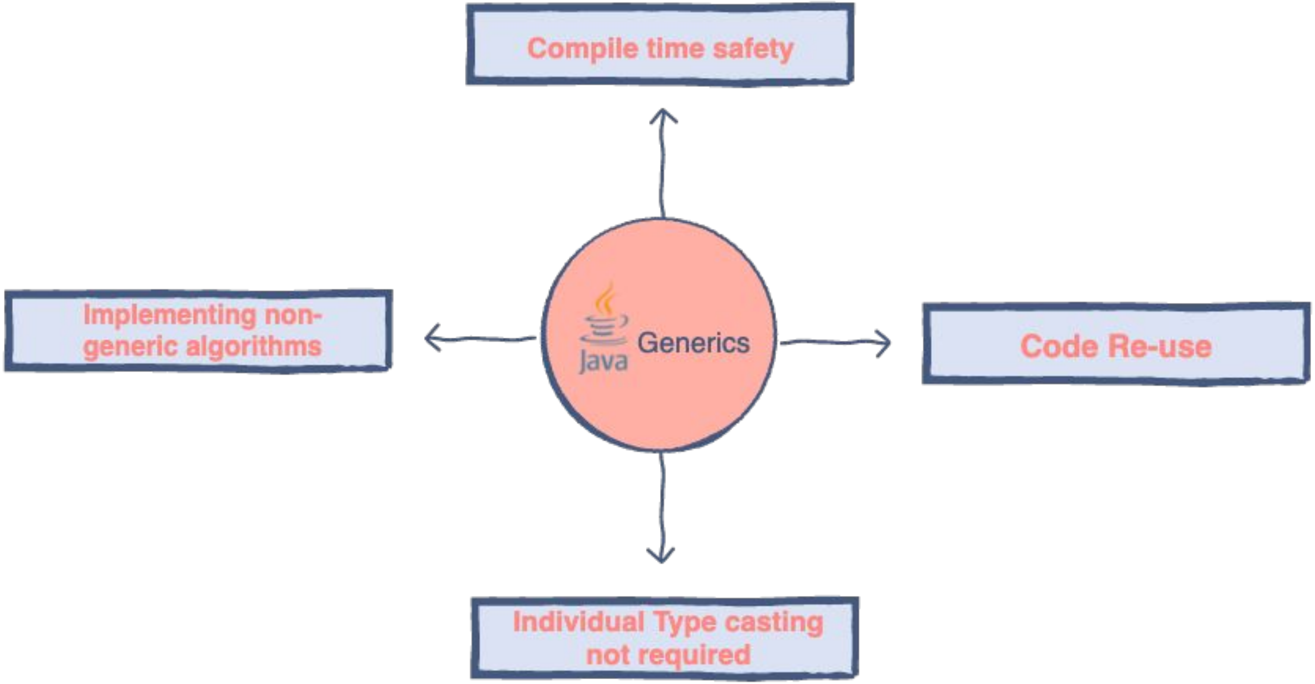
```
byte
short
int
long
float
double
boolean
char
```

Corresponding library class

```
package java.lang

class Byte
class Short
class Integer
class Long
class Float
class Double
class Boolean
class Character
```

Advantages of Java Generics



Compile time safety

Implementing non-generic algorithms

Code Re-use

Individual Type casting not required

Advantages of Java Generics

- Stronger type checks at compile time. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts.

