# **COURSEWARE**

Agile Fundamentals  Jira  Git  Databases Introduction  Java Beginner  Maven  Testing (Foundation)  Java Intermediate  O Optionals O JDBC CRUD Exceptions O SOLID Principles O Single Responsibility O Open/Closed Liskov Substituiton O Interface Segregation O Dependency Inversion O Best Practice O Design Patterns O Creational Design Patterns O Structural Design Patterns O Structural Design Patterns O Collection & Map O HashSets O HashMaps C Enums O Logging O Generics O Lambda Expressions O Streams O Complexity O Input and Output O Input and Output O Local Type Inference	Professional Skills	
Git  Databases Introduction  Java Beginner  Maven  Testing (Foundation)  Java Intermediate  O Optionals O JDBC CRUD O Exceptions O SOLID Principles O Single Responsibility O Open/Closed O Liskov Substituiton O Interface Segregation O Dependency Inversion O Best Practice O Design Patterns O Creational Design Patterns O Structural Design Patterns O Structural Design Patterns O Collection & Map O HashSets O HashMaps O Enums O Logging O Generics O Lambda Expressions O Streams O Complexity O Input and Output	Agile Fundamentals	
Databases Introduction  Java Beginner  Maven  Testing (Foundation)  Java Intermediate  O Optionals O JDBC CRUD O Exceptions O SOLID Principles O Single Responsibility O Open/Closed O Liskov Substituiton O Interface Segregation O Dependency Inversion O Best Practice O Design Patterns O Creational Design Patterns O Structural Design Patterns O Structural Design Patterns O Collection & Map O HashSets O HashMaps O Enums O Logging O Generics O Lambda Expressions O Streams O Complexity O Input and Output	Jira	
Java Beginner  Maven  Testing (Foundation)  Java Intermediate  Optionals JDBC CRUD  Exceptions SOLID Principles Single Responsibility Open/Closed Liskov Substituiton Interface Segregation Dependency Inversion Best Practice Design Patterns Creational Design Patterns Structural Design Patterns Behavioural Design Patterns Collection & Map HashSets HashMaps Enums Logging Generics Lambda Expressions Streams Complexity Input and Output	Git	
Maven  Testing (Foundation)  Java Intermediate  Optionals  JDBC CRUD  Exceptions  SOLID Principles  Single Responsibility  Open/Closed  Liskov Substituiton  Interface Segregation  Dependency Inversion  Best Practice  Design Patterns  Creational Design Patterns  Structural Design Patterns  Structural Design Patterns  Collection & Map  HashSets  HashMaps  Enums  Logging  Generics  Lambda Expressions  Streams  Complexity  Input and Output	Databases Introduction	
Testing (Foundation)  Java Intermediate  Optionals  JDBC CRUD  Exceptions  SOLID Principles  Single Responsibility  Open/Closed  Liskov Substituiton  Interface Segregation  Dependency Inversion  Best Practice  Design Patterns  Creational Design Patterns  Structural Design Patterns  Structural Design Patterns  Collection & Map  HashSets  HashMaps  Enums  Logging  Generics  Lambda Expressions  Streams  Complexity  Input and Output	Java Beginner	
Java Intermediate  Optionals JDBC CRUD Exceptions SOLID Principles Single Responsibility Open/Closed Liskov Substituiton Interface Segregation Dependency Inversion Best Practice Design Patterns Creational Design Patterns Structural Design Patterns Behavioural Design Patterns Collection & Map HashSets HashMaps Enums Logging Generics Lambda Expressions Streams Complexity Input and Output	Maven	
<ul> <li>Optionals</li> <li>JDBC CRUD</li> <li>Exceptions</li> <li>SOLID Principles</li> <li>Single Responsibility</li> <li>Open/Closed</li> <li>Liskov Substituiton</li> <li>Interface Segregation</li> <li>Dependency Inversion</li> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	Testing (Foundation)	
<ul> <li>JDBC CRUD</li> <li>Exceptions</li> <li>SOLID Principles</li> <li>Single Responsibility</li> <li>Open/Closed</li> <li>Liskov Substituiton</li> <li>Interface Segregation</li> <li>Dependency Inversion</li> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	Java Intermediate	
<ul> <li>Exceptions</li> <li>SOLID Principles</li> <li>Single Responsibility</li> <li>Open/Closed</li> <li>Liskov Substituiton</li> <li>Interface Segregation</li> <li>Dependency Inversion</li> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Structural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Optionals
<ul> <li>SOLID Principles</li> <li>Single Responsibility</li> <li>Open/Closed</li> <li>Liskov Substituiton</li> <li>Interface Segregation</li> <li>Dependency Inversion</li> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	JDBC CRUD
<ul> <li>Single Responsibility</li> <li>Open/Closed</li> <li>Liskov Substituiton</li> <li>Interface Segregation</li> <li>Dependency Inversion</li> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Exceptions
<ul> <li>Open/Closed</li> <li>Liskov Substituiton</li> <li>Interface Segregation</li> <li>Dependency Inversion</li> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	SOLID Principles
<ul> <li>Liskov Substituiton</li> <li>Interface Segregation</li> <li>Dependency Inversion</li> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Single Responsibility
<ul> <li>Interface Segregation</li> <li>Dependency Inversion</li> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Open/Closed
<ul> <li>Dependency Inversion</li> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Liskov Substituiton
<ul> <li>Best Practice</li> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Interface Segregation
<ul> <li>Design Patterns</li> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Dependency Inversion
<ul> <li>Creational Design Patterns</li> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Best Practice
<ul> <li>Structural Design Patterns</li> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Design Patterns
<ul> <li>Behavioural Design Patterns</li> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Creational Design Patterns
<ul> <li>Collection &amp; Map</li> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Structural Design Patterns
<ul> <li>HashSets</li> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Behavioural Design Patterns
<ul> <li>HashMaps</li> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	Collection & Map
<ul> <li>Enums</li> <li>Logging</li> <li>Generics</li> <li>Lambda Expressions</li> <li>Streams</li> <li>Complexity</li> <li>Input and Output</li> </ul>	0	HashSets
<ul><li>Logging</li><li>Generics</li><li>Lambda Expressions</li><li>Streams</li><li>Complexity</li><li>Input and Output</li></ul>	0	HashMaps
<ul><li>Generics</li><li>Lambda Expressions</li><li>Streams</li><li>Complexity</li><li>Input and Output</li></ul>	0	Enums
<ul><li>Lambda Expressions</li><li>Streams</li><li>Complexity</li><li>Input and Output</li></ul>	0	Logging
<ul><li>Streams</li><li>Complexity</li><li>Input and Output</li></ul>	0	Generics
<ul><li>Complexity</li><li>Input and Output</li></ul>	0	Lambda Expressions
<ul><li>Input and Output</li></ul>	0	Streams
	0	Complexity
O Local Type Inference	0	Input and Output
	0	Local Type Inference

