DZone

Understanding the Use Cases of Java Generics

Here's a deep dive into the world of Generics, including an overview of how they work and five use cases where they come in handy.

by Narendran Solai Sridharan · Aug. 14, 17 · Java Zone · Tutorial

Understanding general use cases of generics solves half of the problem. First things first, we should know what they are, why to consider them, and where they apply.

What Is It?

Consider a simple add method, seen below. You cannot pass long, float, or double types as inputs to this method, right?

```java
public static int add(int a, int b) {
    return a + b;
}
```

If we can abstract out the data type from the method, we get a new method, as seen below. Here, **<T>** is the **type parameter**, similar to a parameter we declare for a method. The value we pass **<Integer>** or **<Double>** for the type parameter is the **type argument**, similar to our method argument we pass.

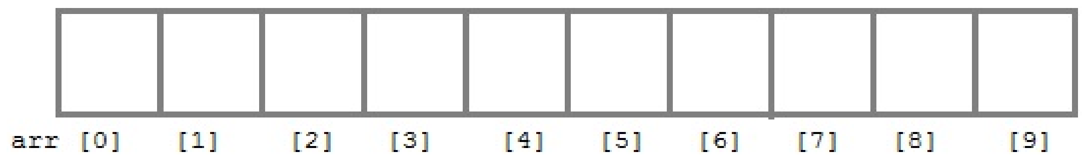
```java
public class Main {
    //In the below 2 methods, only Data Type which extends Number are allowed.
    public static < T extends Number > double addStaticMethod(T a, T b) {
        return a.doubleValue() + b.doubleValue();
    }
    public < T extends Number > double addInstanceMethod(T a, T b) {
```

```
    public < T extends Number > double addInstanceMethod(T a, T b) {
        return a.doubleValue() + b.doubleValue();
    }

    public static void main(String[] args) {
        // static method invocation with Type Argument. It is type safe, only Integer is allowed.
        System.out.println(Main. < Integer > addStaticMethod(3, 4));
        //static method invocation without Type Argument, not type safe. Both Integer and Float is allowed.
        System.out.println(addStaticMethod(3, 4.3));

        Main m = new Main();
        //Instance method invocation with Type Argument.
        System.out.println(m. < Double > addInstanceMethod(3.2, 4.3));
    }
}
```

Consider a **data structure** now. For simplicity, let's think about Array. Can we create an array of any type? No, we can't. we can create an array of Integer**,** Float, or of any specific type. Forget about the array implementation of any language and ask,"**Can we abstract out the data type from this data structure?**"



Answer: Yes, we can. In Java, **ArrayList** is such a class that does it. When you say `List<String> = new ArrayList<>();` , it creates an array of String. When you pass Integer as type argument instead of a string, it creates an array of Integer, and so on.

Despite having said that about ArrayLists, we aren't going get into their implementation due to its complexity. So, we shall take a single box and investigate how to make the box — a Generic box from a Specific Typed box.

Consider the following code. You can put a String into a SpecilizedStringBox Object and you can get a String out of it.

```java
public class SpecilizedStringBox  {

    private String item;

    public String getItem() {
        return item;
    }

    public void setItem(String item) {
        this.item = item;
    }
}
```

Now if we abstract out the data type "String" from the SpecilizedStringBox, we get a generic box represented by the following code, which can take String, Integer, Boolean, or any data type, of course.

```java
public class GenericBox<T>  {

    private T item;

    public T getItem() {
        return item;
    }

    public void setItem(T item) {
        this.item = item;
    }
}
```

So, using Generics is all about abstracting the type from a method or a class to create generic methods or classes applicable to more type than a specific type.

Why?

A simple answer is to abstract out data types, allowing you to reuse code and to maintain it more easily.

Where Do They Apply?

It looks like we can apply Generics by refactoring an existing specific typed method or a box. Until we deal with data structures and primitive data types, it looks easy, but we do create a lot of data types with various classes. Mixing the Generic Programming Paradigm with OOP makes it very difficult to make a choice of whether to apply generics. Understanding where you can apply them solves half of the problem.

This article lets you know some of the use cases of generics, including where they are generally applied, and also make sure that you too can apply generics if you encounter such use cases.

Java incorporated Generics in Java 5.0 to achieve:

1. Type safety ensures that once the type argument is applied, no other data type is allowed into the method or box and avoids the requirement of casting.
2. Generic programming/parametric polymorphism

C++ Template Programming helps us achieve Generic Programming/Parametric Polymorphism. The same Algorithmic Templates can be morphed according to their data types (predefined/user-defined) thereby reusing the same code/program. Nothing different. The same is applicable for Java.