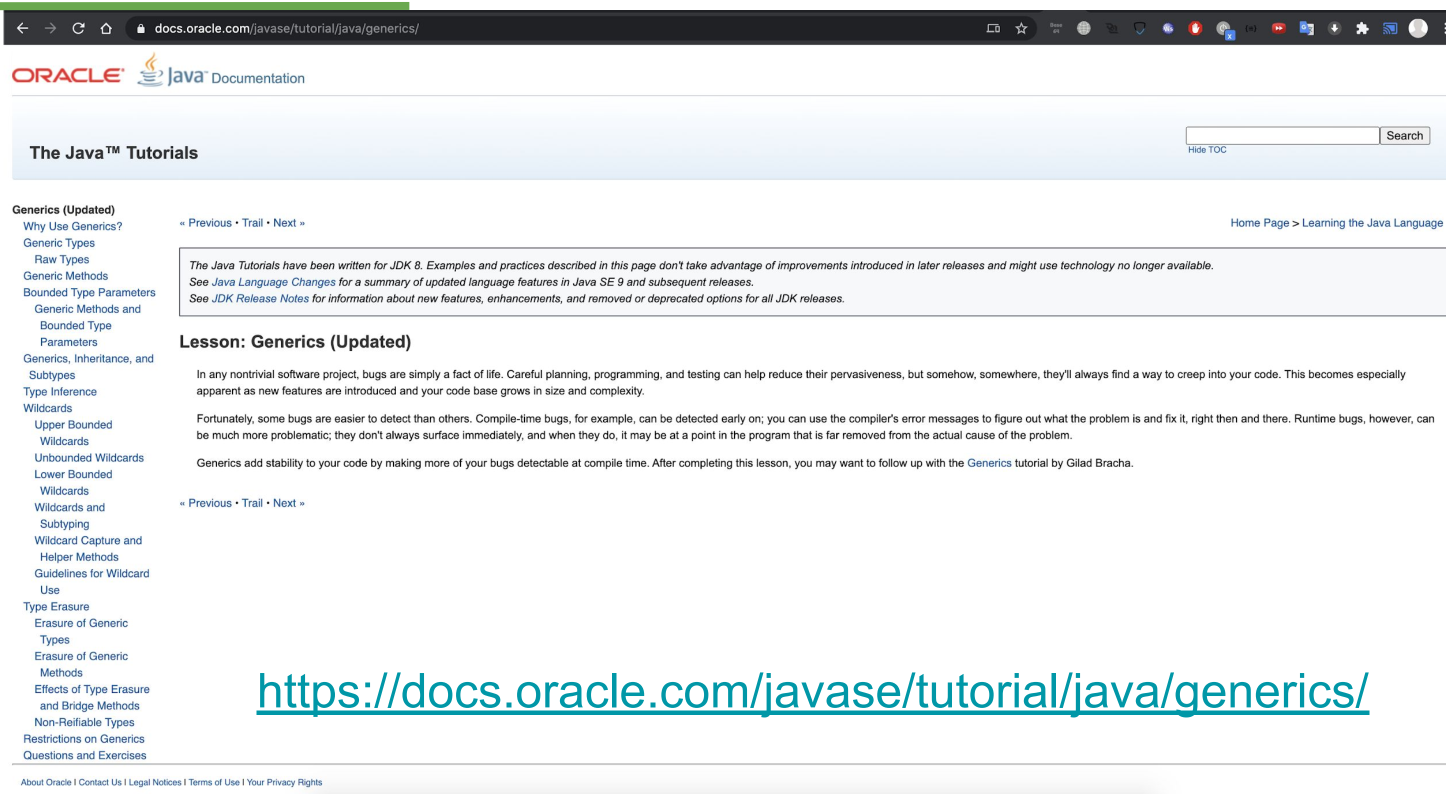
```java
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

→

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

- Enabling programmers to implement generic algorithms.
- Code reuse.

For More Details



docs.oracle.com/javase/tutorial/java/generics/

ORACLE Java Documentation

The Java™ Tutorials

Search

Hide TOC

Generics (Updated)

- Why Use Generics?
- Generic Types
  - Raw Types
- Generic Methods
- Bounded Type Parameters
  - Generic Methods and Bounded Type Parameters
- Generics, Inheritance, and Subtypes
- Type Inference
- Wildcards
  - Upper Bounded Wildcards
  - Unbounded Wildcards
  - Lower Bounded Wildcards
  - Wildcards and Subtyping
  - Wildcard Capture and Helper Methods
  - Guidelines for Wildcard Use
- Type Erasure
  - Erasure of Generic Types
  - Erasure of Generic Methods
  - Effects of Type Erasure and Bridge Methods
  - Non-Reifiable Types
- Restrictions on Generics
- Questions and Exercises

« Previous • Trail • Next »

Home Page > Learning the Java Language

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Lesson: Generics (Updated)

In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness, but somehow, somewhere, they'll always find a way to creep into your code. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.

Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there. Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem.

Generics add stability to your code by making more of your bugs detectable at compile time. After completing this lesson, you may want to follow up with the Generics tutorial by Gilad Bracha.

« Previous • Trail • Next »

About Oracle | Contact Us | Legal Notices | Terms of Use | Your Privacy Rights

https://docs.oracle.com/javase/tutorial/java/generics/

Warm Up Exercises

Will the following class compile? If not, why?

