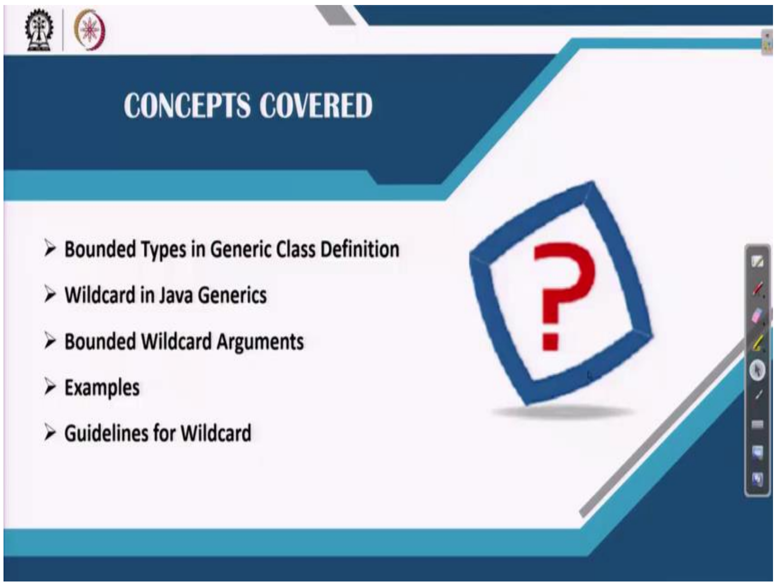
Data Structures and Algorithms Using Java
Professor. Debasis Samanta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 05
Bounded Argument Generic Class

So, this is in continuation of the last lectures on the topic generic programming. We today discuss about an advance feature of it, it is called the bounded arguments in generic programming. So, in the last video, you can recall we have pointed out one precaution that a programmer should take.

(Refer Slide Time: 00:57)



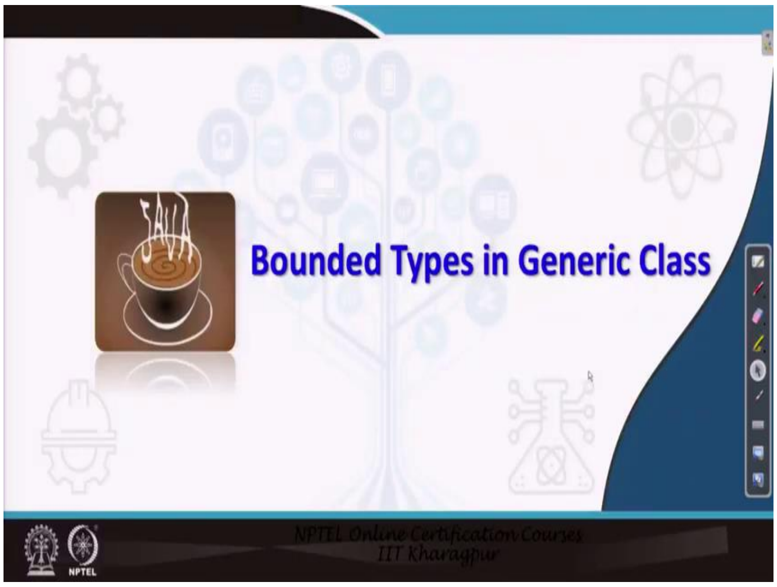
CONCEPTS COVERED

- Bounded Types in Generic Class Definition
- Wildcard in Java Generics
- Bounded Wildcard Arguments
- Examples
- Guidelines for Wildcard

Now, today's video lectures is regarding to that steps actually. So, first I will discuss about as a precautionary measure the concept of bounded types in generic programming and how you can define a class specifying the bounded type arguments. There is an extra feature in this regard also, this is called the wildcard in Java.

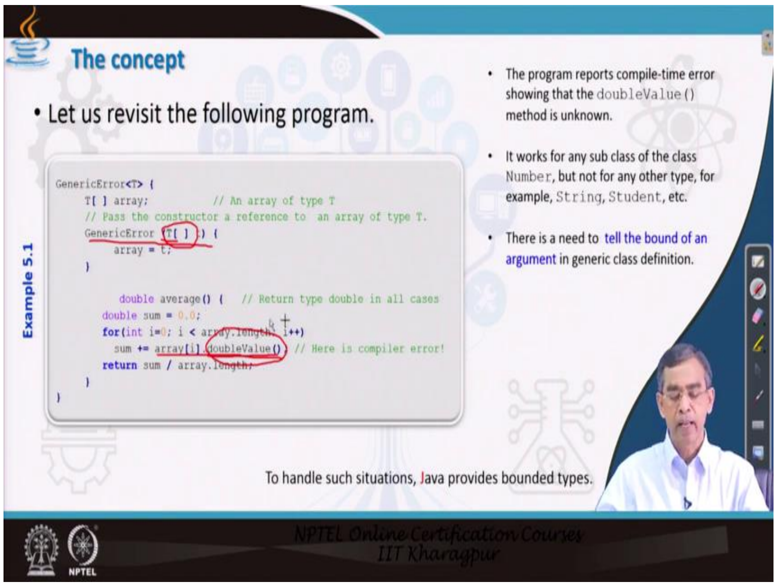
So, we will discuss about wildcard, so far the generic programming is concerned and then there is again extension of this wildcard concept it is called the bounded wildcard arguments. We will finally highlights its applications with some examples to understand the concept better.

(Refer Slide Time: 01:46)



Bounded Types in Generic Class

NPTEL Online Certification Courses
IIT Kharagpur



The concept

- Let us revisit the following program.

Example 5.1

```
GenericError<T> {
    T[ ] array;           // An array of type T
    // Pass the constructor a reference to  an array of type T.
    GenericError (T[ ] t) {
        array = t;
    }

        double average() {   // Return type double in all cases
      double sum = 0.0;
      for(int i=0; i < array.length; i++)
        sum += array[i].doubleValue(); // Here is compiler error!
      return sum / array.length;
    }
}
```

- The program reports compile-time error showing that the doubleValue() method is unknown.
- It works for any sub class of the class Number, but not for any other type, for example, String, Student, etc.
- There is a need to tell the bound of an argument in generic class definition.

To handle such situations, Java provides bounded types.

NPTEL Online Certification Courses
IIT Kharagpur

Now, let us come to the discussion about, so the bounded types in generic class definition. So, the problem that we have faced in the last one example that you can recall, so there is a problem, problem that if you define a template in generic programming, this is for a class or for a method whatever it may be, but somehow if you do not define its scope properly, then it may invite certain problem.

So, problem regarding for a particular objects, if you want to objects of a particular type if you want to access a methods which is not defined in that class, then it will give a compilation error as you can, we seen in the last examples here. In this example, so this is a program and we define this program with a generic class and T is the template and we use this template class definition to access a method for that type. So, T is a type and then here array is basically is an object of that type and we want to access a method double value under an assumption that this double value method is defined for this type T.

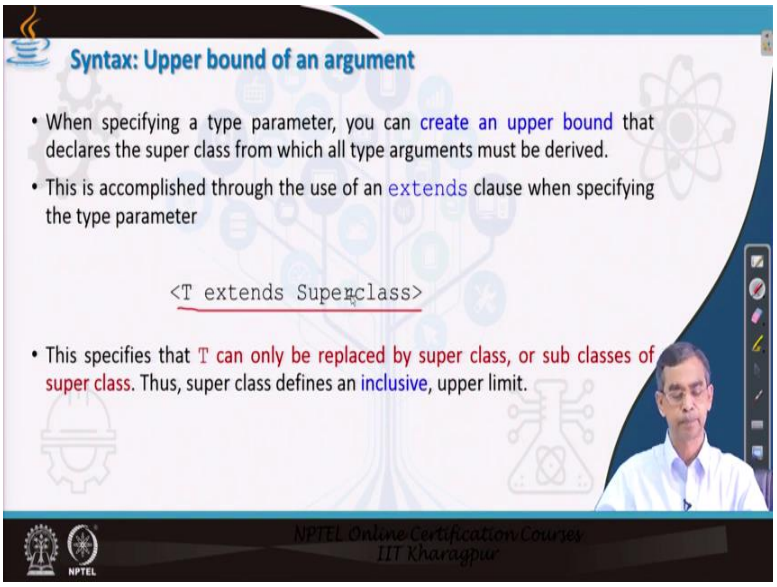
Now, this double value method is in fact is defined in a class called number. Now you know, if a method is defined in a class, then any objects which is of that class or subclass of that class is basically under this method accessing, this means that this double value method is defined in number class and its subclass like integer, double, float, long, short all these things they are the subclass of the number class, that means this method resolution is applicable to number as well as any subclass of the class number.

So, this method, this program will not give any error if you call or if you run this program with T type which belongs to either number class or any of its subclass. However, it will give compilation error if we define this T some other class, for example string class. So, this program will work for numeric values supposed to be but will not work for other types.

Now that, that basically is a problem because scope is not properly defined and compiler, Java compiler will not be able to resolve this scope at the time of compilation this because double value is not defined for any type that is why and then it will basically gives a compile time error.

Now, so this problem is basically to solve using the concept it is called the bound of an argument. So, somehow a programmer should specify the bound of an argument, so this argument, for example this argument what should be the bound? Bound means the scope of it or that validity of this argument belongs to a particular class or its subclass, some of that information needs to be mentioned while you write the program and that is why the concept of bounded class is coming bounded argument is coming.

(Refer Slide Time: 05:50)



Syntax: Upper bound of an argument

- When specifying a type parameter, you can create an upper bound that declares the super class from which all type arguments must be derived.
- This is accomplished through the use of an `extends` clause when specifying the type parameter

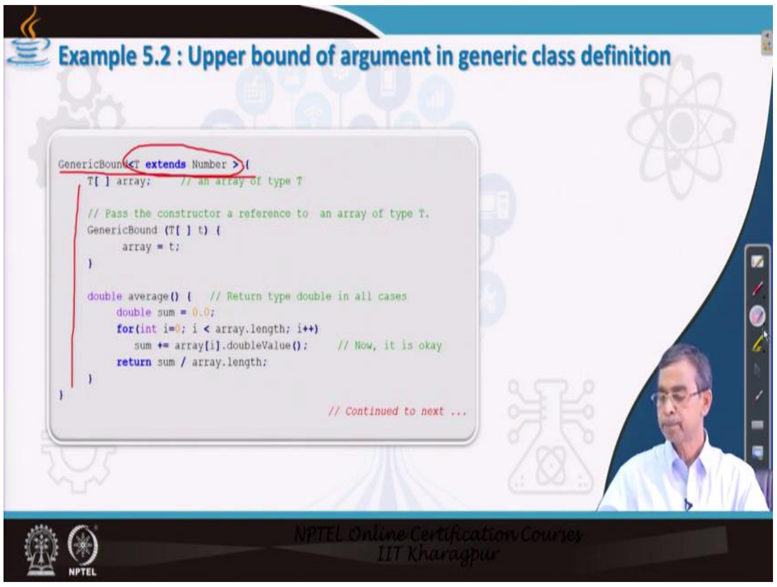
```
<T extends Superclass>
```

- This specifies that T can only be replaced by super class, or sub classes of super class. Thus, super class defines an inclusive, upper limit.

Now so far, bounding of argument is concerned, there are many bounding procedural first let us discuss about upper bound of an argument, that means while you declare a class and in that class definition if you want to mention a type generically, say T then we can specify its bound upper bound of that T template. Now, the syntax is pretty simple, this syntax you can follow in the previous examples we have discussed that, so this is the class name and T angular bracket close, but if you want to say that T has its upper bound belongs to a particular class, then we can use this symbol.

