**Java Generics Tutorial**

Generic style programming is a very power-full feature for a programming language. Lots of programming languages are compatible with this aspect, some of the well knowns are, C++, Phyton and of course Java. Java supported generics starting with JDK 1.5 version by the year 2005. Although it’s a seasoned concept, I still don’t think its maturely used in software projects so that is why I wan’t to write about this topic.

A picture containing graphical user interface

Description automatically generated

A very complete definition for generics style programming is mentioned in Wikipedia, let’s have a look:

“G*eneric programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameter”.*

Although it is a very precise definition, I must say, it is not the simplest one to start with, you may feel confused, please carry on to read the remaining.

This definition says, at its most basic form, that types (Classes and Interfaces) may be parameters to other Classes or Methods. When we write a class or a method , we tend to think and code as concrete as it could be. Generics comes into play to think and design in a broader perspective, to make our classes more usable, more extendable.

**So Why generics?**

Answer is hidden at the roots of java. As you know, Object class is the root of the all classes and the super class or ancestor of all the classes in java. So, initially , before generics were implemented in java, Object class has been used to refer other classes. That was a problematic approach.

Look at this code;

List aList = new ArrayList();aList.add(“zero”);aList.add(new Integer(0));String aNumber = (String) aList.get(0);

This is completely a runnable code. List members are added as Objects to the collection. But it is very open to future bugs in production. What happens when this list was referenced from else where and used without casting, being assumed that all the members are numbers. Then without any cast or casting to Numbers, runtime bugs may inevitably will raise.

The problem here is that compiler doesn’t know the intention of the programmer. ***What do the programmer intend to achieve is unclear!***That made me remember the Key-maker in Matrix movie☺



Matrix Key-Maker

It is best to predict programming errors to early stages of software life cycle. Compile time errors are more innocent and cute than runtime errors. Compile time errors are easier to solve and less risky. But How? I think, this is a matter of explicity and implicity. Define yourself more explicitly and your code too. Be the key maker!

If the solution is expressed in more explicit way, if the programmer tells the compiler about his/her intention , then this approach is possible. The compiler will handle the rest and detect situations that may be problematic at compile time.

Let’s refactor the code using generics:

List<Integer> aList = new ArrayList<Integer>();  
aList.add(new Integer(0));  
Integer zero = aList.get(0);

By using <Integer> we have expressed our intention that this list collection will consist of Integers. Our key here as key makers of the code are <>. Notice that, we didn’t use any type casting at last line, compiler knows it will return an Integer. Also adding another type except from Integer, will result in a compile time error:

Graphical user interface, text, application

Description automatically generated

**Usage scenarios:**

There are two kind of generics usage, using it by generic type parameter to class definitions or using it as type parameters to generic methods. We will dive in details using examples.

We will need an example model for diving in to the details, I will use a problem domain of modeling a house of animals which holds multiple Animal objects and let them to express some behaviors like sounding. And we will try to model this collection type first without generics and then using generic programming techniques.

So for our problem domain, here is a basic Animal class which will be extended later by Dog and Cat classes.

public abstract class Animal extends Creature implements Comparable<Animal>, Soundable { public static final double *FAV\_NUMBER* = 1.6180;  
 private static final int *MAX\_ANIMALS\_ALLOWED* = 1000000;  
 private static final String *COMMON\_NAME* = “Specy”; @Override  
 public int compareTo(Animal animal) {  
 return name.compareTo(animal.getName());  
 } @Getter @Setter  
 private String name;  
 @Getter @Setter  
 private int weight; *//-2³¹ to +2³¹* @Getter @Setter  
 private boolean hasOwner = false;  
 @Getter @Setter  
 private byte age; *//-127 to +128* @Getter @Setter  
 private long uniqueId;  
 @Getter @Setter  
 private double speed;*//64 bit decimal number* @Setter  
 private boolean hungry;…}

An Animal has some specific attributes like name, age, weight, hasOwner, etc. Animal implements Soundable interface to make a sound.

