
ORACLE
Academy

Java Programming

3-1

Generics

ORACLE
Academy

Overview

- This lesson covers the following topics:
  - Use enumerated types
  - Create a custom generic class
  - Use the type interface diamond to create an object
  - Use generic methods
  - Use wildcards

Enumerations

- Enumerations (or enums) are a specification for a class where all instances of the class are created within the class
  - Enums are a datatype that contains a fixed set of constants
  - Enums are good to use when you already know all possibilities of the values or instances of the class
  - If you use enums instead of strings or integers you increase the checks at compile time
- Can be declared on their own or within another class

Enumerations BankExample

- Say we wish to store the type of bank account within our Account Class
- We could have Savings and Credit as possible options
- As long as we specify that the class is of type enum, we can create these account types inside the class itself as if each was created outside of the class

Enums can make your code a lot more readable.
Example - Rather than access a seemingly random number in your code you could access a name that would add meaning to what you are trying to achieve.

Adding Enumerations to the JavaBank

1. Add the following enum class to the JavaBank project

```
public enum AccountType {
   SAVINGS,
   CREDIT
}//end enum AccountType
```

These are the initializations of all the Account Types. Constants are always written in uppercase.

2. In the Account class create a private instance variable that will hold an account type under the bonusValue field

```
private AccountType type;
```

Adding Enumerations to the JavaBank

3. Update the Account constructor to take a fourth parameter of type AccountType named type

```
Account(String name, int num, int amt, AccountType type)
```

4. Assign the value of the instance field "type" to that of the parameter "type". Must come after the super call

```
//constructor for Account
Account(String name, int num, int amt, AccountType type)
{
   super(name, num, (amt + calculateInitialBonusValue(amt)));
   bonusValue = calculateInitialBonusValue(amt);
   this.type = type;
}//end constructor method
```

Adding Enumerations to the JavaBank

5. Create a toString() method in the Account class that will display the account type before calling the super class' toString() method

```
@Override
public String toString() {
   return "\nAccount Type   : " + this.type +
          super.toString();
}//end method toString
```

Now when you try to display the values of an Account objects instance fields to screen it will call this toString() method instead of the one in the AbstractBankAccount class!

Adding Enumerations to the JavaBank

6. Open the TestBank class
7. Update the constructor calls to include an enum value that sets the account type to savings

```
// Using constructor with values
Account a1 = new Account("Sanjay Gupta",11556,300, AccountType.SAVINGS);
Account a2 = new Account("He Xai",22338,500, AccountType.SAVINGS);
Account a3 = new Account("Ilya Mustafana",44559,1000, AccountType.SAVINGS);
```

8. Change the print statements to console output statements that will show the values of the objects

```
// Print accounts to the console
System.out.println(a1);
System.out.println(a2);
System.out.println(a3);
```

Enumeration Task

- **Quick Task:** Updating the constructor calls

a) Open the TestCreditAccount class and update any Account objects constructor calls to use the enum to create a savings account

b) Open the TestCustomerAccount class and update any Account objects constructor calls to use the enum to create a savings account

c) You will update the JavaBank application in the practical exercises for this lesson!

Enumeration Task Suggested Solution

- **Quick Task:** Updating the constructor calls.

a) TestCreditAccount

```
AbstractBankAccount a1 = new Account("Sanjay Gupta",11556,300,
AccountType.SAVINGS);
AbstractBankAccount a2 = new Account("He Xai",22338,500,
AccountType.SAVINGS);
AbstractBankAccount a3 = new Account("Ilya Mustafana",44559,1000,
AccountType.SAVINGS);
```

b) TestCustomerAccount

```
bankAccount[0] = new Account("Sanjay Gupta",11556,300,
AccountType.SAVINGS);
bankAccount[1] = new Account("He Xai",22338,500, AccountType.SAVINGS);
bankAccount[2] = new Account("Ilya Mustafana",44559,1000,
AccountType.SAVINGS);
```

Enumerations Iteration

- We could print out our enums by using a for loop

```
for (AccountType act : AccountType.values())
   System.out.println("Value: " +  act.name()
                   + ", position: " + act.ordinal());
//endfor
```

- Would produce:

```
Value: SAVINGS, position: 0
Value: CREDIT, position: 1
```

Enums have a number of inbuilt methods, this example shows the name and ordinal methods.