Let's get into the common use cases of Generics.

Use Case Type 1: Algorithms and Data Structures Are First-Class Use Cases for Generics

Algorithms go hand in hand with data structures. A simple change in the data structure in an algorithm could change its complexity.

Data in the data structure has a type. Abstracting out this type with a type parameter is achieved with Generics.

Input parameters of any algorithm have a data type. Abstracting out the types from input parameters is achieved with Generics.

So, Generics are well-suited for any specific algorithm working with a particular data structure. In fact, Generics were designed mainly for Java's Collection API.

If you write your own data structure, do try to apply Generics. Apart from the Java Collection API, you could encounter Generics better use in Guava, Apache Common Collections, FastUtils, JCtools, and Eclipse Collections.

Use Case Type 2: Value Typed Boxes or Single Element Containers

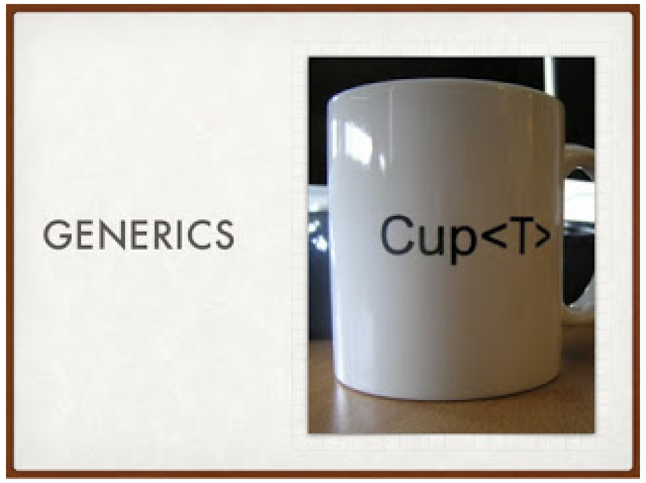
Data structures with the Generics type are Generics Boxes. Classes such as ArrayList, LinkedList, etc., represent data structures and act as Generics Boxes of their kind.

Sometimes, Generic boxes do appear as a single element instead of a collection. They just act as holders or wrappers of data of a particular data type. For example: Entry<K, V> in a Map, Node<K,V>, Pair<K, V>, and algebraic data types like Optional<T>, Choice<U, V>, etc.

ThreadLocal and AtomicReference are very good examples of Single Element Containers that apply algorithms required for concurrent access.

Such use cases sometimes justify their use while others don't. A box can hold any type of item — earlier, we could put anything into a box. But now we start to classify: This box is for toys, the next box is for pens, etc.

A Cup as a Holder can hold either Tea, Coffee, or any beverage. Cup is a good example of real-time object types (Tea, Coffee, etc.) holders. A Bus can carry both Men or Women. If we make it type safe to allow only Women, we can call it a Ladies'/Women's Bus. Needless to say, it may be or may not be appropriate. The catch is that business use cases, especially wrappers or Holders, also provide opportunities to apply Generics. Ask whether the business wrapper or holder's usage is inclined toward the data structure kind of use. If so, generics usage will do better.



Use Case Type 3: Generic Util Methods With Abstract Classes

Generic algorithms need not be always tied to particular data structures or algorithms. Sometimes, it can be applied to most abstract groups of data structures based on the contract the concrete implementations satisfy.

In Java, we have the "**Collections**" util Class.

Check the following methods from the class to get an idea of what kind of methods can be implemented.

Collection Factories Methods (Empty, Singleton):

- emptyList, emptyMap, emptySet, etc.,

- singleton, singletonList, singletonMap etc.,

- ### Wrapper Methods (Synchronized, UnModifiable, Checked Collection):

- synchronizedCollection, synchronizedSet, synchronizedMap, etc.,
- unmodifiableCollection, unmodifiableSet, unmodifiableList, etc.,
- checkedCollection, checkedList, checkedSet, etc.,

A Few More Generic Methods Fall Into Four Major Categories

1. Changing the element order in a list: reverse, rotate, shuffle, sort, swap

2. Changing the contents of a list: copy, fill, replaceAll

3. Finding extreme values in a collection: max, min

4. Finding specific values in a list: binarySearch, indexOfSubList, lastIndexOfSubList

They represent reusable functionality, in that they are applied to Lists (or in some cases to Collections) of any type. We can find a lot of Generics methods applicable to most of the Collection types in general.