So, that for example generic array T extends number this means that, this T is only valid or scope of this T is only valid to number class or any of its subclass. So that is why the concept it is there, so this is the simple syntax that you should for while you are defining a generic class and if you want to mention that the generic class definition has it's bound or upper bound up to this class or any or its subclass.

(Refer Slide Time: 07:21)



Example 5.2 : Upper bound of argument in generic class definition

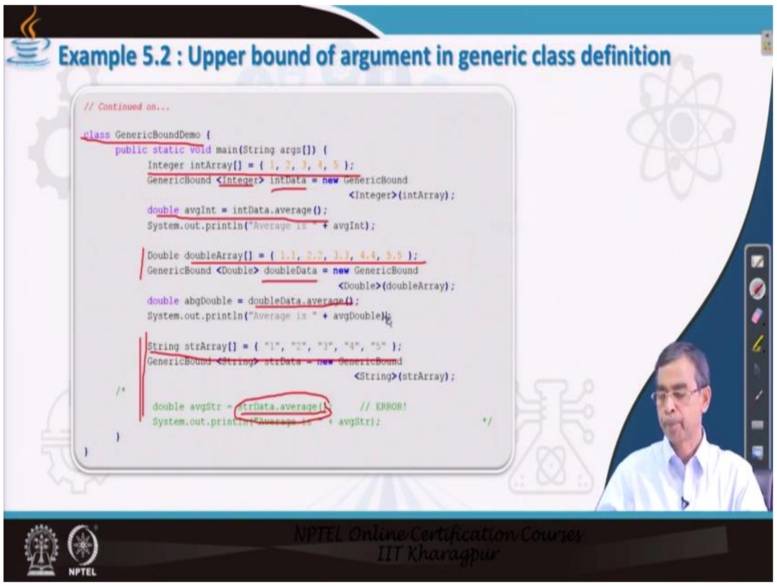
```
GenericBound<T extends Number >{
    T[ ] array;      // an array of type T

    // Pass the constructor a reference to  an array of type T.
    GenericBound (T[ ] t) {
        array = t;
    }

    double average() {    // Return type double in all cases
        double sum = 0.0;
        for(int i=0; i < array.length; i++)
            sum += array[i].doubleValue();     // Now, it is okay
        return sum / array.length;
    }
}
                                        // Continued to next ...
```

NPTEL Online Certification Courses
IIT Kharagpur

Let us see one example in this regard, as we can repeat the same example that we have discussed in the last video lectures, here is the okay I have modified, so generic bound class the new class definition but I have modified these things. So, now this implies that this T has its upper bound to number or any of its sub class, rest of the things are pretty simple this will work fine.

Now, having this kind of, this kind of declaration, we can go a little bit forward to see what is the next part of the program, so this is basically the generic class definition I just want to use it in a main program, so elevating the problem that we face in the last lectures.

(Refer Slide Time: 08:08)



Example 5.2 : Upper bound of argument in generic class definition

```
// Continued on...

class GenericBoundDemo {
    public static void main(String args[]) {
        Integer intArray[] = { 1, 2, 3, 4, 5 };
        GenericBound <Integer> intData = new GenericBound
                                         <Integer>(intArray);

        double avgInt = intData.average();
        System.out.println("Average is " + avgInt);

        Double doubleArray[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        GenericBound <Double> doubleData = new GenericBound
                                         <Double>(doubleArray);

        double abgDouble = doubleData.average();
        System.out.println("Average is " + avgDouble);

        String strArray[] = { "1", "2", "3", "4", "5" };
        GenericBound <String> strData = new GenericBound
                                         <String>(strArray);
    /*
         double avgStr = strData.average();      // ERROR!
         System.out.println("Average is " + avgStr);               */
    }
}
```

NPTEL Online Certification Courses
IIT Kharagpur

So, here is basically the main program or it is a driver class, we just give the program where I want to use the generic class that I have defined using upper bounded arguments. So, we declare a collection it, is an array of integers and then we can recreate this is the object, this object is generic object because it basically takes the integer array. Now, then we can call this method it absolutely no issue because it works and then compiler will not report any error because compiler has resolved that, this double value method which is there in average is resolved through this number specification it is there.

The same is true for other instances also we create another collection of double values and then we create an object of that collection and then call this method up, so it is also does not give any compilation error. However, this will give an error because here we are declaring a string array, array of strings and then we want to call this method average. Now, this average method will not be able to resolve its scope, because the string class is not under this bounded argument. So, it will basically report and again compilation error that you will not be able to use this because bound of your argument is within the number and in its subclass.

So, this will gives an error, but this will not gives an error as in the last video lectures you saw that even if this also there, but without that definition upper bounding and argument it gave error. So, this is a concept that upper bounder argument, that is there in Java programming or we can say the Java generic programming.

(Refer Slide Time: 10:04)

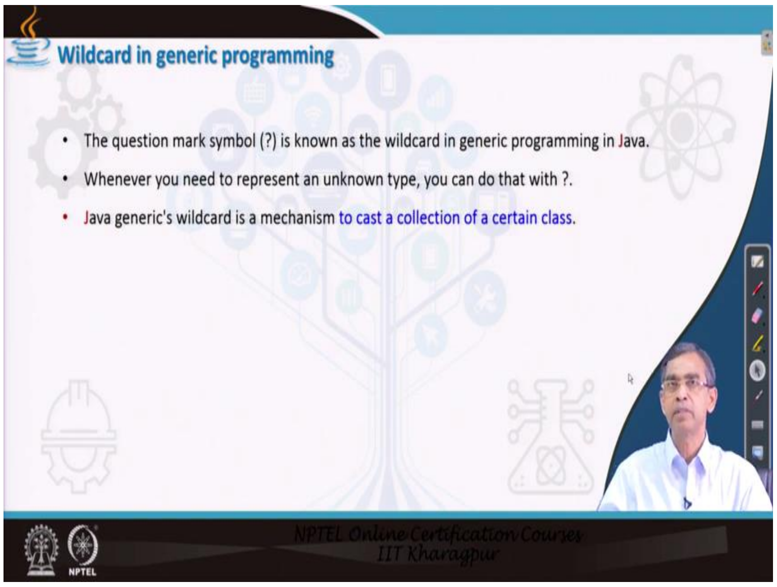
Wildcard in Java Generics

Now, let us see there is an another advancement in this regard, this is again related to the bounding of arguments and it is called wildcard. You probably know what is exactly wildcard, now whenever you have to list in Unix suppose, all the files which has the extension say doc, so that list you can say ls star dot doc, so it is basically, it is a wildcard symbol, that means star will replace any what is called the name of the file but extension should be doc.

Now this is the concept of wildcard, now in Java generic programming the same wildcard concept is used to indicate that it will not problem for any particular casting of a particular type, now again I want to see what exactly the costing of a type. Now you know for the numeric data there is a different types of values that we can have all our of numeric type, for example int, then long it is long int, then short short int, then float, double like this.

Now, if you want to have any calculation of mixing the different type numerical values but of different types, so in an expression if you have some values or operands with say long type with say float type and if you do not do any casting, then it gives an error because casting will not match and type mismatch error it will be there, but if you do casting then this problem can be avoided. So, that is also wildcard is basically related to this casting, now let us see one example, so that we can understand how it matters in your program.

(Refer Slide Time: 12:00)

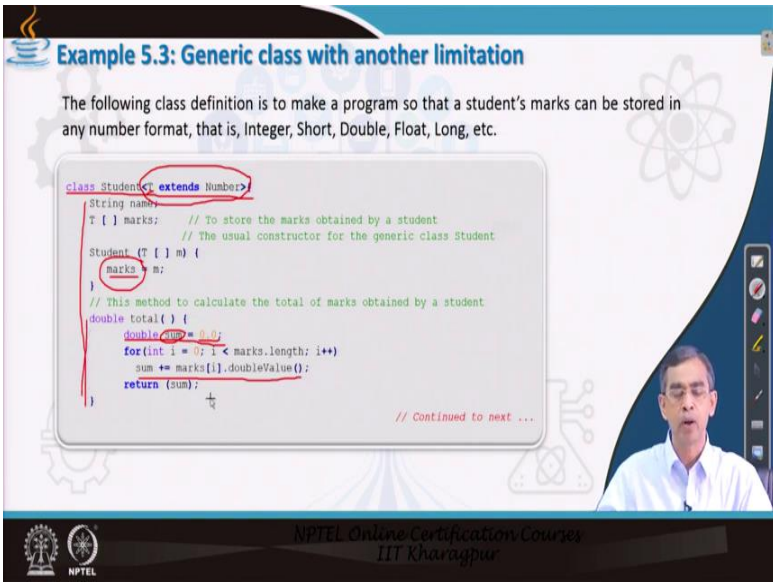


Wildcard in generic programming

- The question mark symbol (?) is known as the wildcard in generic programming in Java.
- Whenever you need to represent an unknown type, you can do that with ?.
- Java generic's wildcard is a mechanism to cast a collection of a certain class.

So, basically here if you want to deal with a collection of a particular type or collections of different types belongs to the same, say numeric concept or string or some other like whatever it is there, then sometimes it may need in require to do the casting the collection to a particular certain class so that it can work, so type mismatch error can be avoided.

(Refer Slide Time: 12:26)



Example 5.3: Generic class with another limitation

The following class definition is to make a program so that a student's marks can be stored in any number format, that is, Integer, Short, Double, Float, Long, etc.

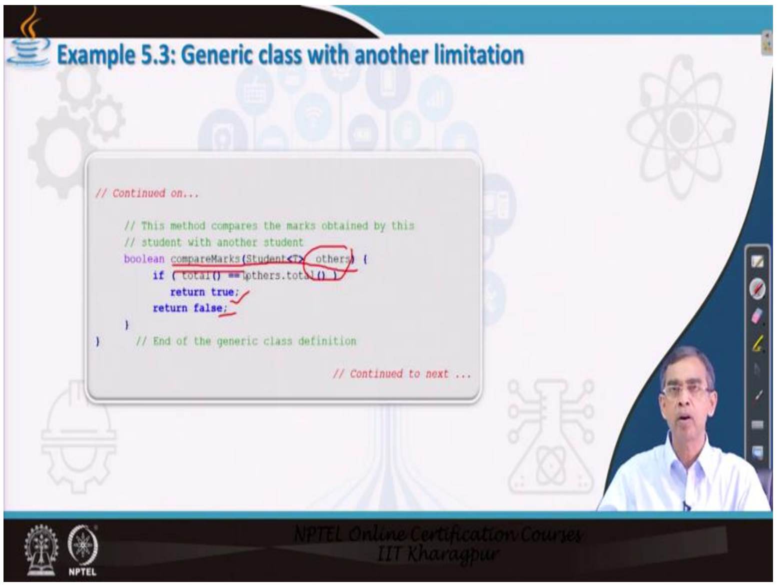
```
class Student<T extends Number>{
    String name;
    T [ ] marks;     // To store the marks obtained by a student
                    // The usual constructor for the generic class Student
    Student (T [ ] m) {
        marks = m;
    }
    // This method to calculate the total of marks obtained by a student
    double total( ) {
          double sum = 0.0;
          for(int i = 0; i < marks.length; i++)
            sum += marks[i].doubleValue();
          return (sum);
    }
                                         // Continued to next ...
```

Now regarding this thing, let us have a very simple example fast and here actually this this program will not work for you, I will tell you the remedy for this, this remedy is using generic using wildcard concept. Now, let us see the program it is a very simple program again, we have to just check the program, so here you can see we are defining one generic class and here also we have mentioned this is basically upper bound argument of this T.