Enumeration Objects

- Enums are like any other class and can have fields, constructors and methods
- By default, enums don't require a constructor definition, their default values are always the string used in the declaration
- You can add additional attributes to the enum fields
- You can define your own constructors to initialize the state of enum types
- Objects can be instantiated from an enum class by going through its constructor

Enumerations AccountType

- Our bank accounts could have an internal code that is used by the bank to identify the account types

```
public enum AccountType {
   CURRENT("CU"),
   SAVINGS("SA"),
   DEPOSIT("DP");

   private String code;

   private AccountType(String code){
      this.code=code;
   }//end method AccountType

   public String getCode() {
      return code;
   }//end method getCode
}//end class AccountType
```

Enum fields with attributes (Each account type has its own code).

Enum constructors must be either private or default

Both abstract and concrete methods are allowed in enums.

Enumerations AccountType

- We can have more than one field in the enum
- If we wanted a minimum percent rate we could have:

```
public enum AccountType {
   CURRENT("CU", 1.0),
   SAVINGS("SA", 2.0),
   DEPOSIT("DP", 0.0);

   private String code;
   private double rate;

   private AccountType(String code, double rate){
      this.code = code;
      this.rate = rate;
   }//end constructor method
```

Each field now has 2 attributes

A field must exist for every attribute

The attribute values are set through the constructor. These are not passed but assigned by the enum type

- Another internal field was included with another parameter added to the constructor to set the value

Enumerations AccountType

- By adding additional getters the value of the fields can be returned

```
public String getCode() {
   return code;
}//end method getCode

public double getRate() {
   return rate;
}//end method getRate
```

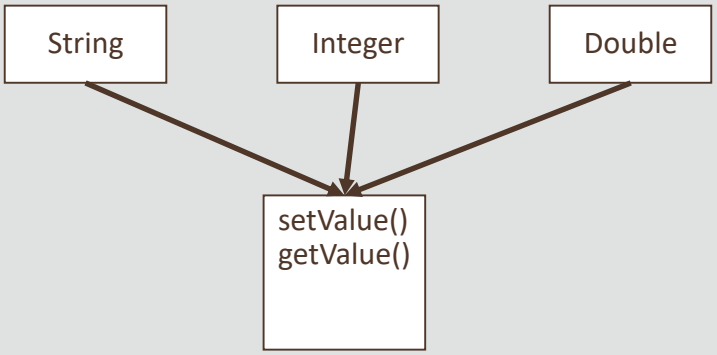
- We can now access the field values held by the enum

```
AccountType account = AccountType.Deposit;
System.out.println("Type: " + account.name()
                +"\nCode: " + account.getCode()
                +"\nRate: " + account.getRate());
```

Output: `Type: Deposit, Code: DP, Rate: 0.0`

Problem with types

- Often in programming we want to write code which can be used by more than one type with the same underlying behavior



We don't want to have to repeat our code line by line only changing what data types it can access.

Simple Class Example

- If we wanted a very simple class to get and set a string value we could define this as:

```
public class Cell {

   private String data;

   public void setValue(String celldata) {
      data = celldata;
   }//end method setValue

   public String getValue() {
      return data;
   }//end method get

}//end class Cell
```

We have one private field with a mutator and accessor.

Simple Driver Class

- Using a simple driver class we could set and retrieve a string value

```
public class CellDriver {

   public static void main(String[] args) {
      Cell cell = new Cell();
      cell.setValue("One");
      System.out.println(cell.getValue());
   }//end method main
}//end class CellDriver
```

- This example would only ever work with Strings
- Although this is a very simple class without much coding, if it had been more complex we may wish to reuse the algorithms with other data types

Flexible Class

- We could change the String primitive type to Object

```
public class Cell {
   private Object data;

   public void setValue(Object celldata) {
      data = celldata;
   }//end method setValue

   public Object getValue() {
      return data;
   }//end method get
}//end class Cell
```

- Changing all occurrences of String to Object would give us the flexibility to use other datatypes

Flexible Driver Class

- Now our driver class can set the type of data we wish to store

```
public class CellDriver {

   public static void main(String[] args) {
      Cell cell = new Cell();
      cell.setValue(1);
      int num = (int)cell.getValue();
      System.out.println(num);
   }//end method main

}//end class CellDriver
```

- Our cell cast can be used on different types, but only one type at a time. If we cast to the wrong type then we will most likely produce a casting exception