HTML

# Generics

#### Contents

- Overview
- Tutorial
  - Bounded Generics
    - Upper Bound
    - Lower Bound
- Exercises

#### Overview

Generics were introduced with the aim of reducing bugs and adding an extra layer of abstraction over types.

Generics enforce type correctness at compile time and enable implementing generic algorithms without causing any extra overhead to our applications as we do not need to define a type, this will not however work with primitive data types.

### **Tutorial**

Generics add stability to our code by making more bugs detectable at compile time.

If we have a method that takes an object, it will accept everything except primitive data types.

If we cast the object to an Integer at the end and then pass in a String, we will end up getting a runtime exception.

```
public class Cage<T> {
    private T object;

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

public T get() {
        return object;
    }
}
```

In the above example we are defining a class that uses generics, which is shown by the <T> in the class definition.

We also declare a variable called object of the generic type T, and have two methods add, which sets the class variable of object to the generic sent in, and get which returns the generic object in the class.

NOTE: It is known that Object is the supertype of all Java classes, however a collection of Object is not the supertype of any collection.

For example a List<Object> is not the supertype of List<String>.

Using generics for this class allows it to be used by various objects, so if we had a class of Elephant and a class of Lion, both could use the methods within the Cage class due to the fact that it is using generics.

However if the add method took a parameter of type Elephant, then the Lion class would no longer be able to use the add method.

## **Bounded Generics**

CCC Javascript Spring Boot Selenium Sonarqube Advanced Testing (Theory) Cucumber MongoDB **Express NodeJS** React **Express-Testing** Networking Security Cloud Fundamentals **AWS Foundations AWS Intermediate** Linux **DevOps** Jenkins Introduction Jenkins Pipeline

Markdown

IDE Cheatsheet

Generics can be bounded, which means they can be restricted.

For example we can specify that a method accepts a type and all its subclasses (upper bound) or a type and its superclasses (lower bound).

### **Upper Bound**

To declare an upper bound generic we can do the following.

```
import java.util.List;

public class Cage<T extends Animal> {

    private List<T> list;

    public void add(List<T> list) {
        this.list = list;
    }

    public List<T> get() {
        return list;
    }
}
```

The generics used within the class above will all be restricted to using types that are either the type Animal or a subclass of Animal.

Note that in this context, the extends keyword is used in a general sense to mean either extends (classes) or implements (interfaces).

We can also have multiple upper bounds, to do this simply use a "&" followed by the next upper bound.

```
import java.util.List;

public class Cage<T extends Mammal & Reptile> {

    private List<T> list;

    public void add(List<T> list) {
        this.list = list;
    }

    public List<T> get() {
        return list;
    }
}
```

In this case we will only allow the types Mammal and Reptile as well as their subclasses.

#### **Lower Bound**

Lower bound can be defined as shown below.

```
import java.util.List;

public class Cage {

   private List<? super Elephant> list;

   public void add(List<? super Elephant> list) {
       this.list = list;
   }

   public List<? super Elephant> get() {
       return list;
   }
}
```

With lower bound we don't define the generic in the class definition, instead wherever the generic would be defined we also need to define the lower bound.

The lower bound of <? super Elephant> means that we restrict the generic to an unknown class that is a superclass of Elephant.

The "?" within the generic specifies a wildcard and simply refers to an unknown type.

# **Exercises**

There are no exercises for this module.