Now, rest of the things are very simple, here only the thing that you should note that we declare a temporary variable of type double, the variable name is sum initialized a 0.0 and then this is the for loop it basically for a given collection which is stored in marks, marks is basically collection of say numbers, numbers of any type like and then it will basically calculate the sum of all values which are stored in this array marks and then it basically calculate the sum of all this and then return sum whatever it is, you can say, so total.

So, this method I basically calculate the total of all the values which is stored in the array. So, this is pretty simple one concept. Now let us see using this generic class definition in your main program the driver class is look like this.

(Refer Slide Time: 13:58)



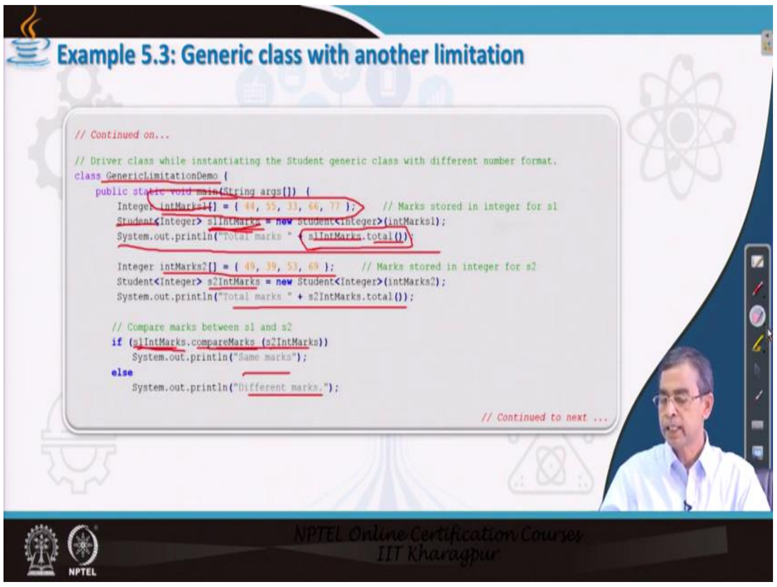
Example 5.3: Generic class with another limitation

```
// Continued on...

    // This method compares the marks obtained by this
    // student with another student
    boolean compareMarks(Student<T> others) {
        if ( total() == others.total() )
            return true;
        return false;
    }
}     // End of the generic class definition

                                        // Continued to next ...
```

So, this is another method, I should say another method also belongs to this class, this method is basically compare. So, is basically compare one collection with others or we can say compare the total returned by one collection with other collection. So for a given collection, if we call this method to compare other collection which can be passed as an argument, then we will see if the total of the current class is same as the total of the comparable class is same, then it will return true otherwise it will return false.

So, this is a compare simply it will compare the sum of two collections and the method will be called for one class and argument will be passed for other class for which we want to compare its total. So this is the concept now let us have the main class.

(Refer Slide Time: 14:56)



Example 5.3: Generic class with another limitation

```
// Continued on...

// Driver class while instantiating the Student generic class with different number format.
class GenericLimitationDemo {
    public static void main(String args[]) {
        Integer intMarks1[] = { 44, 55, 33, 66, 77 };    // Marks stored in integer for s1
        Student<Integer> s1IntMarks = new Student<Integer>(intMarks1);
        System.out.println("Total marks " + s1IntMarks.total());

        Integer intMarks2[] = { 49, 39, 53, 69 };    // Marks stored in integer for s2
        Student<Integer> s2IntMarks = new Student<Integer>(intMarks2);
        System.out.println("Total marks " + s2IntMarks.total());

       // Compare marks between s1 and s2
       if (s1IntMarks.compareMarks (s2IntMarks))
           System.out.println("Same marks");
       else
           System.out.println("Different marks.");

                                                            // Continued to next ...
```

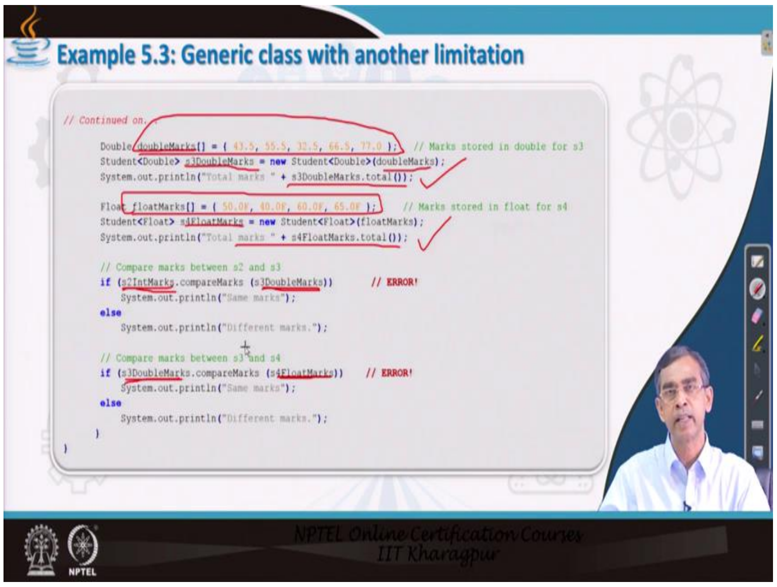
Our main class is look like this, we have to little bit careful about because we have called this method for different collections with the different varieties. So, this is basically the main program the main method and here you see we declare one collection of type integer, so it is basically array of integers and we store int marks this one and we also declare another objects this is an array of integer objects and this is basically the collection here, we just using the student generic class we create the object, so this is basically s1 means students one set of where the set values are integer marks actually.

So, this is the concept here and then it basically gives a, I mean call for this s1 interior marks or total methods, total method is defined in the student class, so for that object we call this method it works for you. Now we create another array of integers this one and then we create another collection s2 integer marks, so these are the two sets s1 and s2, all are of same type integer and then we again find the total absolutely no issue.

Now here you see we call the compare marks, compare marks for the current set with the another set. So, two sets are there s1 and s2, so we call this one and compare marks, compare the two sets sum and if it is true that means they their totals are same, then it will return true so it will print same marks, otherwise it will print different marks.

Now here this part of the program, absolutely there is no problem because we are calling this and we are calling this it is working and you see that this part of the program will work but however if we little bit go further and we will see some other parts of the program, then it may not work for you. Now, let us see first there is another part of the program extending the same thing but not for integer array some other array of collections.

(Refer Slide Time: 17:17)



Example 5.3: Generic class with another limitation

```
// Continued on..

    Double doubleMarks[] = { 43.5, 55.5, 32.5, 66.5, 77.0 };  // Marks stored in double for s3
    Student<Double> s3DoubleMarks = new Student<Double>(doubleMarks);
    System.out.println("Total marks " + s3DoubleMarks.total());

    Float floatMarks[] = { 50.0F, 40.0F, 60.0F, 65.0F };    // Marks stored in float for s4
    Student<Float> s4FloatMarks = new Student<Float>(floatMarks);
    System.out.println("Total marks " + s4FloatMarks.total());

    // Compare marks between s2 and s3
    if (s2IntMarks.compareMarks (s3DoubleMarks))       // ERROR!
       System.out.println("Same marks");
    else
       System.out.println("Different marks.");

    // Compare marks between s3 and s4
    if (s3DoubleMarks.compareMarks (s4FloatMarks))    // ERROR!
       System.out.println("Same marks");
    else
       System.out.println("Different marks.");
   }
}
```

Now, we are just extending this one it is in the same fashion, but we create another array of numbers it is a floating-point numbers, it is the double marks array and we create an collection objects s3 double marks of the type student passing this array to it, so this s3 double marks is basically is a collection of this one and it will find the total.

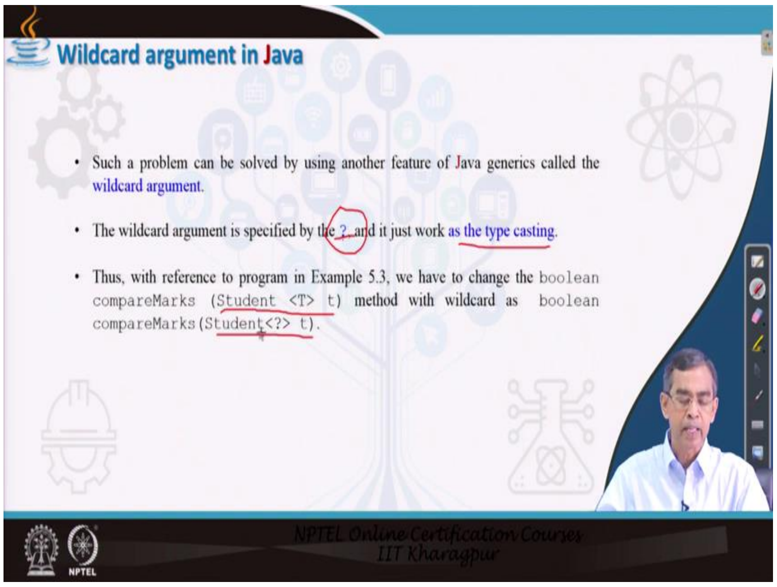
So, total will be calculated using the method it is there, you will see that this will work for you and another is a float marks another array it is at float type, we create another collection and then that for the collection we calculate total, also it will not give any error, while you are comparing you are compiling the program.

However, this part of the program will not work for you. Now let us see here s2 int marks compare mark s3 double marks, this part also s3 double marks compare s4 float marks alternatively if you could call s1 int marks compare s4 float mark it will also not work whatever it is there, what we can see is that, this is a type of collection basically is a integer and this is the type of collection basically double values, this is the collection of type double values, whereas this is the collection of float values.

Now what is the problem? We are calling this is a object of particular type and comparing the objects of another type. Now the two objects are not of same type and that basically invites a problem called the casting of type problem, so it will not work for you.

So, this is a one limitation that if we can call a method for a particular type with another type, which is there in the compare method, compare marks basically in the earlier definition its supposed to have the same type for which we are calling and for which we are sending, so that is why the problem the type mismatch. So, this basically gives a compilation error and in this program therefore, needs certain remedy.

(Refer Slide Time: 19:44)



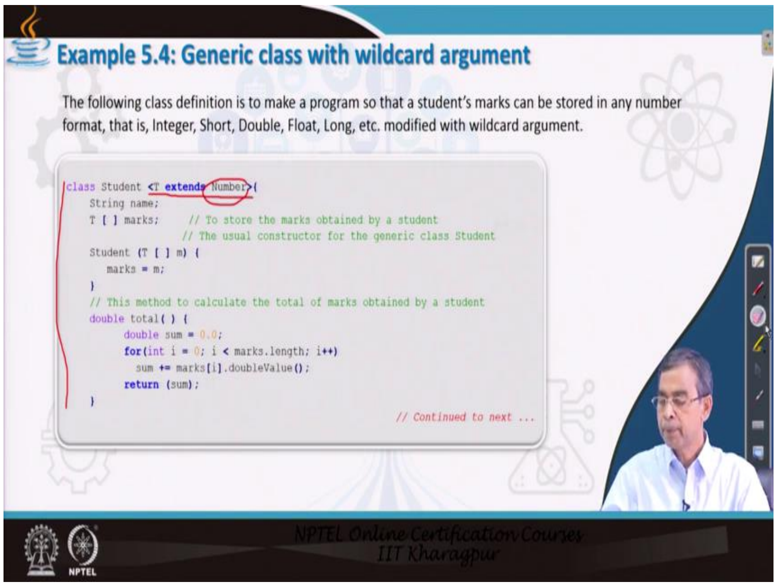
Wildcard argument in Java

- Such a problem can be solved by using another feature of Java generics called the wildcard argument.
- The wildcard argument is specified by the ? and it just work as the type casting.
- Thus, with reference to program in Example 5.3, we have to change the `boolean compareMarks (Student <T> t)` method with wildcard as `boolean compareMarks(Student<?> t)`.