Flexible Driver Class

- The problem with this is if we pass a String in the set method and try to cast as int then we will receive a casting error at runtime

```
public class CellDriver {

   public static void main(String[] args) {
      Cell cell = new Cell();
      cell.setValue("One");
      int num = (int)cell.getValue();
      System.out.println(num);
   }//end method main

}//end class CellDriver
```

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
at genericsexercise.CellDriver.main(CellDriver.java:8)
```

Generic Classes

- A generic class is a special type of class that associates one or more non-specified Java types upon instantiation
- This removes the risk of the runtime exception "ClassCastException" when casting between different types
- Generic types are declared by using angled brackets - <> around a holder return type. E.g. <E>

<E> is simply an example. We could have written <T> , <Type1> or <Element1>

Generic Cell Class

- We can modify our Cell class to make it generic

```
public class Cell<T> {
   private T data;

   public void setValue(T celldata)
   {
      data = celldata;
   }//end method setValue

   public T getValue() {
      return data;
   }//end method get

}//end class Cell
```

- We have not had to make many changes to make our class generic
- We remove specific data type references and change them to the generic type T

Generic Cell Driver Class

- We can now set the type at creation

```
public class CellDriver {

   public static void main(String[] args) {
      Cell<Integer> integerCell = new Cell<Integer>();
      Cell<String> stringCell = new Cell<String>();
      integerCell.setValue(1);
      stringCell.setValue("One");
      int num = integerCell.getValue();
      String str = stringCell.getValue();
      System.out.println("Integer Value: " + num);
      System.out.println("String Value : " + str);
   }//end method main
}//end class CellDriver
```

Now java knows what type each Cell instance should store, it can do a type check at compile time.

Initializing a Generic Object

- Initializing a Generic object with one type, Example:

```
public class Cell<T> {
   private T data;

   public void setValue(T celldata)
   {
      data = celldata;
   }//end method setValue

   public T getValue() {
      return data;
   }//end method get

}//end class Cell
```

```
Cell<Integer> integerCell = new Cell<Integer>();
```

Initializing a Generic Object

- You define the generic class name followed by the type in the diamond brackets and then give the object a name

```
ClassName<Data Type> ObjectName = new ClassName<Data Type>();
```

- The data type on the left of the new operator is optional

```
ClassName<Data Type> ObjectName = new ClassName();
```

Both ways of declaring a generic object are correct.

Initializing a Generic Object

- To initialize a Generic object with two types:

```
ClassName<Data Type, Data Type> ObjectName = new ClassName();
```

- Example

```
Example<String, Integer> showMe = new Example<>();
```

- The only difference between creating an object from a regular class versus a generics class is the diamond brackets<String, Integer>

This is how to tell the Example class what type of data types are to be used with that particular object.

Updated Cell class with two types

```
public class Cell<T, T2> {
   private T t;
   private T2 k;

   public void setValue(T celldata, T2 i) {
      t = celldata;
      k = i;
   }//end method setValue

   public T getT1Value() {
      return t;
   }//end method getT1Value

   public T2 getT2Value() {
      return k;
   }//end method getT2Value

   public String toString(){
      return("cell type is: Type1: " + t.getClass() + " and  Type2: "
             + k.getClass());
   }//end method toString
}//end class Cell
```

Updated CellDriver class using two types

```
public class CellDriver {
   public static void main(String[] args) {
     Cell<Integer, String> mixCell = new Cell<Integer, String>();
     Cell<Integer, Integer> integerCell = new Cell();

     mixCell.setValue(1, "4");
     integerCell.setValue(45, 60);

     int mcType1 = mixCell.getT1Value();
     String mcType2 = mixCell.getT2Value();

     int icType1 = integerCell.getT1Value();
     int icType2 = integerCell.getT2Value();

     System.out.println(mixCell);
     System.out.println(integerCell);
       System.out.println("The numerical value is: " +
                          mcType1 + ". The text value is: " + mcType2);
     System.out.println("The first numerical value is: " + icType1 +
                        " and the second is : " + icType2);
   }//end main
}//end class CellDriver
```

Initializing a Generic Object

- For the mixCell object, Type1 is an Integer type, and Type2 is a String type
- For the integerCell object, both types are Integers