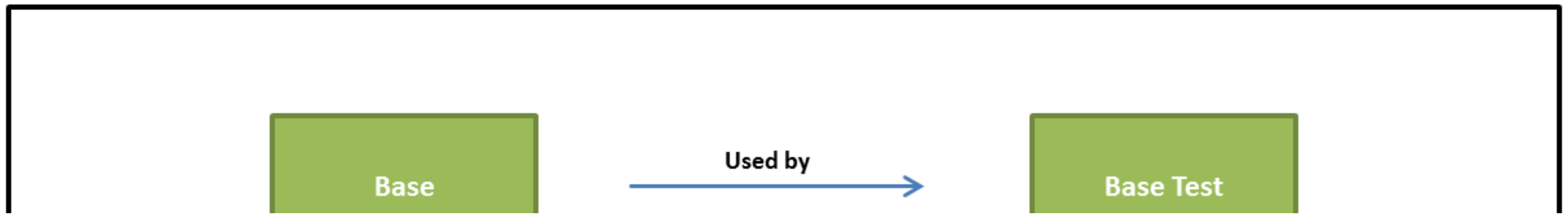
Use Case Type 4: Generic Methods in Parallel Hierarchies of Classes

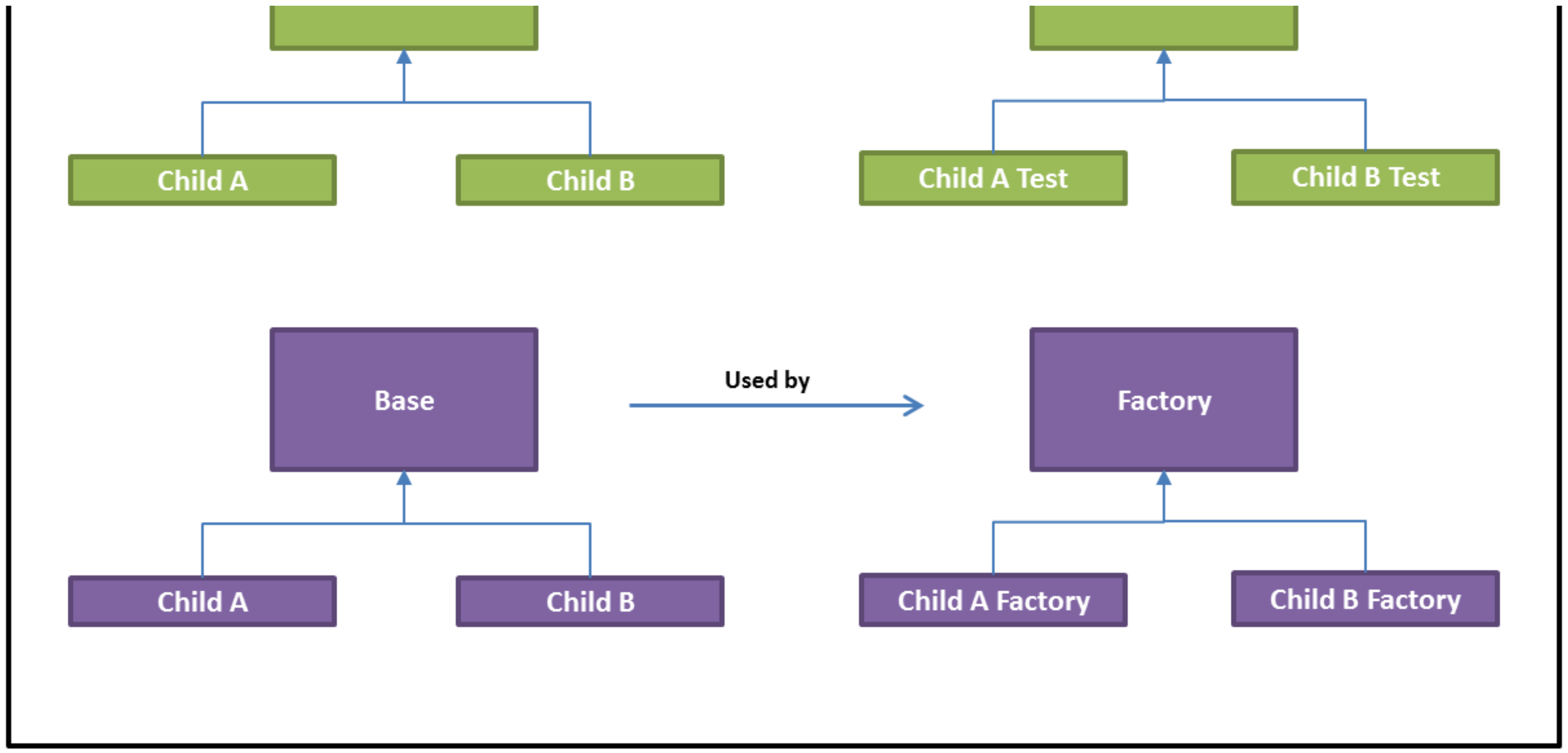
JpaRepository, CrudRepository in Spring Framework have been built with Generics. Create, update, find, findAll, delete, etc. are generic methods applicable for all entities.

