ORACLE® ACADEMY

Java Programming

2-3
Generics

ORACLE ACADEMY

Overview

This lesson covers the following topics:

- Create a custom generic class
- Use the type interface diamond to create an object
- Use generic methods
- Use wildcards
- Use enumerated types

Problem

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

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 set(String celldata)
  {
    data = celldata;
  }
  public String get() {
    return data;
  }
}
```

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.set("Test");
    System.out.println(cell.get());
  }
}
```

- 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 set(Object celldata)
  {
    data = celldata;
  }
  public Object get() {
  return data;
 }
}
```

- This would then 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.set(1);
    int num = (int)cell.get();
    System.out.println(num);
  }
}
```

- 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.

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>

Generic Cell Class

- We can modify our Cell class to make it generic.

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

  public void set(T celldata)
  {
     t = celldata;
  }
  public T get() {
     return t;
  }
}
```

Generic Cell Driver Class

- We can now set the type at creation.

```java
public class CellDriver {
  public static void main(String[] args) {
    Cell<Integer> integerCell = new Cell<Integer>();
    Cell<String> stringCell = new Cell<String>();
    integerCell.set(1);
    stringCell.set("Test");
    int num = integerCell.get();
    String str = stringCell.get();
  }
}
```

Initializing a Generic Object

- How to initialize a Generic object with one type, Example:

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

- With two types:

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

- The only difference between creating an object from a regular class versus a generics class is <String, Integer>.
- This is how to tell the Example class what type of types you are using with that particular object.

Initializing a Generic Object

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

- In other words, Type1 is a String type, and Type2 is an Integer type.
- The benefit to having a generic class is that you can identify multiple objects of type Example with different types given for each one, so we could initialize another object Example with <Double, String>.

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

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;
…}
```

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;
…}
```

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.

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

- To define a generic method printArray for returning the contents of an array we would declare it as.

```
public class GenericMethodClass {
   public static <T> void printArray(T[] array){
      for ( T arrayitem : array ){
            System.out.println( arrayitem );
      }
   }
```

Generic Methods

- This would allow printing of multiple array types.

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

printArray( integerArray );
printArray( stringArray );
```

- Output

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

Generic Wildcards

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

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>();
```

Unbounded Wildcards

- We are going to create a method called printArrayList. Its goal is to print an arrayList of any type.

```
public static void printList(List<?> list) {
  for (Object elem: list)
     System.out.println(elem);
  System.out.println();
}
```

- We could then pass any type of arrayList.

```
ArrayList<Integer> li = new ArrayList<Integer>();
li.add(1);
li.add(2);
ArrayList<String>  ls =  new ArrayList<String>();
ls.add("one");
ls.add("two");
printList(li);
printList(ls);
```

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:

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

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 you want to write a method 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.

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

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.

Enumerations BankExample

- For example, say we wish to store the type of bank account within our Account Class.
- We could have Current, Savings, and Deposit 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.

Enumerations Bank Code Simple Example

This keyword enum initializes the class AccountType as an enum type.

```
public enum AccountType {
          Current,
          Savings,
          Deposit
}
```

These are the initializations of all the Account Types

- We could assign any one of these to a field in our class.

```
AccountType type = AccountType.Deposit;
```

ORACLE ACADEMY

Enumerations Iterate

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

```
for (AccountType at : AccountType.values())
   System.out.println(at+", Value: "+at.name()+", ord:"+ at.ordinal());
```

- Would produce:

```
Current, Value: Current, ord:0
Savings, Value: Savings, ord:1
Deposit, Value: Deposit, ord:2
```

Enumerations AccountType

- Our bank account type might also have an internal code that is used by the bank.

```
public enum AccountType {
  Current("CU"),
  Savings("SA"),
  Deposit("DP");

  private String code;

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

  public String getCode() {
     return code;
  }
}
```

Constructor, setting the code value

Enumerations AccountType

- We can now access the code value from the enum.

```
AccountType type = AccountType.Deposit;
String code = type.getCode();
System.out.println(code);
```

Terminology

Key terms used in this lesson included:

- Generic Class
- Type Interface Diamond
- Use generic methods
- Use wildcards
- Use enumerated types

Summary

In this lesson, you should have learned how to:

- Create a custom generic class
- Use the type interface diamond to create an object
- Use generic methods
- Use bounded and unbounded wildcards
- Use enums

ORACLE® ACADEMY