The remedy is in the form of what is called the wildcard. Now in Java the wildcard symbol, that means it basically replaces T by a symbol; it is called the interrogative symbol it is just like this one. So if we write in place of T with this one, then it basically says that is wildcard, wildcard indicated that this basically works for whatever be the type belongs to the same category, so this is basically called the wildcard symbol used as a typecasting.

Now the symbol that means earlier in declaration as you can see we wrote student T t, so if we use wildcard then we should write student then this T, so if we use it then the typecasting can be enforced. So this symbol in place of template T is basically the wildcard symbol it is there. Now let us see the problem, modified problem with this wildcard and if we run it and you will see that it will run for you.

(Refer Slide Time: 20:56)



Example 5.4: Generic class with wildcard argument

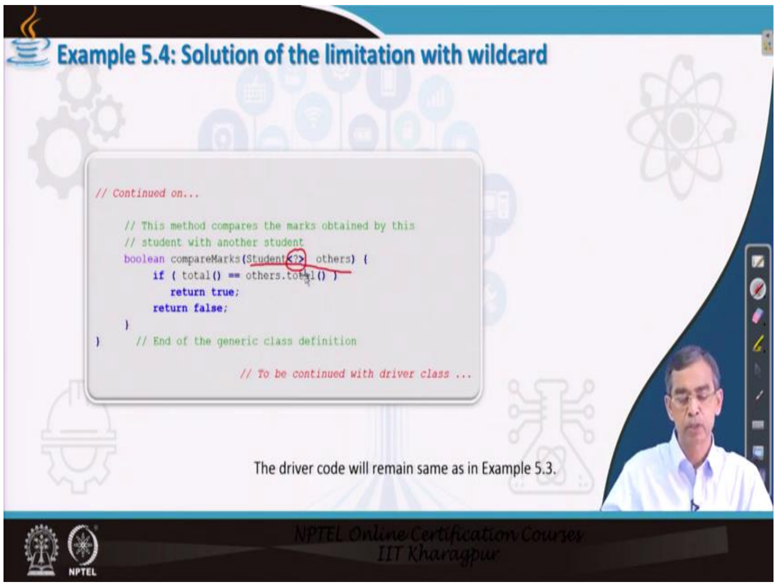
The following class definition is to make a program so that a student's marks can be stored in any number format, that is, Integer, Short, Double, Float, Long, etc. modified with wildcard argument.

```
class Student <T extends Number>{
    String name;
    T [ ] marks;     // To store the marks obtained by a student
                    // The usual constructor for the generic class Student
    Student (T [ ] m) {
       marks = m;
    }
    // This method to calculate the total of marks obtained by a student
    double total( ) {
          double sum = 0.0;
          for(int i = 0; i < marks.length; i++)
            sum += marks[i].doubleValue();
          return (sum);
    }
                                                  // Continued to next ...
```

NPTEL Online Certification Courses
IIT Kharagpur

So here is the program that program again same thing, only the modification that I have to do is basically, here it is same because I want to extend this generic class that it should have its scope within the number class or any subclass, so this part is fine, absolutely there is no problem only the modification or remedy, correction that is required it is there in the compare marks, because compare marks facing problem.

(Refer Slide Time: 21:25)



Example 5.4: Solution of the limitation with wildcard

```
// Continued on...

    // This method compares the marks obtained by this
    // student with another student
    boolean compareMarks(Student<?>  others) {
         if ( total() == others.total() )
            return true;
         return false;
    }
}      // End of the generic class definition

                              // To be continued with driver class ...
```

The driver code will remain same as in Example 5.3.

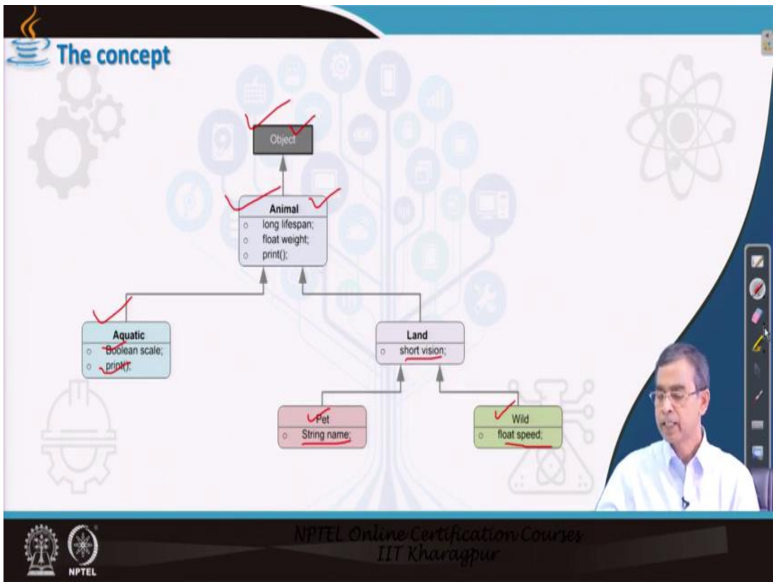
NPTEL Online Certification Courses
IIT Kharagpur

So here is the compare marks and in the compare marks you can see we have defined in a deeper in our earlier it basically student T others. So, basically the thing is that others is a type of student class. Now here we put the wildcard in place of T in the previous example it was T here and then is basically that student has any type, that means for which method for which object you are calling it will not matter here it will work, so typecasting is automatically enforced.

So, with this correction or modification the program is now ready for you to compile and run it and here is the main program or the main program is remains same you can run it and you will see this. So, there is a little modification that we have incorporated to avoid certain limitation which is there.

Now this is the concept of the bounding and argument why we are defining generic class also we have discussed about wildcard of defining in a generic class definition. Now, like the bounding of arguments there is also necessity of bounding the wildcard arguments. Now, I will discuss about how the wildcard arguments bounding can be done.

(Refer Slide Time: 22:48)



The concept

Now to discuss this concept, this concept is little bit complex because it has many intricacies are there. So, my submission is that let us have a example, example I will try it to give as trivial as possible but since the thing is itself complex, so if we do not go a little bit complicated example you will not be able to understand it but you have to hold your patience and to understand it follow, if required you can repeat the videos several times so that you can understand it.

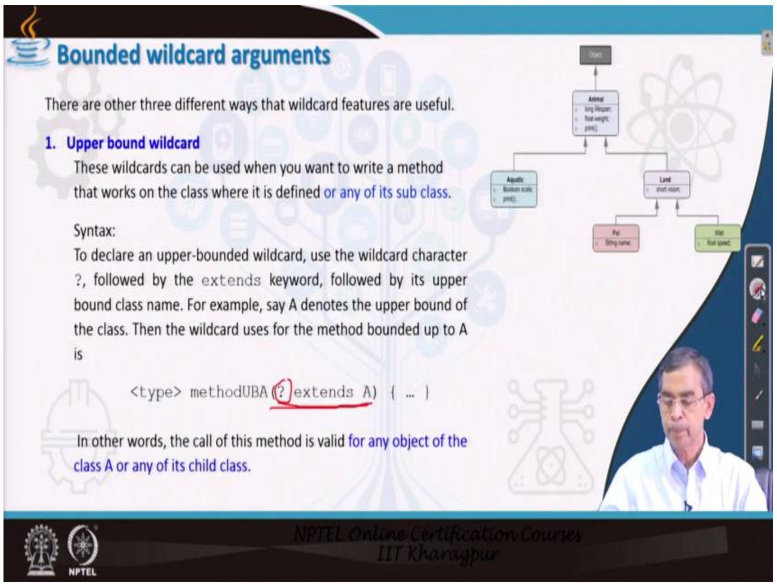
Now, let us see how we can proceed it. We have to have certain examples, so basically we consider a class hierarchy, it is a basically animal class hierarchy. So, this basically is a superclass animal it has field of long, float and one method print. So, long is basically to store its sum values, now there is a two subclasses of this class superclass animal they are called aquatic, aquatic has its own field and this method print is basically is an over writing method.

Another subclass of this class animal is land, it does not have any print method this means this print method is binded to it and this is the only one field it is there. Now, similarly land has its own two subclasses namely pet and wild, it has arguments field is string one name and is the speed. So, these are the simple class hierarchy where the different classes are there, some classes are inherited from some other classes and everything and you know, so these basically the class definition and for every class by default there is a superclass.

So, it is a basically galaxy super it is called, this is the object class. You do not need to define but in Java system all class are basically is a inherited from this object class, so our animal class also as far the same convention and rule it is basically is a child class of this super galaxy class called objects. So, this is the example that we want to follow it, now what is actually our recommend, in our program we are now going to write a program, where we want to define everything generically.

So, objective is that we want to maintain a list, the list contains any type of animals there in the list, so in order to do these things we have to follow the generic class concept. So, that is why. Now let us see first, how all that class, which are here in this hierarchy can be defined one by one, that means we want to define first animal class, aquatic class, land class, pet class, wild class with generic concept or whatever it is there. Now let us see how we can define it, there are few small part of the program regarding definition of each class in our things it is there.

(Refer Slide Time: 26:00)



Bounded wildcard arguments

There are other three different ways that wildcard features are useful.

1. **Upper bound wildcard**

   These wildcards can be used when you want to write a method that works on the class where it is defined or any of its sub class.

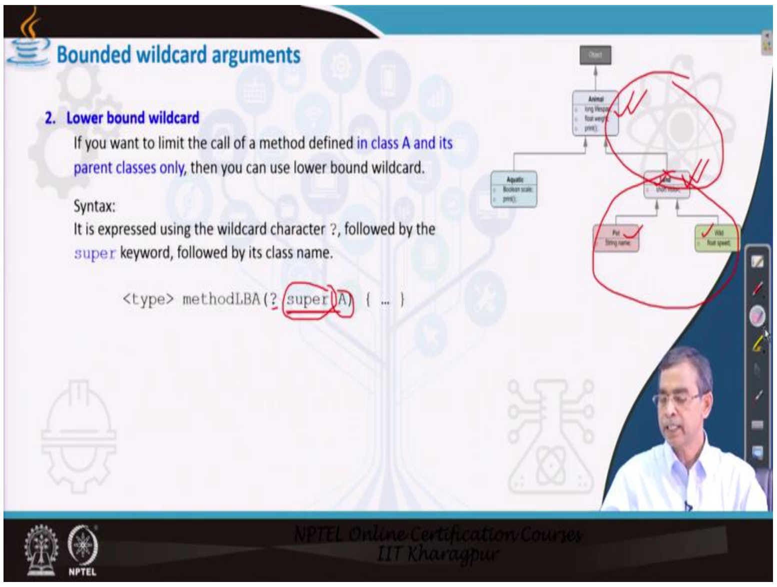
   Syntax:
   To declare an upper-bounded wildcard, use the wildcard character ?, followed by the `extends` keyword, followed by its upper bound class name. For example, say A denotes the upper bound of the class. Then the wildcard uses for the method bounded up to A is

   ```
   <type> methodUBA(? extends A) { … }
   ```

   In other words, the call of this method is valid for any object of the class A or any of its child class.