```
Cell<Integer, String> mixCell = new Cell<Integer, String>();
Cell<Integer, Integer> integerCell = new Cell();
```

- The benefit to a generic class is that you can identify multiple objects of type
- Using the Cell class we could initialize another object with <Double, String> types

Java will know the type at compile time, so this will help remove invalid casting.

Type Parameter Names

- The most commonly used type parameter names are:
  - E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types

There is no rule to what parameter name you use, but consistency helps readability.

Working with Generic Types

- When working with generic types, remember the following:
  - The types must be identified at the instantiation of the class
  - Your class should contain methods that set the types inside the class to the types passed into the class upon creating an object of the class
  - One way to look at generic classes is by understanding what is happening behind the code

Generic Classes Code Example

- This code can be interpreted as a class that creates two objects, Type1 and Type2
  - Type1 and Type2 are not the type of objects required to be passed in upon initializing an object
  - They are simply placeholders, or variable names, for the actual type that is to be passed in

```
public class Example<Type1, Type2>{
  private Type1 t1;
  private Type2 t2;
     …
}//end class Example
```

Generic Classes Code Example

- These placeholders allow for the class to include any Java type; they become whatever type is initially used at the object creation
- Inside of the generic class, when you create an object of Type1 or Type2, you are actually creating objects of the types initialized when an Example object is created

```
public class Example<Type1, Type2>{
  private Type1 t1;
  private Type2 t2;
     …
}//end class Example
```

Generic Methods

- So far we have created Generic classes, but we can also create generic methods outside of a generic class
- Just like type declarations, method declarations can be generic—that is, parameterized by one or more type parameters
- A type interface diamond is used to create a generic method

The type interface diamond is “<>”.

Type Interface Diamond

- A type interface diamond enables you to create a generic method as you would an ordinary method, without specifying a type between angle brackets
- Why a diamond?
  - The angle brackets are often referred to as the diamond <>
  - Typically if there is only one type inside the diamond, we use <T> where T stands for Type
  - For two types we would have <K,T>

Type Interface Diamond

- You can use any non reserved word as the type holder instead of using <T>. We could have used <T1>
- By convention, type parameter names are single, uppercase letters
- This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name

Generic Methods Example

1. Create a genericmethodexample project.
2. Create the following code that defines a generic method printArray for returning the contents of an array:

```
public class GenericMethodClass {

   public <T> void printArray(T[] array){
      for( T arrayitem : array ){
         System.out.println( arrayitem );
      }//endfor
   }//end method printArray

}//end class GenericMethodClass
```

The <T> tells the compiler that the method is generic and the type of T will be declared later.

Generic Methods

3. This would allow the printing of multiple array types

```
public class GenericMethodDriver {
   public void main(String[] args) {
      GenericMethodClass gmc = new GenericMethodClass();

      Integer[] integerArray = {1, 2, 3};
      String[] stringArray = {"This","is","fun"};

      gmc.printArray(integerArray);
      gmc.printArray(stringArray);
   }//end method main
}//end class GenericMethodDriver
```

- Output

```
1
2
3
This
is
fun
```

Passing a **non-array** type would cause problems for your generic method!

Generic Wildcards

- Wildcards with generics allows us greater control of the types we use
- They fall into two categories:
  - Unbounded
    - <?>
  - Bounded
    - <? extends type>
    - <? super type>

In the previous example where we only ever wanted to iterate through an array we would only have wanted to limit this to the array type.

Unbounded Wildcards

- <?> denotes an unbounded wildcard, It can be used to represent any type
- Example – arrayList<?> represents an arrayList of unknown type

```
ArrayList<?> array1 = new ArrayList<Integer>();
array1 = new ArrayList<Double>();
```

- You cannot add to array1 as the type is unknown

Two scenarios where this can be useful are:

1. If you are writing a method that can be implemented using functionality provided in the Object class.
2. When the code is using methods in the generic class that don't depend on the type parameter e.g. List.size or List.clear

Unbounded Wildcards

4. Create a method called printList in the GenericMethodClass class
5. Its goal is to print an arrayList of any type

```
   public void printList(List<?> list) {
      for (Object elem: list)
         System.out.println(elem);
      //endfor
      System.out.println();
   }//end method printList
}//end class GenericMethodClass
```

6. You will have to import the List library

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

Unbounded Wildcards

- public void printList(List<?> list)
  - This declares a list of unknown type as the parameter
- You could have written a generic method to achieve the same result using the Type assigned in the generic class
- public <T> void printList(List<T> list)

In our simple example the wildcard is slightly better as it identifies that the type is not used elsewhere. It is simply a method to iterate through a list of unknown type.

Unbounded Wildcards

7. Update the GenericMethodDriver to create two arraylists, populate the lists and then call the printList method(import the ArrayList library)