public interface Soundable {  
 public String sound();  
}

*An interface with only one method is called “Functional Interface” in functional programing. It is a gateway to a new programming paradigm, Lambda Expressions. That will be a subject of a new tutorial about lambda expressions.*

Back to our subject, A Dog class extended from Animal would be

public class Dog extends Animal{  
 public static final String *DOG\_SOUND* = “Bark”; public Dog(String name){  
 super(name);  
 } public Dog(String name, int weight, boolean hasOwner, byte age){  
 super(name, weight, hasOwner, age);  
 } @Override  
 public String sound() {  
 *//Local variables must be initialized before use* System.*out*.println(String.*format*(“%s : %s”, getName(), *DOG\_SOUND*));  
 return *DOG\_SOUND*;  
 }}

We are keeping it as simple as it could be, And a Cat class would be;

public class Cat extends Animal {  
 protected static final String *CAT\_SOUND* = “Mav”; public Cat(String name) {  
 super(name);  
 } public Cat(String name, int weight, boolean hasOwner, byte age) {  
 super(name, weight, hasOwner, age);  
 } @Override  
 public String sound() {  
 System.*out*.println(String.*format*(“%s : %s”, getName(), *CAT\_SOUND*));  
 return *CAT\_SOUND*;  
 } @Override  
 public int compareTo(Animal animal) {  
 return getName().compareTo(animal.getName());  
 }}

That model is enough for now, let’s get back to talking about **generics**.

**Generic types:**

First let’s model a house for Animal classes without using Generics. An Animal House class will have a name and have a collection of residents, may add new residents, get the count of the residents and so on.

public class BasicAnimalHouse { private final String name;  
 private List<Animal> animals; public BasicAnimalHouse(String name){  
 this.name = name;  
 this.animals = new ArrayList<>();  
 } public String getName() {  
 return name;  
 } public void add(Animal… args){  
 for(Animal animal:args){  
 animals.add(animal);  
 }  
 } public List<Animal> getAnimals() {  
 return animals;  
 } public int getNumberOfAnimals(){  
 return animals.size();  
 }}

This is a concrete definition for House class, it operates on concrete Animal class. Here, what you may observe would be the features for this House class are not specific to Animal, adding a member, getting one etc. If we may generalize this class, it may be used by other domain entities, not only for animals.

So how will we adopt use of generics on this class.

First a generic type definition will take place in the class definition to tell compiler that this class is parameterized by types.

Let’s give a name to the new class as GenericHouse;

public class GenericHouse<T> {…}

Then we will model the class data and behaviors using type parameter T.

public class GenericHouse<T> { private final String name;  
 private List<T> members; public GenericHouse(String name){  
 this.name = name;  
 this.members = new ArrayList<T>();  
 } public String getName() {  
 return name;  
 } public void add(T… args){  
 for(T member:args){  
 members.add(member);  
 }  
 } public List<T> getMembers() {  
 return members;  
 }}

What we have here is a more generic class that would be usable by not only for Animals but also for other entities. While coding the type argument, the programmer will specify the usage intention for this class. That is called type argument. Type argument is where we invoke the generic code;

GenericHouse<Animal> genericHouse = new GenericHouse<>(“generic animal house”);  
genericHouse.add(aCat,aDog);  
System.*out*.println(genericHouse);

We are explicitly saying to the compiler that this house will be used for animals.

**Using Inheritance with Generic types**

We have created a broader class that will serve to many domain entities. Let’s add new functionality to House class. Think about a method for sounding all the animals in the House, the members, one by one.

public class GenericHouse<T> { private final String name;  
 private List<T> members;public void soundAllAnimals() {  
 members.forEach(Animal::sound);  
}}

This will result in a compiler error, cause compiler doesn’t know anything about the parameterized type whether its and Animal or not. Only we specified it while invoking.

How will we solve it, This generic implementation is too broad, we will need focus to be more specific that members are Animals and Soundable. Solution is using bounded generic types and using inheritance.