So, here is basically and here I want to say one thing there are two bounds or three bounds rather that can be extended, so far the wildcard is concerned, the first is called upper bound of wildcard it is the same as in (())(26:20) of T extends superclass A, if we write this extends A, then it is basically bounded wildcard, upper bounded wildcard argument. Similarly, for the lower bounded wildcard argument also can be defined.

(Refer Slide Time: 26:40)



Bounded wildcard arguments

2. **Lower bound wildcard**

   If you want to limit the call of a method defined in class A and its parent classes only, then you can use lower bound wildcard.

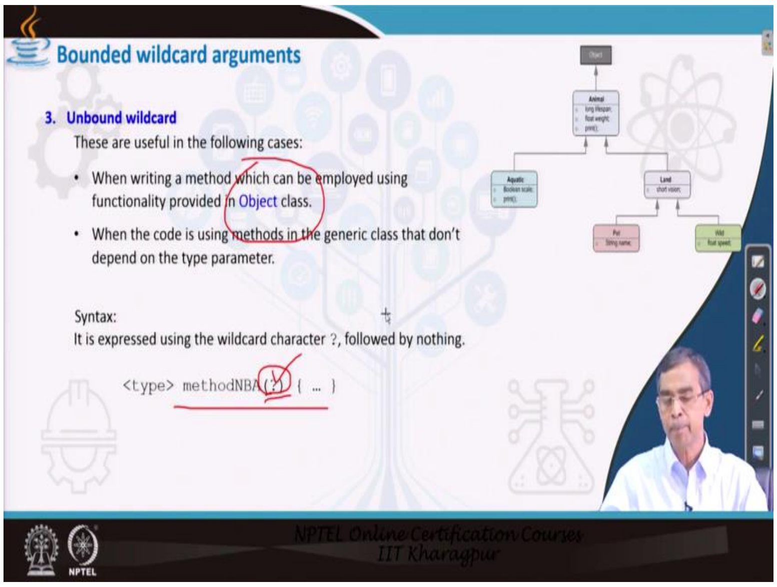
   Syntax:
   It is expressed using the wildcard character ?, followed by the `super` keyword, followed by its class name.

   ```
   <type> methodLBA(? super A) { … }
   ```

So, here is the lower bounded while car argument syntax as you see, so this basically this is the (())(26:47) of extends, right keyword we just use super and this is the name of the class, so what is the concept? If for a particular method defined in any class, if you use upper bound this means that method has its scope belong to this or any of its superclass, subclass so if we define a method here with upper bound argument, then that method has its scope belong to this class as well as any of its subclass.

On the other hand, if it is a lower bound and defining this method it has extend in this class and any of its superclass, so this one, so this basically is the lower bound and this basically is the upper bound argument. So, for a method defined in this class with these two bounds has the scope either any of its class or superclass or any of its class and its subclass.

(Refer Slide Time: 27:56)



Bounded wildcard arguments

3. Unbound wildcard

These are useful in the following cases:

- When writing a method which can be employed using functionality provided in Object class.
- When the code is using methods in the generic class that don't depend on the type parameter.

Syntax:

It is expressed using the wildcard character ?, followed by nothing.

```
<type> methodNBA(?) { … }
```

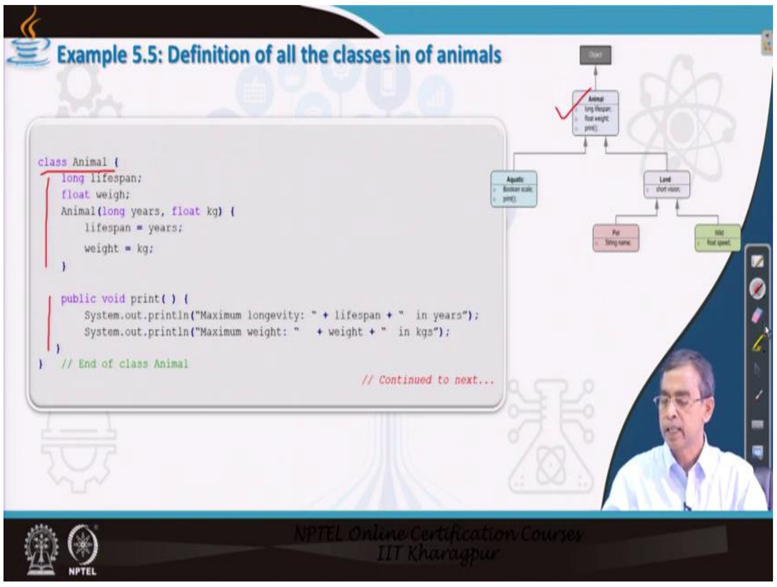
So, this basically is the concept of two bounds and there is unbound, unbound I will discuss unbound is basically there is no bound actually so that is why it is called the unbound and for this unbound we do not have to mention anything. So, here only we mention this one is basically unbound, what exactly it implies that.

So, this if this method is declared with unbounded arguments, then it is basically it has scope to any methods defining object class and the code methods in the generic class do not depend on any type parameter. So, that is why the concept, so it is basically same as the wildcard concept that we have discussed in the previous example it is same, only the thing is that it is related to object class only.

Now so with this understanding, let us first discuss the different classes and they are methods in each what we will do is that we will try to make a complete shape of all this class in addition to will define a method in each class is specific method and this method will basically print all the values those are there in that class or those are basically having in the by inheritance.

For example, for this method by inheritance this is the field value, these values and these values are also accessible to any method if we define here. For example, print method or any other method if we want to define here. Now so let us see how we can define all these method very quickly.

(Refer Slide Time: 29:35)

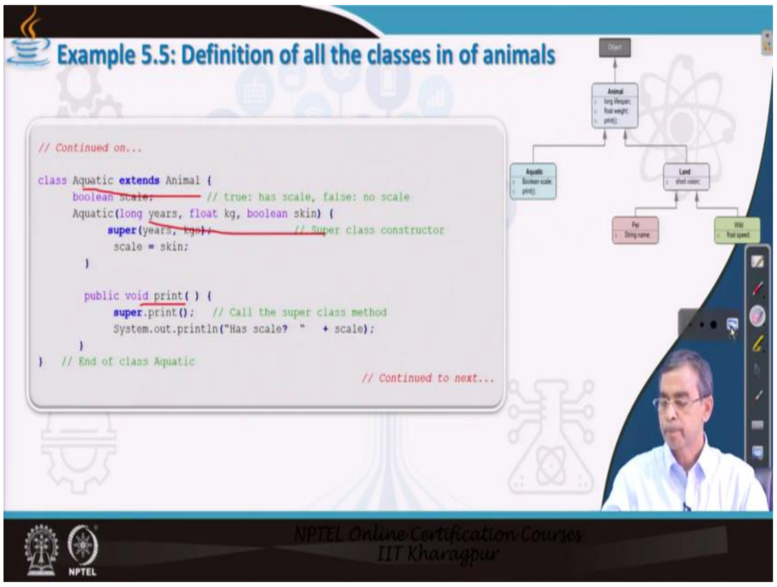


Example 5.5: Definition of all the classes in of animals

```
class Animal {
    long lifespan;
    float weigh;
    Animal(long years, float kg) {
        lifespan = years;
        weight = kg;
    }

    public void print( ) {
        System.out.println("Maximum longevity: " + lifespan + "  in years");
        System.out.println("Maximum weight: "   + weight + "  in kgs");
    }
}   // End of class Animal
                                            // Continued to next...
```

Here is the first examples of defining the different methods in each class. Now this is the example of defining class animal and these are the simple method constructor and this is the print method is a very simple method that we have defined here, so this basically is the declaration of the class animal here, all the methods are very simple to understand so I do not want to discuss this is in details.

(Refer Slide Time: 30:14)

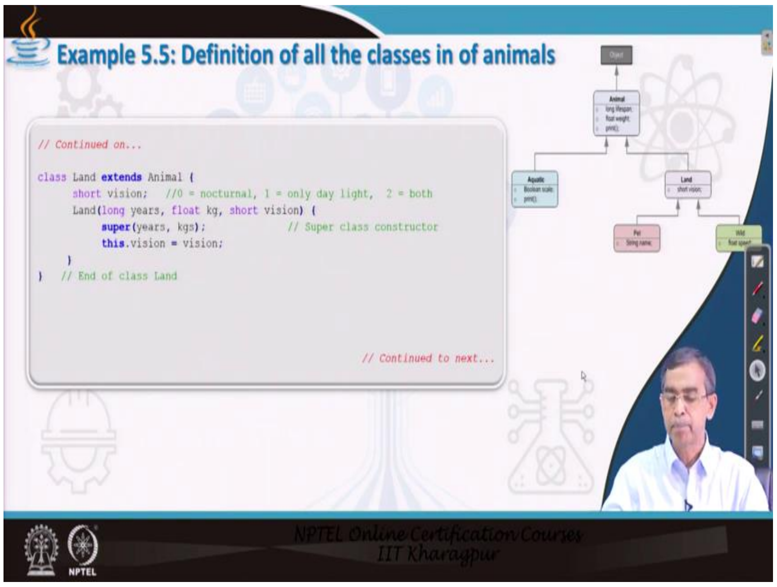


Example 5.5: Definition of all the classes in of animals

```
// Continued on...

class Aquatic extends Animal {
      boolean scale;          // true: has scale, false: no scale
      Aquatic(long years, float kg, boolean skin) {
            super(years, kgs);                  // Super class constructor
             scale = skin;
        }

        public void print( ) {
             super.print();   // Call the super class method
             System.out.println("Has scale?  "   + scale);
        }
}  // End of class Aquatic
                                                    // Continued to next...
```

Now likewise, there is another method so these basically aquatic, aquatic method is here aquatic method is basically extends animal by inheritance and this is the print method that we declared, so it is very simple all these class declaration is simple.

(Refer Slide Time: 30:34)

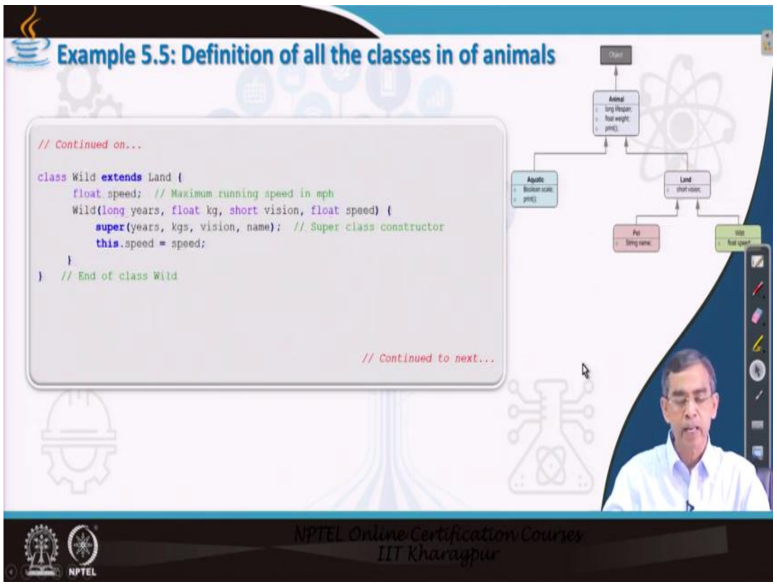


Example 5.5: Definition of all the classes in of animals

```
// Continued on...

class Land extends Animal {
      short vision;   //0 = nocturnal, 1 = only day light,  2 = both
      Land(long years, float kg, short vision) {
           super(years, kgs);                // Super class constructor
           this.vision = vision;
      }
}  // End of class Land

                                                    // Continued to next...
```

Likewise we will declare land, land method, the land extends animal very simple it has basically one simple methods it is there, that is the constructor there is no other methods it is there.