```
        ArrayList<Double> array1 = new ArrayList<Double>();
        array1.add(25.5);
        array1.add(34.9);
        array1.add(45.7);
        gmc.printList(array1);

        ArrayList<Integer> array2 = new ArrayList<Integer>();
        array2.add(10);
        array2.add(20);
        array2.add(30);
        gmc.printList(array2);

    }//end method main
}//end class GenericMethodDriver
```

Upper Bounded Wildcard

- <? extends Type> denotes an Upper Bounded Wildcard
- Sometimes we want to relax restrictions on a variable
- Lets say we wished to create a method that works only on ArrayLists of numbers
  - ArrayList<Integer>
  - ArrayList<Double>
  - ArrayList<Float>
- We could use an upper bounded wildcard

Upper Bounded Wildcard

8. Create the following method in the GenericMethodClass that will return a double value:

```
public double sumOfList(ArrayList<? extends Number> arrayList) {
    double s = 0.0;
    for (Number n : arrayList)
        s += n.doubleValue();
    //endfor
    return s;
}//end method sumOfList
```

9. Update the GenericMethodDriver to include two output statements that call the sumOfList() method

```
      System.out.println(gmc.sumOfList(array1));
      System.out.println((int) gmc.sumOfList(array1));
   }//end method main
}//end class GenericMethodDriver
```

Upper Bounded Wildcard

- The upper bound wildcard sets the type by extending subtypes of a specific supertype, in this case Number
- public double sumOfList(ArrayList<? extends Number> arrayList) {
- This only allows numeric arrayLists to be processed

```
        ArrayList<String> sArray = new ArrayList<String>();
        System.out.println(gmc.sumOfList(sArray));
    }//end method main
}//end class GenericMethodDriver
```

- Accesing the method with a String List would produce:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method sumOfList(ArrayList<? extends Number>) in the type
GenericMethodClass is not applicable for the arguments (ArrayList<String>)
at genericmethodexample.GenericMethodDriver.main(GenericMethodDriver.java:36)
```

Lower Bounded Wildcard

- <? super Type> denotes a Lower Bounded Wildcard
  - A lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type
- Say a method is required that puts Integer objects into an ArrayList
  - To maximize flexibility, you would like the method to work on ArrayList<Integer>, ArrayList<Number>, and ArrayList<Object> — anything that can hold Integer values

A Lower Bounded Wildcard is the opposite of an Upper Bounded Wildcard

Lower Bounded Wildcard

10. Create the following method in the GenericMethodClass:

```
public void addNumbers(ArrayList<? super Integer> arrayList) {
    for (int i = 1; i <= 10; i++) {
        arrayList.add(i);
    }//endfor
}//end method addNumbers
```

11. Update the GenericMethodDriver to include the following code that will use addNumbers()

```
      ArrayList<Integer> intArray = new ArrayList<Integer>();
      gmc.addNumbers(intArray);
      gmc.printList(intArray);
   }//end method main
}//end class GenericMethodDriver
```

Lower Bounded Wildcard

- When the generic method addNumbers was created it extended number. It would not accept an arrayList of Strings
- The addNumbers only accepts Integer types (and its superclasses) so if the following code was tried:

```
        ArrayList<Double> doubleArray = new ArrayList<Double>();
        gmc.addNumbers(doubleArray);
    }//end method main
}//end class GenericMethodDriver
```

- As it's a Double it would produce the following error:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method addNumbers(ArrayList<? super Integer>) in the type
GenericMethodClass is not applicable for the arguments (ArrayList<Double>)
at genericmethodexample.GenericMethodDriver.main(GenericMethodDriver.java:36)
```

Terminology

- Key terms used in this lesson included:
  - Generic Class
  - Type Interface Diamond
  - Use enumerated types
  - Use generic methods
  - Use wildcards

Summary

- In this lesson, you should have learned how to:
  - Use enums
  - Create a custom generic class
  - Use the type interface diamond to create an object
  - Use generic methods
  - Use bounded and unbounded wildcards


ORACLE
Academy