Generics & Wrapper Classes

In This Lecture

1. Wrapper Classes
2. Autoboxing & Un-boxing
3. Generics
4. Bounded Generics

Wrapper Classes

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

```
class myClass {
    int data
}
```

Need of Wrapper Classes

1. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
2. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
3. An object is needed to support synchronization in multithreading.

Autoboxing & Unboxing

The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double, etc.

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

Generics

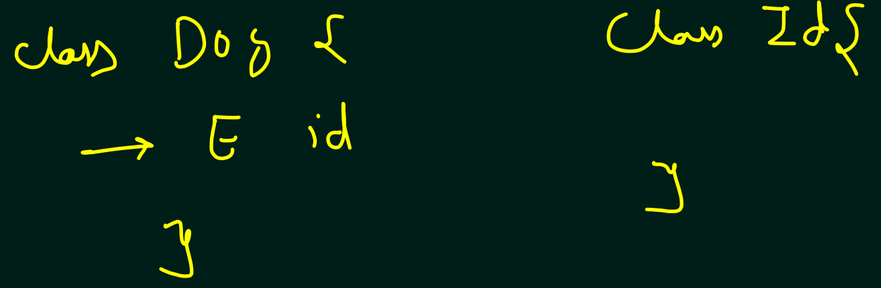
Generics means parameterized types. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

```
// create a generics class
class GenericsClass<T> {

  // variable of T type
  private T data;

  public GenericsClass(T data) {
    this.data = data;
  }

  // method that return T type variable
  public T getData() {
    return this.data;
  }
}
```



Java Generics Method

Similar to the generics class, we can also create a method that can be used with any type of data. Such a class is known as Generics Method.

```
public <T> void genericMethod(T data) {...}
```

Here, the type parameter <T> is inserted after the modifier public and before the return type void.
We can call the generics method by placing the actual type <String> and <Integer> inside the bracket before the method name.

```
demo.<String>genericMethod("Java Programming");

demo.<Integer>genericMethod(25);
```

Bounded Generic Types

In general, the type parameter can accept any data types (except primitive types). However, if we want to use generics for some specific types (such as accept data of number types) only, then we can use bounded types.
In the case of bound types, we use the extends keyword.
Here, GenericsClass is created with bounded type. This means GenericsClass can only work with data types that are children of Number (Integer, Double, and so on).

```
class GenericsClass <T extends Number> {

  public void display() {
    System.out.println("This is a bounded type generics class.");
  }
}
```