(Refer Slide Time: 30:47)

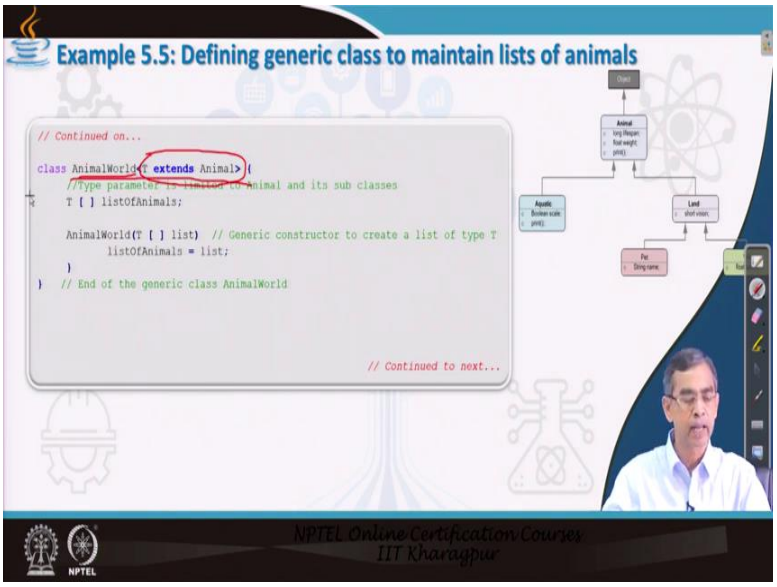


Example 5.5: Definition of all the classes in of animals

```
// Continued on...

class Wild extends Land {
      float speed;  // Maximum running speed in mph
      Wild(long years, float kg, short vision, float speed) {
          super(years, kgs, vision, name);  // Super class constructor
          this.speed = speed;
      }
}   // End of class Wild

                                                  // Continued to next...
```

Now likewise pet and wild method is defined. So all methods are defined, so that mean all classes are defined. Now our objective is to make a program generically where any elements can be stored as a collection, we can create a collection of all pet animals can create a collection of all wild animals land animals or aquatic animals or animals of any type aquatic, land, pet and everything that means generically we can define, so that we can maintain a collection of any type.

(Refer Slide Time: 31:20)

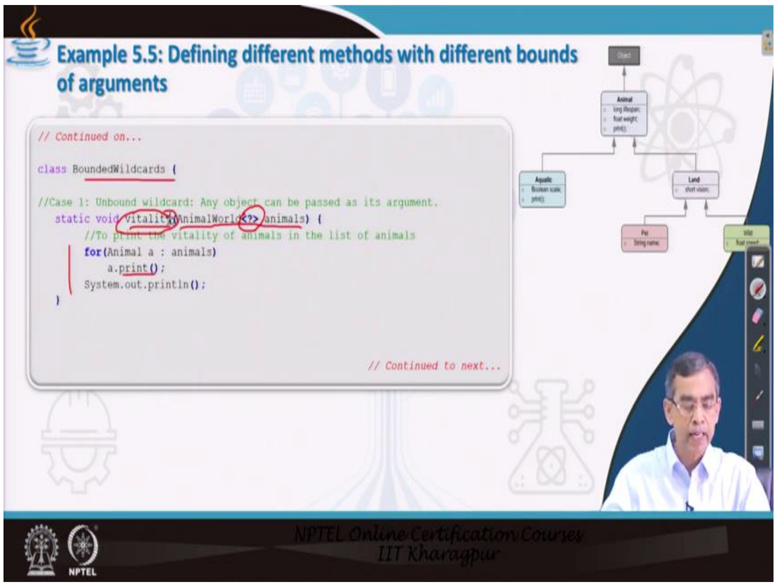


Example 5.5: Defining generic class to maintain lists of animals

```
// Continued on...

class AnimalWorld<T extends Animal> {
      //Type parameter is limited to Animal and its sub classes
      T [ ] listOfAnimals;

      AnimalWorld(T [ ] list)  // Generic constructor to create a list of type T
             listOfAnimals = list;
      }
}   // End of the generic class AnimalWorld

                                                  // Continued to next...
```

Now so this is the one program that we are going to declare using our generic programming concept, now this basically class that we are going to declare is called animal world class and generically with upper bound argument that it has this class is basically has the scope to this class as well as any subclass of this one and these are the basically simple concept of declaration of the generic class. So, this is there and then we want to declare few methods of it, let us see how we can declare the different methods of it.

(Refer Slide Time: 32:01)

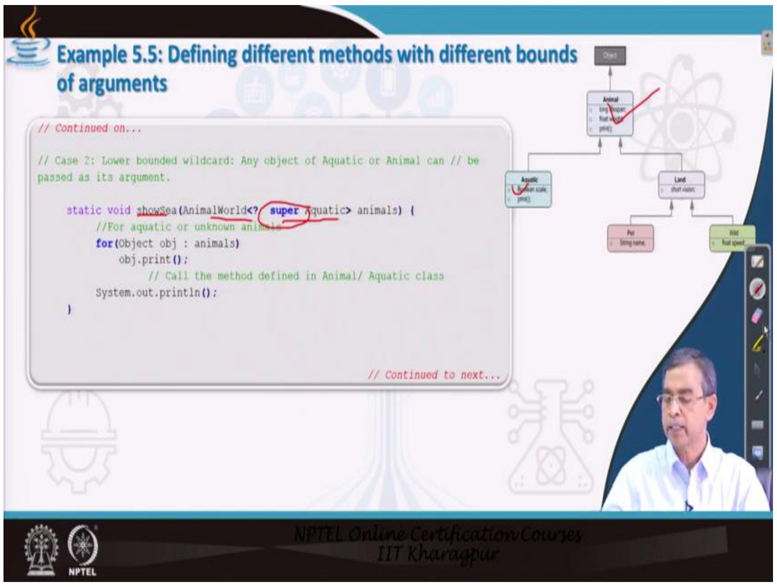


Example 5.5: Defining different methods with different bounds of arguments

```
// Continued on...

class BoundedWildcards {

//Case 1: Unbound wildcard: Any object can be passed as its argument.
   static void vitality(AnimalWorld<?> animals) {
        //To print the vitality of animals in the list of animals
        for(Animal a : animals)
            a.print();
        System.out.println();
   }

                                                   // Continued to next...
```

Now so this is an example this is basically the bounded wild card program, here unbound wild card we want to use and here you can note it, so vitality is the one method we are going to declare here which is belong to this class and this class takes this type the animal world animal world of any type of animals basically and it will basically print, that means for each type of animals it will basically called the print method. So, vitality is the one method which we have declared in this class, so that it can work as an argument of this animal world, animal world can be any animals which is there in this list.

(Refer Slide Time: 32:53)

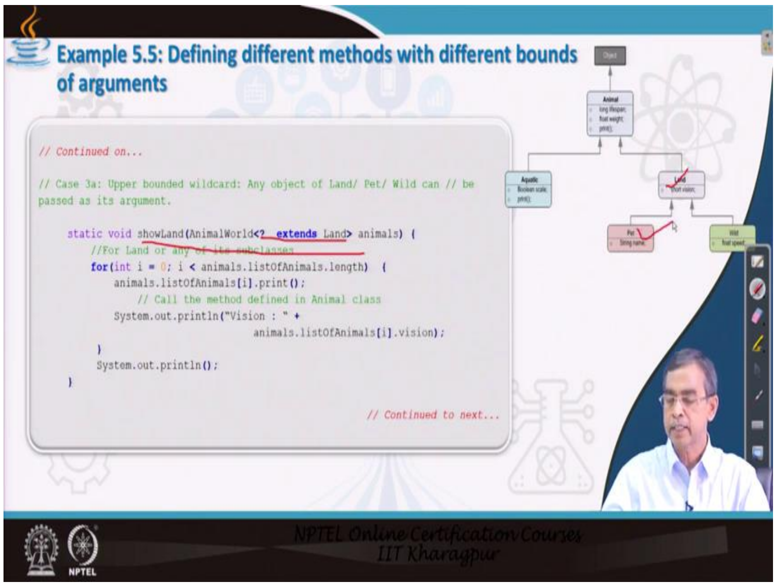


Example 5.5: Defining different methods with different bounds of arguments

```
// Continued on...

// Case 2: Lower bounded wildcard: Any object of Aquatic or Animal can // be
passed as its argument.

    static void showSea(AnimalWorld<? super Aquatic> animals) {
        //For aquatic or unknown animals
        for(Object obj : animals)
            obj.print();
                // Call the method defined in Animal/ Aquatic class
        System.out.println();
    }

                                              // Continued to next...
```

So, this is the one example now let's like vitality we just declare few more method like so see, animal world super aquatic. So, this basically is an example whereas the super means it basically applicable to this method as well as its superclass. So, that is why super thing is there. So, it is a lower bound argument that we have declared.

(Refer Slide Time: 33:20)

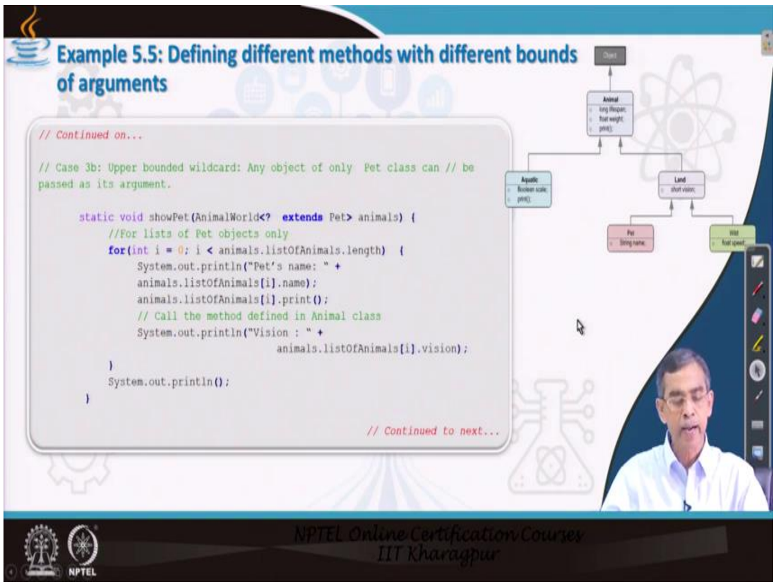


Example 5.5: Defining different methods with different bounds of arguments

```
// Continued on...

// Case 3a: Upper bounded wildcard: Any object of Land/ Pet/ Wild can // be
passed as its argument.

    static void showLand(AnimalWorld<? extends Land> animals) {
       //For Land or any of its subclasses
       for(int i = 0; i < animals.listOfAnimals.length)  {
           animals.listOfAnimals[i].print();
               // Call the method defined in Animal class
           System.out.println("Vision : " +
                                    animals.listOfAnimals[i].vision);
        }
        System.out.println();
    }

                                              // Continued to next...
```

Now similarly there are other methods belongs to show land, so it is basically the method is called show land the same as and is the extend land means it is basically upper bound argument land and it's it this method can call has the scope of this one.