```
public final class Algorithm {
   public static <T> T max(T x, T y) {
       return x > y ? x : y;
   }
}
```

Warm Up Exercises

Will the following method compile? If not, why?

```java
public static void print(List<? extends Number> list) {
    for (Number n : list) {
        System.out.print(n + " ");
    }
    System.out.println();
}
```

Warm Up Exercises

Will the following method compile? If not, why?

```java
public class Singleton<T> {

   public static T getInstance() {
       if (instance == null) {
           instance = new Singleton<T>();
       }
       return instance;
   }

   private static T instance = null;
}
```

Warm Up Exercises

Consider this class:

```
class Node<T> implements Comparable<T> {
   public int compareTo(T obj) { /* ... */ }
   // ...
}
```

Will the following code compile? If not, why?

```
Node<String> node = new Node<>();
Comparable<String> comp = node;
```

Exercise 1

Design a class that acts as a library for the following kinds of media: book, video, and newspaper. Provide one version of the class that uses generics and one that does not. Feel free to use any additional APIs for storing and retrieving the media.

Exercise 1 solution: Media class

```java
abstract class Media {
   private String author;
   private String title;

   public Media(String author, String title) {
       this.author = author;
       this.title = title;
   }
   public String getAuthor() { return author; }
   public void setAuthor(String author) { this.author = author; }
   public String getTitle() { return title; }
   public void setTitle(String title) { this.title = title; }
}

class Book extends Media {
   public Book(String author, String title) { super(author, title); }
}

class Video extends Media {
   public Video(String author, String title) { super(author, title); }
}

class Newspaper extends Media {
   public Newspaper(String author, String title) { super(author, title); }
}
```

Exercise 1 solution

Non-generic solution

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

@SuppressWarnings("unchecked")
public class Library {
    private List resources = new ArrayList();
    public void addMedia(Media x) {
        resources.add(x);
    }
    public Media retrieveLast() {
        int size = resources.size();
        if (size > 0) {
            return (Media)resources.get(size - 1);
        }
        return null;
    }
}
```

Generics solution

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

public class GenericLibrary<E extends Media> {
    private List<E> resources = new ArrayList<E>();
    public void addMedia(E x) {
        resources.add(x);
    }
    public E retrieveLast() {
        int size = resources.size();
        if (size > 0) {
            return resources.get(size - 1);
        }
        return null;
    }
}
```

Exercise 1: Testing solution

```
class MainApp {
   public static void main (String[] args)  {
       GenericLibrary<Book> myLib = new GenericLibrary<>();
       myLib.addMedia(new Book("Author1", "Title1"));
       myLib.addMedia(new Book("Author2", "Title2"));
       myLib.addMedia(new Book("Author3", "Title3"));

       Book lastBook = myLib.retrieveLast();
       System.out.println(
           lastBook.getClass().getName()  + " : " +
           lastBook.getTitle()  + " by " +
           lastBook.getAuthor()
       );
   }
}
```

```
class MainApp {
   public static void main (String[] args)  {
       Library myLib = new Library();
       myLib.addMedia(new Book("Author1", "Title1"));
       myLib.addMedia(new Book("Author2", "Title2"));
       myLib.addMedia(new Book("Author3", "Title3"));

       Book lastBook =  (Book)  myLib.retrieveLast();
       System.out.println(
           lastBook.getClass().getName()  + " : " +
           lastBook.getTitle()  + " by " +
           lastBook.getAuthor()
       );
   }
}
```

Note:

- In the non-generic version, type casting the return value from the *retrieveLast* method to *Book* is necessary.
- The type cast is no longer needed in the generic version because the *GenericLibrary* instance has been constrained to deal with *Book* objects.

Exercise 2

Sketch the class definition and method signatures for a Stack behaviour (LIFO), parameterized by the type of element on the stack. Give the method signatures for *push*, *pop*, and *isEmpty*.

Exercise 3

Sketch the class definition and method signatures for a *Dictionary* class, which allows one to store or look up a value indexed by a key. Give the method signatures for *get*, *put*, *isEmpty*, *keys*, and *values*. The last two methods should return parameterized collections (This class is similar to the builtin class HashMap in the Java collections library)

References

- https://data-flair.training/blogs/java-generics/
- https://docs.oracle.com/javase/tutorial/java/generics/
- https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html
- https://courses.cs.washington.edu/courses/cse341/08au/java/exercises.pdf