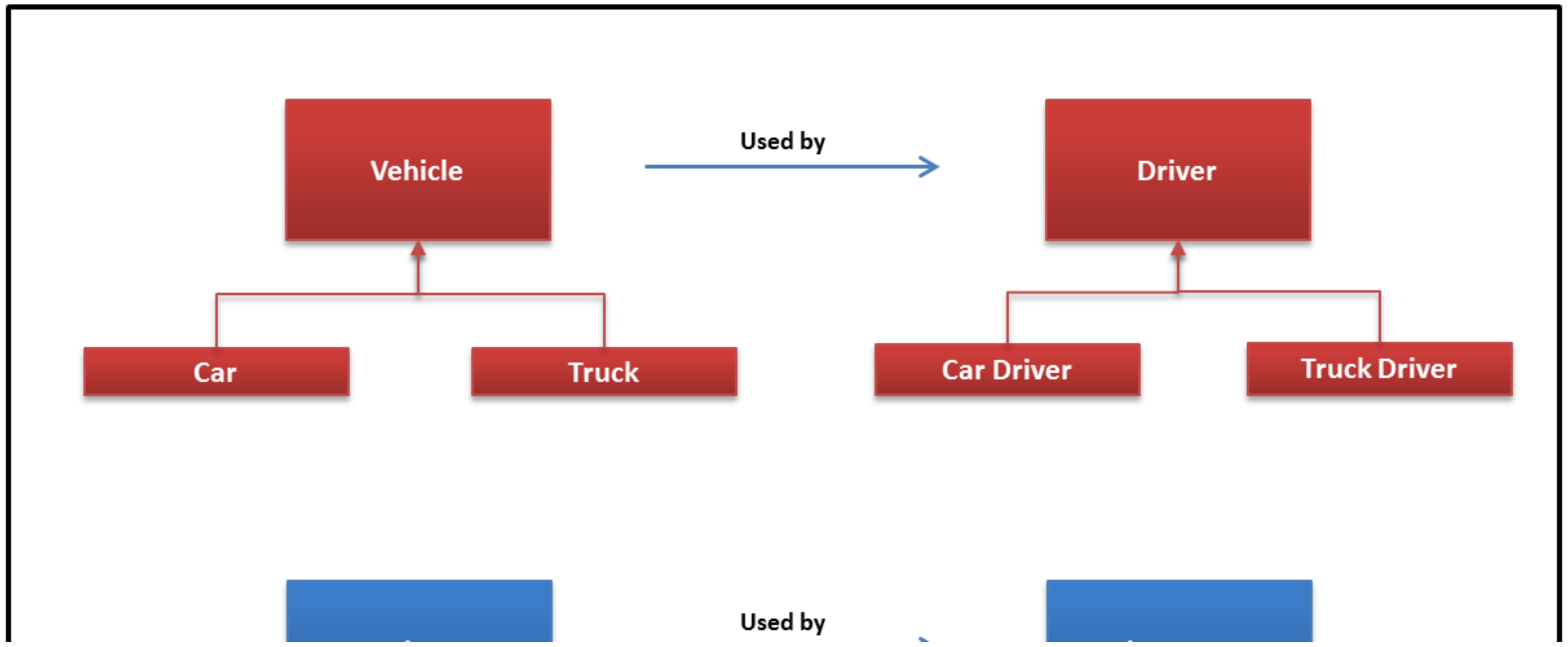
For every entity, a parallel DAO class needs to be created, hence a parallel hierarchy of classes appears in these cases. The DAO pattern is not the only case where they appear.

