# Generics

**Fundamentals** In programming, you may want to write code that can be used by more than one (1) type with the same underlying behavior. You don’t want to repeat your code line by line, only changing what data types it can access. For example, the methods setValue() and getValue() should be able to set and retrieve other data types besides String.

Generic types (or generics) associate one or more non-specified Java types upon creation. These are declared within angle brackets **<>**.

Example:

In the second line, **private T data**, T is used to declare an object called data. data will be an object of the type passed to T. For example, if type String is passed to T, then in that instance, data will be of type String.

**public void setValue(T dt) { data = dt;**

**}**

The parameter of the setValue() method is of type T. This means that the actual type of dt is determined by the type passed to T when a Gen object is created. Also, because both the parameter dt and the member variable data are of type T, they will both be of the same actual type when a Gen object is created.

**public T getValue(){ return data;**

**}**

The type parameter T can also be used to specify the return type of a method, as is the case with the getValue() method. Because data is also of type T, its type is compatible with the return type specified by getValue().

**public class Gen<T> { private T data;**

**public void setValue(T dt) { data = dt;**

**}**

**public T getValue(){ return data;**

**}**

**public void showType() { System.out.println("Type of T is " +**

**data.getClass().getName());**

**}**

**}**

**public void showType() { System.out.println("Type of T is " +**

**data.getClass().getName());**

**}**

The T in the first line, **public class Gen<T>**, is the name of a type parameter. This name is used as a placeholder for the actual type that will be passed to Gen when an object is created. Thus, T is used within Gen whenever the type parameter is needed. Because Gen uses a type parameter, Gen is a generic class, which is also called a parameterized type. There is no special significance to the name T. Any valid identifier could have been used, but T is traditional. It is recommended that type parameter names be single-character capital letters. The most commonly used parameter names are E - Element, K

- Key, N - Number, T - Type, V - Value.

The showType() method displays the type of T by calling getName() on the

Class object returned by the call to getClass() on data. The getClass() method is defined by Object and is thus a member of all class types. It returns a Class object corresponding to the type of the class of the object on which it is called. Class defines the getName() method, which returns a string representation of the class name.

The following class named GenDemo demonstrates the generic Gen class.

**public class GenDemo {**

**public static void main(String[] args) { Gen<Integer> iData = new Gen<Integer>(); Gen<String> sData = new Gen<String>(); iData.setValue(1);**

**iData.showType(); sData.setValue("One"); sData.showType();**

**int num = iData.getValue();**

**String str = sData.getValue(); System.out.println("Integer value: " + num); System.out.println("String value: " + str);**

**}**

**}**

To initialize a generic object, you define the generic class name followed by the

type in the angle brackets and then give the object a name, such as the following:

# Gen<Integer> iData = new Gen<Integer>();

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

# Gen<Integer> iData = new Gen<>();

You can also initialize a generic object with two (2) types.

# Gen<Integer, String> g = new Gen<>();

Example:

**Wildcard Arguments** The **wildcard argument** represents an unknown type and is specified by the question mark. Wildcards allow greater control of the types you use. They fall into two (2) categories: unbounded and bounded.

**int mdType1 = mixData.getT1Value(); String mdType2 = mixData.getT2Value(); int idType1 = iData.getT1Value();**

**int idType2 = iData.getT2Value();**

**System.out.println("First pair: " + mdType1 + " and " + mdType2);**

**System.out.println("Second pair: " + idType1 + " and " + idType2);**

**}**

**}**

<?> denotes an **unbounded wildcard**. It can be used to represent any type. Example:

**public void printList(List<?> list) { for (Object elem : list)**

**System.out.println(elem + " "); System.out.println();**

**}**

The above method can be used to print a list of any type.

**public class Gen<T, T2> { private T data1; private T2 data2;**

**public void setValue(T dt1, T2 dt2) { data1 = dt1;**

**data2 = dt2;**

**}**

**public T getT1Value(){ return data1;**

**}**

**public T2 getT2Value(){ return data2;**

**}**

**}**

**List<Integer> li = Arrays.asList(1, 2, 3);**

**List<String> ls = Arrays.asList("one", "two", "three"); printList(li);**

**printList(ls);**

The following class demonstrates the generic class.

**public class GenDemo {**

**public static void main(String[] args) { Gen<Integer, String> mixData = new Gen<>(); Gen<Integer, Integer> iData = new Gen<>(); mixData.setValue(1, "One"); iData.setValue(7, 25);**

If you want to relax restrictions on a variable, you can use an **upper bounded wildcard**. It restricts the unknown type to be a specific type or a subtype of that type. To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword and its upper bound. For example, you can create a method that works only on a list of numbers.

**public double sumOfList(List<? extends Number> list) { double s = 0.0;**

**for (Number n : list)**

**s += n.doubleValue(); return s;**

**}**

The **sumOfList** method returns the sum of the numbers in a list.

**List<Integer> li = Arrays.asList(1, 2, 3); System.out.println("sum = " + sumOfList(li)); List<Double> ld = Arrays.asList(1.2, 2.3, 3.5); System.out.println("sum = " + sumOfList(ld));**

A **lower bounded wildcard** restricts the unknown type to be a specific type or a super type of that type. A lower bounded wildcard is expressed using the wildcard character ('?'), followed by the super keyword and its lower bound. Let’s say you want to create a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on anything that can hold Integer values, such as Integer, Number, and Object.

To write the method that works on lists of Integer and the super types of Integer, you would specify **List<? super Integer>**. For example, the following method adds the numbers 1 through 10 to the end of a list:

**public void addNumbers(List<? super Integer> list) { for (int i = 1; i <= 10; i++) {**

**list.add(i);**

**}**

**}**