(Refer Slide Time: 33:44)

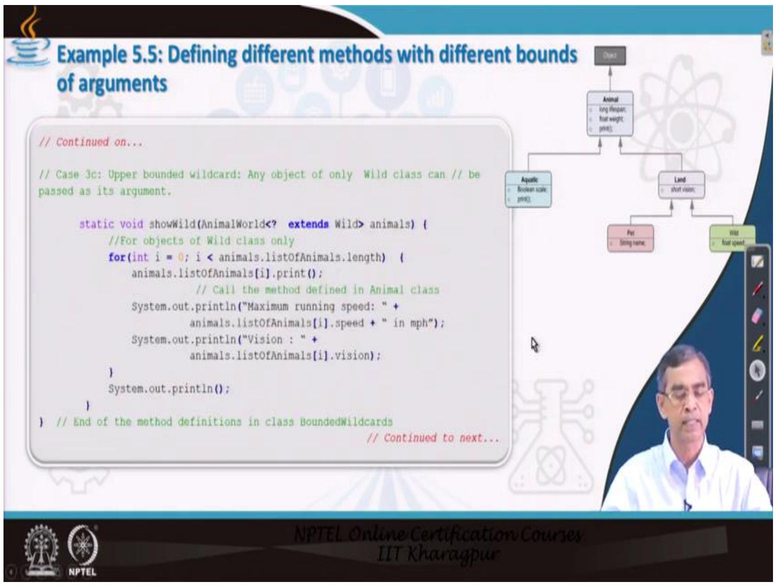


Example 5.5: Defining different methods with different bounds of arguments

```
// Continued on...

// Case 3b: Upper bounded wildcard: Any object of only  Pet class can // be
passed as its argument.

      static void showPet(AnimalWorld<?  extends Pet> animals) {
          //For lists of Pet objects only
          for(int i = 0; i < animals.listOfAnimals.length)  {
              System.out.println("Pet's name: " +
              animals.listOfAnimals[i].name);
              animals.listOfAnimals[i].print();
              // Call the method defined in Animal class
              System.out.println("Vision : " +
                                   animals.listOfAnimals[i].vision);
          }
          System.out.println();
      }

                                                   // Continued to next...
```

Likewise, we can declare other methods namely for pet and wild. So, namely show pet and show while the methods are there. So, you can go through the code of this method you can understand these methods are doing nothing only printing whatever the values are applicable belong to those classes by inheritance as well as on its own definition. So, this is the only thing it is there.

(Refer Slide Time: 34:02)

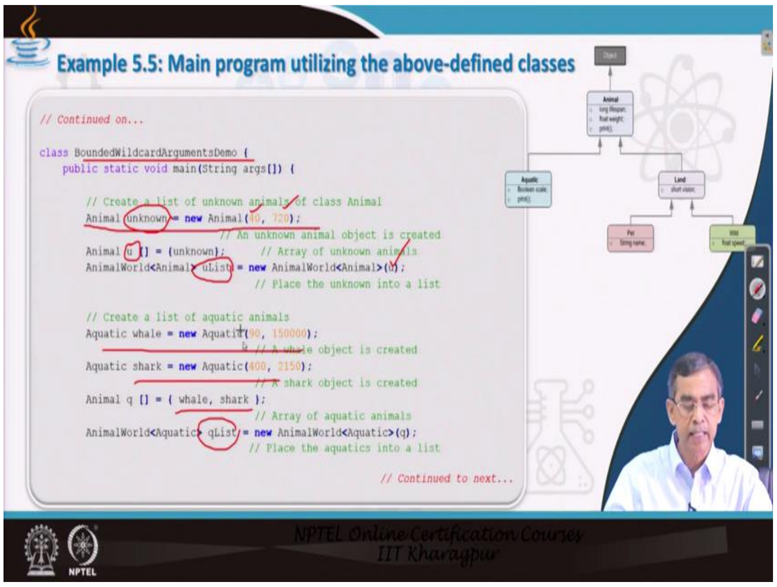


Example 5.5: Defining different methods with different bounds of arguments

```
// Continued on...

// Case 3c: Upper bounded wildcard: Any object of only  Wild class can // be
passed as its argument.

      static void showWild(AnimalWorld<?  extends Wild> animals) {
          //For objects of Wild class only
          for(int i = 0; i < animals.listOfAnimals.length)  {
             animals.listOfAnimals[i].print();
                       // Call the method defined in Animal class
             System.out.println("Maximum running speed: " +
                       animals.listOfAnimals[i].speed + " in mph");
             System.out.println("Vision : " +
                       animals.listOfAnimals[i].vision);
          }
          System.out.println();
      }
}  // End of the method definitions in class BoundedWildcards
                                                   // Continued to next...
```

So this is the, so now here this is the same method for show pets and show wild.

(Refer Slide Time: 34:09)



Example 5.5: Main program utilizing the above-defined classes

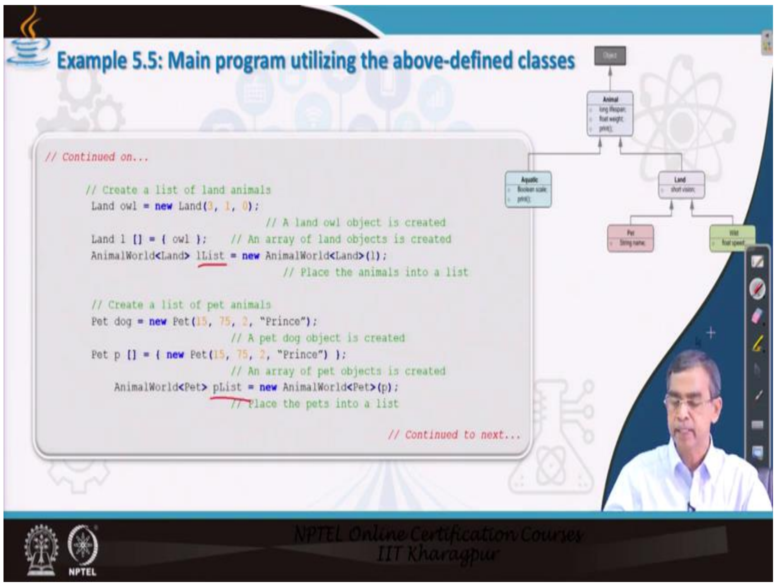
```
// Continued on...

class BoundedWildcardArgumentsDemo {
    public static void main(String args[]) {

        // Create a list of unknown animals of class Animal
        Animal unknown = new Animal(40, 720);
                              // An unknown animal object is created
        Animal u [] = {unknown};        // Array of unknown animals
        AnimalWorld<Animal> uList = new AnimalWorld<Animal>(u);
                                       // Place the unknown into a list

        // Create a list of aquatic animals
        Aquatic whale = new Aquatic(90, 150000);
                                       // A whale object is created
        Aquatic shark = new Aquatic(400, 2150);
                                       // A shark object is created
        Animal q [] = { whale, shark };
                                       // Array of aquatic animals
        AnimalWorld<Aquatic> qList = new AnimalWorld<Aquatic>(q);
                                      // Place the aquatics into a list

                                                  // Continued to next...
```

Now here is the main program this program needs to be checked very carefully and here actually this is the main program driver class. So, bounded wildcard argument demonstration program. Now, here we create a list of animals, the animals is the unknown done on unknown animals.

So, it basically field, animals has a long life span on white, so 40 is a life or it is 720 and we can create an array of animals so only one elements in this array u. So, u is an array of unknown animals and we can create a collection u list using generic class and u is the current list in this one.

Now similarly, we create another arrays what is called the collection of all whales is a shark create them in a array of all the aquatic animals and then create another queue list just like u list for unknowns it is the q list for aquatic.

(Refer Slide Time: 35:17)

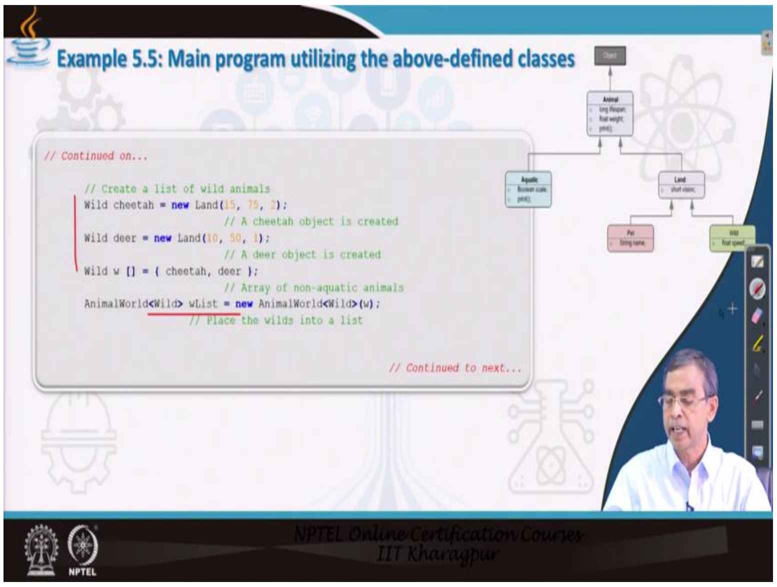


Example 5.5: Main program utilizing the above-defined classes

```
// Continued on...

     // Create a list of land animals
      Land owl = new Land(3, 1, 0);
                                   // A land owl object is created
      Land l [] = { owl };    // An array of land objects is created
      AnimalWorld<Land> lList = new AnimalWorld<Land>(l);
                                      // Place the animals into a list

      // Create a list of pet animals
      Pet dog = new Pet(15, 75, 2, "Prince");
                              // A pet dog object is created
      Pet p [] = { new Pet(15, 75, 2, "Prince") };
                              // An array of pet objects is created
          AnimalWorld<Pet> pList = new AnimalWorld<Pet>(p);
                              // Place the pets into a list

                                                    // Continued to next...
```

Now likewise we can create some the list of animals belongs to each class and this is the same extension the L list is basically animal of land animals and this basically P list is the animal of pet animals.

(Refer Slide Time: 35:31)

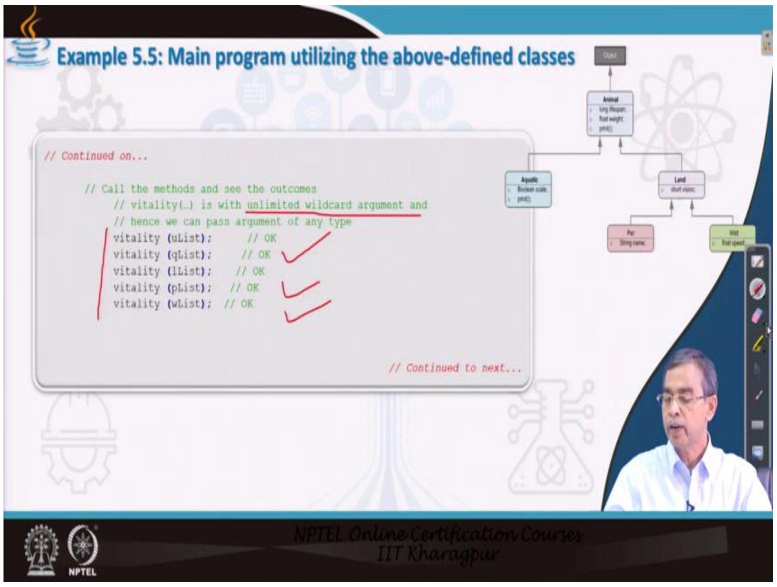


Example 5.5: Main program utilizing the above-defined classes

```
// Continued on...

    // Create a list of wild animals
    Wild cheetah = new Land(15, 75, 2);
                        // A cheetah object is created
    Wild deer = new Land(10, 50, 1);
                        // A deer object is created
    Wild w [] = { cheetah, deer };
                        // Array of non-aquatic animals
    AnimalWorld<Wild> wList = new AnimalWorld<Wild>(w);
                  // Place the wilds into a list

                                         // Continued to next...
```

And likewise this is the list of wild animals. So, w list we can say. So, these are the different wild animals is created and list is created. So, what we have done so far in this generic program, we created a number of collections, see the list of all animals belongs to the different categories like aquatic, land, pet, wild as well as animal. Now let us see if we call the different methods which we have defined with the different wildcard arguments.