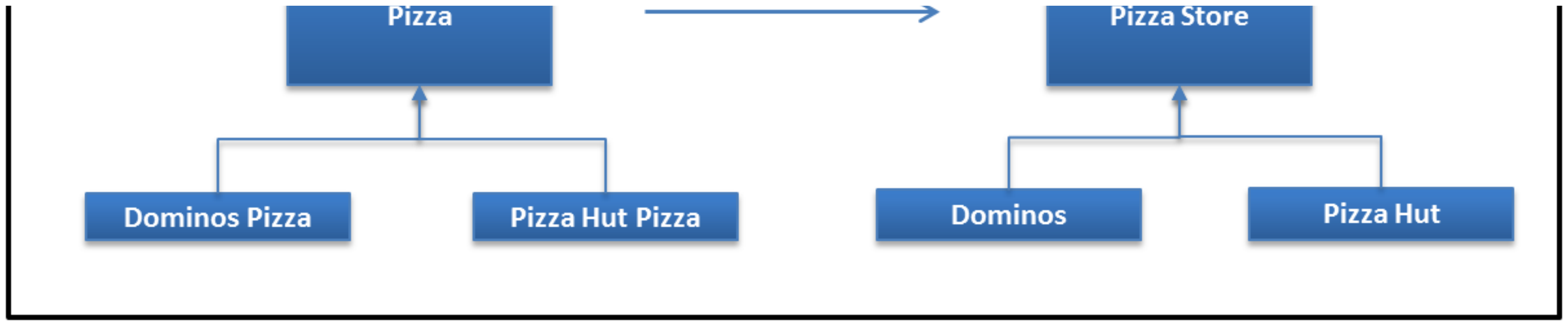
It usually occurs if we apply the Strategy Pattern to solve our business problem by decoupling the method from an object to supply many possible instances of the method.

Whenever we add a new class, we add a parallel Test case. If we require factories, we add a parallel factory class. Parallel hierarchies of classes occur in business use cases. Consider a new vehicle, say "Bus", is added to following vehicle hierarchy. In that case, we may need to add the "Bus Driver" class.

