**Object Oriented Development**

Generics

Tutorial

© FDM Group Ltd 2020. All Rights Reserved.

Any unauthorised reproduction or distribution in part  
or in whole will constitute an infringement of copyright.

# What does this tutorial cover?

This tutorial will introduce and discuss generics, and how to use them in code.

# How long will the tutorial take to complete?

Around 2-3 hours

# What should you have already completed?

You should have completed the slides on collections, and inheritance

# What do you need?

In order to complete this tutorial you will need:

* Java Development Kit 1.8 or above
* Eclipse IDE Kepler or above

# What does this tutorial cover?

* What is a generic?
* How are generics used in Java code?
* How do you use a generic as a return value/parameter?
* How do you use generics with extends?
* What is a wildcard? What is the difference between a generic and a wildcard?
* How do you use wildcards with super/extends?

# What is a generic?

Collections are used to store a group of similar data in one location. But, how does a collection know what it is storing inside of itself? When we declare a collection, we have to declare its *type parameter*. For example, List<String> would be a List that has String as its type parameter. Thus, this List would only be able to store Strings or subclasses of Strings inside of it.

But, when you look at the List API on Oracle’s documentation ( <https://docs.oracle.com/javase/8/docs/api/java/util/List.html> ), you will see that instead of storing a class in between the arrowheads, it stores an E instead. This is a *generic*, which will be replaced by the reference type you place in between the arrowheads at runtime.

# How are generics used in code?

The code below creates and implements a List using a String class as its type parameter. If you look at the API, you will see the get(index i) method returns a type E. But, when we call it in the code, it will return a type of String.

List<String> names = **new** ArrayList<String>();

names.add("Bob"); // In API, it is add(E element);

String name = names.get(0); // In API, it is get(int index) and returns E

System.***out***.println(name); // Bob

To declare a class to use a generic type parameter, it must be done at the class declaration. For this example, create a new class called Farmer, and after the class name, place <P> after the name of the class. (The P is our generic element, which is short for produce. We want our farmer to be able to farm a variety of produce).

**public** **class** Farmer<P> {}

The P can be replaced with any other letter or phrase. We chose P because it is common practice to use single letters as generic in order to denote them as generics. No one will confuse a single letter for an actual class.

Now, we will need some way to store the produce our farmer will be farming. Let’s use a List. Instead of our list storing a class, it will store our generic, like so:

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Farmer<P> {

**private** List<P> produceItems = = **new** ArrayList<P>();

}

When you initialize the farmer, you will also be initializing the type parameter of the list.

## Returning a Generic & Using a Generic as a Parameter

Now we need ways to access, and add the items of the produceItems list. First, let’s return an item from the list based on the index. We can do that by creating a method that returns the type parameter.

**public** P getProduce(**int** index) {

**return** produceItems.get(index);

}

Just like in the List class, the P here will be replaced by the class that we declare for our Farmer to produce.

To use the generic as a parameter can be done in a similar manner. The class will need some way to add in items that the farmer can produce, so it will need an addProduce method. The method declaration would be as follows:

**public** **void** addProduce(P produce) { produceItems.add(produce);

}

So far, the overall class should look like this:

**public** **class** Farmer<P> {

**private** List<P> produceItems = **new** ArrayList<P>();

**public** P getProduce(**int** index) {

**return** produceItems.get(index);

}

**public** **void** addProduce(P produce) {

produceItems.add(produce);

}

}

Create a second class called Client. In the Client class, create a main method which will create and initialize a farmer. (For the time being, the Farmer can be ‘producing’ any class in Java.) Add an item to its produceItems list, and then return said item. The code should look like this in the end:

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Farmer<Double> numberFarmer = **new** Farmer<Double>();

numberFarmer.addProduce(30.0);

System.***out***.println("The farmer is farming: " + numberFarmer.getProduce(0));

}

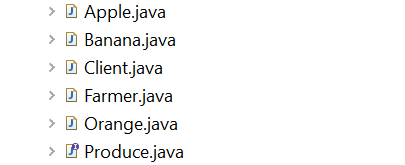
}

# Generics and extends

The problem with our Farmer code is that there is no way for the class to control what classes should and should not be stored into its list. The class should only be concerned with produce, after all. But, we can use the extends keyword on the type parameter to declare it will only accept the base level or subclasses of the base level.

Create a new interface called Produce, and then create three concrete classes that will implement Produce. For the purposes of this tutorial, we will create an Apple, Orange, and Banana class.

Your collection of classes so far should look like this:



In the Farmer class, have the type parameter extend Produce. While the Produce is an interface, the generic element cannot implement an interface, only extend. Generic elements can extend classes as well, whether they are abstract or not.

**public** **class** Farmer<P **extends** Produce> {

Once you do this, the code in the Client class will not compile. That is to be expected. After all, the Farmer cannot farm Doubles, as Doubles are not Produce (or a subclass of Produce). Change the Double declaration in your Client class to be type Produce instead, and add in an instance of each of your subclasses you have declared.