We will extend a house from Generic House and bound it (make it narrower) to a specific type of Animal, we will still be able to use the behaviours of generic House from inheritance.

public class AnimalHouse<T extends Animal> extends GenericHouse<T>{ public AnimalHouse(String name){  
 super(name);  
 } *//now compiler knows family member is an animal* public void soundAllAnimals() {  
 super.getMembers().forEach(Animal::sound);  
 }}

Now compiler may call sound() for all members cause it knows members are Animal or subclasses of Animal like Dog or Cat.

**Bounded type parameter:**

<T extends Animal> means any subclass of Animal can be the member of this house. This is called bounded type parameter.

This is possible now,

AnimalHouse<Animal> animalHouse = new AnimalHouse<>(“extended generic tiny house”);  
animalHouse.add(aCat,aDog);  
animalHouse.soundAllAnimals();

Or we may be more specific, such as a house for only cats would be created;

AnimalHouse<Cat> catHouse = new AnimalHouse<>(“cat house”);  
catHouse.add(aCat);

**Generic Methods:**

A generic method is a method defined using type parameters as parameters. There is no direct relation to generic class, a generic method could take place in a concrete class or in a generic class , it doesn’t have to be inside a generic class.

To define a generic method, first we use type parameter before the return type of the method.

A sample method would take an array as parameter and returns a list of array members.

**public** <T> List<T> **fromArrayToList**(T[] a) {**return** Arrays.stream(a).collect(Collectors.toList());}

<T> should be declared before return type of method.

Bounded types are still usable for generic methods, if we want this method for only available to Numeric classes, specifying <T **extends** **Number**> would do the job.

**public** <T **extends** **Number**> List<T> **fromArrayToList**(T[] a){ …}

**Using wildcards:**

As we have mentioned <T> is used for generic type declaration, likewise sometimes “?” question mark is used for representing unknown types. But first let’s clarify the situation about when to use wildcards in code.

Wildcards came into play to define a relationship between extended collection objects. First of all, think about Integer and Number classes. Integer is a subtype of Number class. But what about List<Integer> and List<Number>.

As you may guess those two list are not related, the only relation is that both extends from Object.

Diagram

Description automatically generated

Diagram

Description automatically generated

I hear you asking So what? Let’s think simple, there should be a way we may create a sub type relation for such situations. Let’s consider this simple method which is a static utility method for our GenericHouse class, it will sound all the members, and the members are extended from Soundable interface.

public static void soundAllMembers(List< Soundable> list) {  
 for (Soundable elem : list) {  
 elem.sound();  
 }  
}

And here is our animal house we want to sound all members:

AnimalHouse<Animal> animalHouse = new AnimalHouse<>(“ahouse”);  
animalHouse.add(aCat,aDog);GenericHouse.soundAllMembers(animalHouse.getMembers())

Will last line compile?

Answer is no, List<Soundable> is not a super type of List<Animal> although Animal implements Soundable interface.

But, wildcards may represent this relation and fit in the solution. Let’s refactor that method using wildcard:

public static void soundAllMembers(List< ? extends Soundable> list) {  
 for (Soundable elem : list) {  
 elem.sound();  
 }  
}

<? Extends Soundable> means this method will work with subtypes of all Soundable interfaces.

GenericHouse.soundAllMembers(animalHouse.getMembers())

Will compile then successfully! Cheers☺ Here is the importance of wildcards, there are different usages that one is called upper bounded wildcard.

**Type erasure**

So lets’ remember the first definition which was a little bit complex to understand, it will fit in our mind after this examples.

“Generic programming is a style of [computer programming](https://en.wikipedia.org/wiki/Computer_programming) in which [algorithms](https://en.wikipedia.org/wiki/Algorithm) are written in terms of [types](https://en.wikipedia.org/wiki/Data_type) *to-be-specified-later* that are then *instantiated* when needed for specific types provided as [paramete](https://en.wikipedia.org/wiki/Parameter_(computer_programming))r.

So as you have seen, Generics were introduced for tighter compile time checking. But how is that implemented in java? The answer is compiler applies type erasure while compiling code. All the generic types coded such as <T>,<U> replaced with ordinary classes and interfaces, and with Object class. This is called type erasure.

This is the end of this tutorial, hope this is helpful, thanks for reading,

Be the KeyMaker, use generics☺