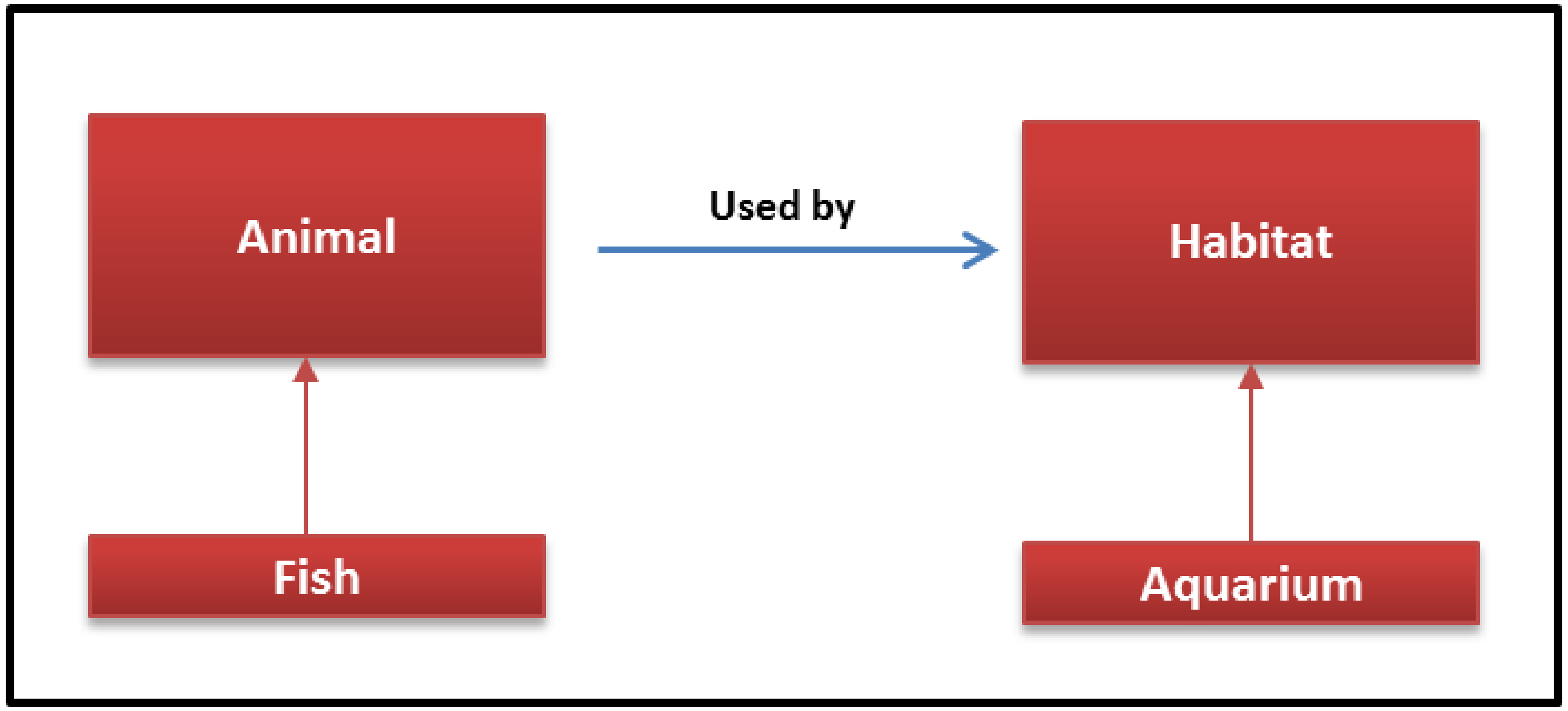



Child A
Child B
Child A Test
Child B Test
Base
Used by
Factory
Child A
Child B
Child A Factory
Child B Factory


Vehicle
Used by
Driver
Car
Truck
Car Driver
Truck Driver
Used by



Below is an example of a parallel hierarchy of classes and Generics.



```
import java.util.ArrayList;
import java.util.Collection;

public abstract class Habitat <A extends Animal> {

 //A generic collection that can hold Animal or any subclass of animal
```

```java
 Collection<A> collection = new ArrayList<A>();

 /*
  * add an Inhabitant to the collection. Should be overridden by subclass
  */
  public abstract  void addInhabitant( A animal);
}
```

```java
public class Aquarium extends Habitat < Fish > {

    @Override
    public void addInhabitant(Fish fish) {
        collection.add(fish);
        System.out.println(Aquarium.class);
    }
}
```

```java
/*
 * Super class for Animal hierarchy
 */
public abstract class Animal {

}
```

```java
public class Fish extends Animal {

}
```

```java
public class Test {
    public static void main(String[] args) {
        Animal animal = new Fish();
        Fish fish = new Fish();
        new Aquarium().addInhabitant(fish);
    }

}
```

Use Case Type 5: To Create Typesafe Heterogeneous Containers

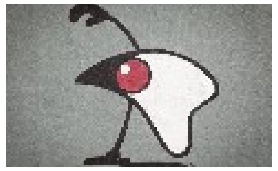
Collection<String> is an example of a homogeneous container. You cannot put anything other than a String into the box. Meanwhile, Collection<Object> is an example of a heterogeneous container. You can put any object into this box. Collection<Object> is not type safe. You need to check for type and cast it, similar to the raw type "Collection" (a raw type is a Generic type without a Generic type argument applied. It takes Object as the default type argument). Java does not provide first-class support for typesafe heterogeneous containers.

In Collection<String>, the type argument "String" is applied to type parameter "T" to make it type safe. Consider Map<String, String>. It takes two type arguments here. The normal use of Generics, exemplified by the Collection API, restricts you to a fixed number of type parameters per container. You can get around this restriction by placing the type parameter on the key of a Map, rather than the container. You can use Class objects as keys for building typesafe heterogeneous containers or maps.

Containers such as a bean creation container, exception handler containers, or service lookup containers are examples of heterogeneous containers where Generics can be used to make them typesafe with dynamic casting with class objects as keys.

I am not giving any code examples for heterogeneous containers, as a dedicated article will be published on them. You can also look for heterogeneous containers in Google in the meantime.

I hope the next time you think about Generics, data structures, boxes, Collections.class methods, parallel hierarchies of classes, and heterogeneous containers come to mind as well. If you think you know other use cases where Generics could be applied in general, I would be glad to hear from you.


5 Important Points about Java Generics

The Trouble With Enums

The Developer's Guide to Collections

Free DZone Refcard

Getting Started With Vaadin 10

Topics: JAVA GENERICS , JAVA , TYPE SAFE , DATA STRUCTURES , TUTORIAL

Java Partner Resources