**public** **static** **void** main(String[] args) {

Farmer<Produce> numberFarmer = **new** Farmer<Produce>();

numberFarmer.addProduce(**new** Banana());

numberFarmer.addProduce(**new** Orange());

numberFarmer.addProduce(**new** Apple());

System.***out***.println("The farmer is farming: " + numberFarmer.getProduce(0)); //prints out "The farmer is farming: Banana@7852e922"

}

Now the Farmer class can be sure that the items that will be added to its collection are of type Produce, and thus can do Produce-specific methods.

**Exercise: Declare a method in the Produce interface called getSpecies(). Implement the method in all of the classes that implement Produce. Set the species of the produce via a setter or constructor. Create a method called printAllSpecies() in the Farmer class that will loop through all of the Produce items in its collection, and print out their species.**

**Solution:**

**Produce.java:**

**public** **interface** Produce {

String getSpecies();

}

**Banana.java:**

**public** **class** Banana **implements** Produce {

**private** String species;

**public** Banana(String species)

{

**this**.species = species;

}

@Override

**public** String getSpecies() {

**return** species;

}

}

**Farmer.java:**

**public** **class** Farmer<P **extends** Produce> {

**private** List<P> produceItems = **new** ArrayList<P>();

**public** P getProduce(**int** index) {

**return** produceItems.get(index);

}

**public** **void** addProduce(P produce) {

produceItems.add(produce);

}

**public** **void** printAllSpecies() {

**for** (Produce produce : produceItems) {

System.***out***.println(produce.getSpecies());

}

}

}

# Wildcards

Wildcards allow one to use generic-specific behaviours without having to declare the behaviour as the class level. It can be applied to variables and methods.

For this next example we will create a Market class. The intended purpose of said class is to set the prices that the Farmers will sell their produce for. The produce items will be matched to their price via a Map.

**import** java.util.Map;

**public** **class** Market {

**private** Map<Produce, WhatGoesHere> producePrices;

}

The problem arises when we want to declare a value class for the Map. If we declare it as a Double, for example, we cannot put in Integers. What we can do is put a wildcard, like so.

**private** Map<Produce, ?> producePrices;

The wildcard in Java is represented by a question mark. You ***cannot*** use a letter or phrase like you did with generics, as it will be interpreted as a class. That is because we have not declared at the class level that this class will be using generics. Notice the lack of arrowheads at the top of the class.

With that in mind, we will need a way to initialize the wildcard. Let’s use a constructor.

**public** Market(Map<Produce, ?> producePrices) {

**this**.producePrices = producePrices;

}

Just like the generic, the wildcard can be used as part of a parameter in a method or a constructor’s declaration. ***However***, the wildcard cannot be used as part of the initialization.

Market market = **new** Market(**new** HashMap<Produce,?>()); // DOES NOT COMPILE- does not know what the ? is.

Instead, the code needs to insert an actual class in place of the question mark. This is where we can declare that it will be using Integers, Doubles, Floats, etc.

Market market = **new** Market(**new** HashMap<Produce, Integer>());

## What is the difference between Wildcards and Generics?

Remember, the main difference between wildcards and generics is that generics are declared at the top with the class alongside the class name, while wildcards are declared at the variable/method level. Unlike generics, wildcards ***CANNOT*** be used as a return type.

## Bounded Wildcards

Right now, the Market can have any class declared for the price. Which means that it is possible for you to declare that the value of the produce items to be LocalDate. In order to prevent such issues, we can apply the extends keyword to the wildcard just like we did with the generic.

**private** Map<Produce, ? **extends** Number> producePrices;

**public** Market(Map<Produce, ? **extends** Number> producePrices) {

**this**.producePrices = producePrices;

}

In this code, we have declared that the producePrices must be a Number or a subclass of Number. Just like with generics, we extend interfaces, and not implement them for wildcards. This kind of wildcard is known as a *bounded wildcard*. It is has an ‘upper limit’ in terms of the highest you can go up the class hierarchy. It will not allow you to declare the price to be Object, for example, since that is above Number.

In the Client class, let’s initialize a new market for apples.

**public** **static** **void** main(String[] args) {

Market appleMarket = **new** Market(**new** HashMap<Apple, Integer>());

}

This code does ***NOT*** compile. This is due to type declarations not working in a polymorphic manner (i.e., it will only accept a Produce declaration on the right side, and not any subclasses of Produce). But, using the power of bounded wildcards, we can declare that the Market should accept any subclass of Produce via extends.

Edit your code in the Market class to instead say:

**private** Map<? **extends** Produce, ? **extends** Number> producePrices;

**public** Market(Map<? **extends** Produce, ? **extends** Number> producePrices) {

**this**.producePrices = producePrices;

}

You can have a ‘lower limit’ as well by using the super keyword. The super keyword tells the wildcard not to accept anything that is below the class you declared.

**private** Map<? **extends** Produce, ? **super** BigInteger> producePrices;

**public** Market(Map<? **extends** Produce, ? **super BigInteger**> producePrices) {

**this**.producePrices = producePrices;

}

# Conclusion

Generics are essential for collections. Without them, we would not be able to use collections without a ton of type checks. Understanding generics is a key aspect in creating your own collection and storage classes.