(Refer Slide Time: 36:00)

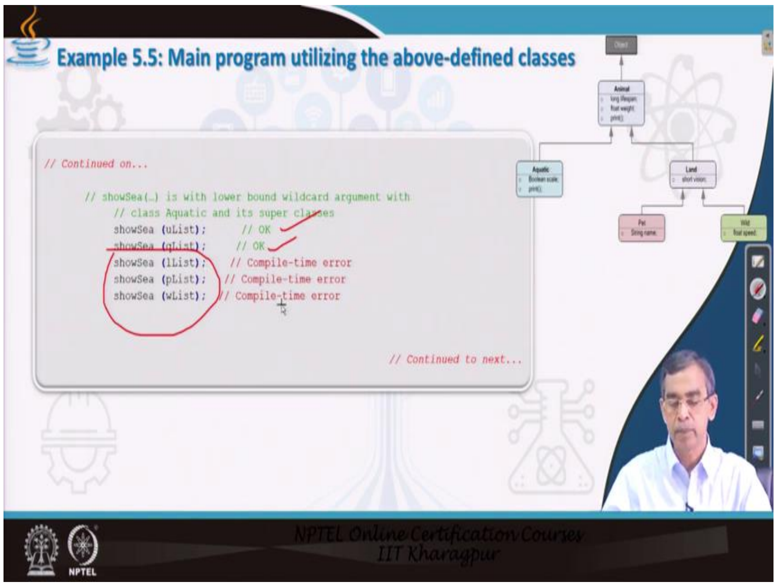


Example 5.5: Main program utilizing the above-defined classes

```
// Continued on...

    // Call the methods and see the outcomes
       // vitality(..) is with unlimited wildcard argument and
       // hence we can pass argument of any type
       vitality (uList);      // OK
       vitality (qList);     // OK
       vitality (lList);    // OK
       vitality (pList);   // OK
       vitality (wList);  // OK

                                         // Continued to next...
```

Now here you see the vitality method we have declared, which we declared with the up unlimited wildcard arguments there, this means that it is applicable to any method whatever it is there. So, all the calls are applicable to this vitality method. However if we call the methods which is declared in each class it may work depending on the arguments or its bounding. Now, here is the one example that you can see.

(Refer Slide Time: 36:37)

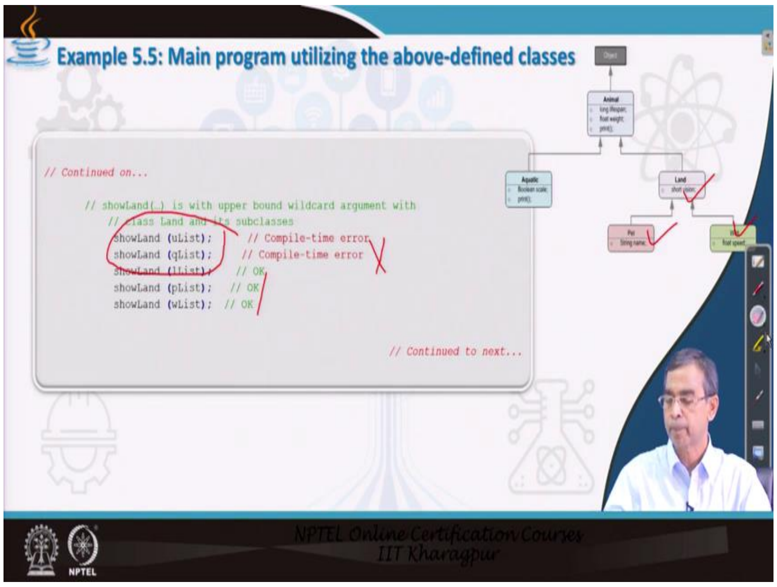


Example 5.5: Main program utilizing the above-defined classes

```
// Continued on...

    // showSea(…) is with lower bound wildcard argument with
       // class Aquatic and its super classes
       showSea (uList);      // OK
       showSea (qList);     // OK
       showSea (lList);    // Compile-time error
       showSea (pList);   // Compile-time error
       showSea (wList);  // Compile-time error

                                        // Continued to next...
```

So, now let us show sea if we apply u list because it is defined there. So, it will work for you, so this is correct q list also because it is defined here it is also super this basically lower bound our argument so it belongs to this class and this one, but it will not work for this kind of list because it is not valid because that bounding is not limited to that.

(Refer Slide Time: 37:07)



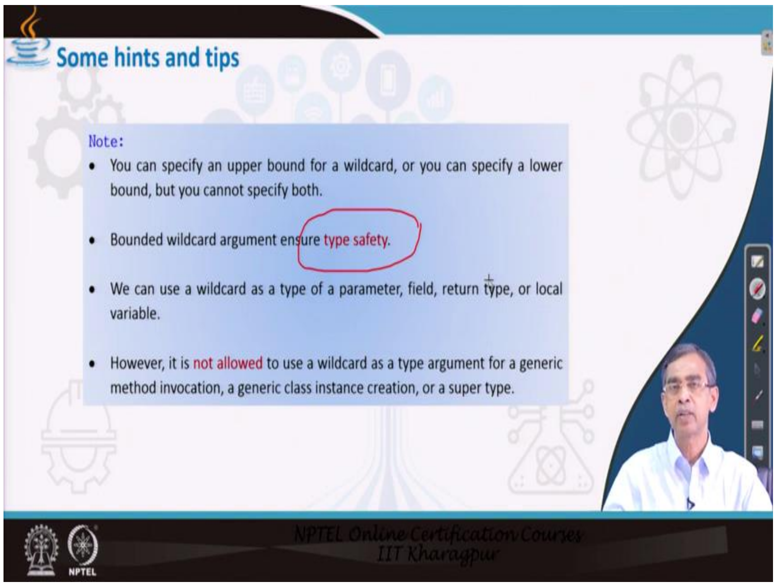
Example 5.5: Main program utilizing the above-defined classes

```
// Continued on...

    // showLand(…) is with upper bound wildcard argument with
      // class Land and its subclasses
       showLand (uList);      // Compile-time error
       showLand (qList);     // Compile-time error
       showLand (lList);    // OK
       showLand (pList);   // OK
       showLand (wList);  // OK

                                        // Continued to next...
```

Now similarly for other method like show land you can check that, for show land has the upper bound wildcard arguments and which is defined in this method, this means that it this method will work for this list as well at all these lists. So, it works for this list however it will not work for other two list the u list and q list because bound is not permissible.

Now finally other two show pet and show wildcard, we have defined at the lower one so it will work for this and this only for others it will not work. Now here is the example as you can see, the show pet is work for the pet list and then show wild is for wild list. Now here we can discuss about the bounding of different arguments or wildcard arguments that means upper bound lower bound and the unbound things are there.

(Refer Slide Time: 38:04)



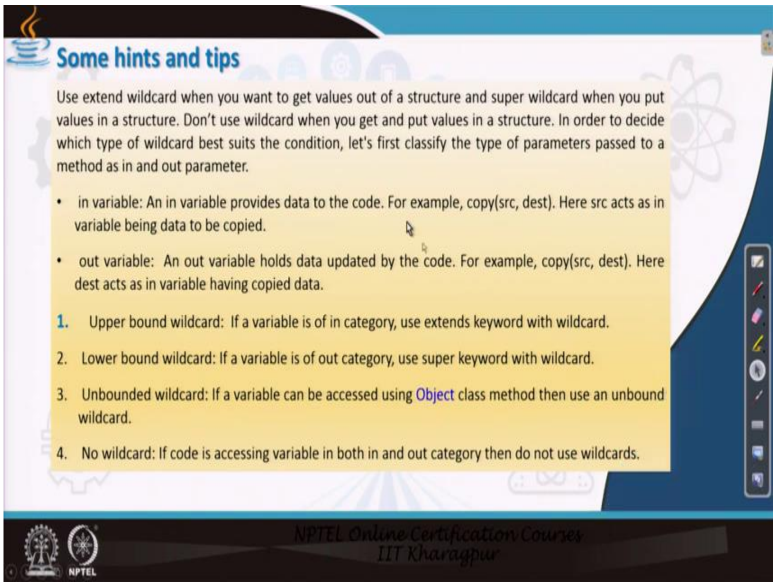
Some hints and tips

Note:

- You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.
- Bounded wildcard argument ensure type safety.
- We can use a wildcard as a type of a parameter, field, return type, or local variable.
- However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a super type.

Now what is exactly the utility of all those bounding is that makes the object or the collection or generic programming to safe to the type, that is why this problem is called type safety. So, all these wildcard bounding arguments or bounded wildcard arguments to ensure type safety and type safety is a very important issue this means that a method, if you want to use in a generic programming you have to have a very clear, what is called in indication that this method has the scope belongs to this or that so that it cannot be allowed to access this method for any illegitimate class. So, that is why this concept it is there.

(Refer Slide Time: 38:54)



Some hints and tips

Use extend wildcard when you want to get values out of a structure and super wildcard when you put values in a structure. Don't use wildcard when you get and put values in a structure. In order to decide which type of wildcard best suits the condition, let's first classify the type of parameters passed to a method as in and out parameter.

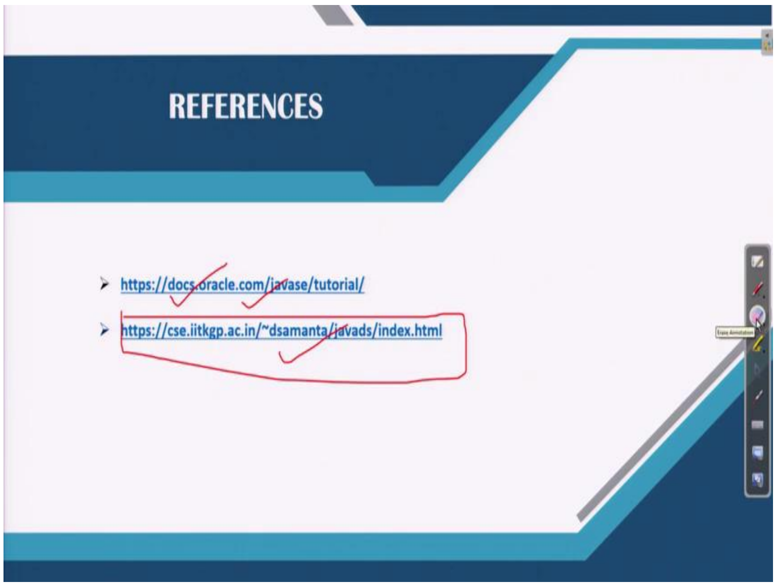
- in variable: An in variable provides data to the code. For example, copy(src, dest). Here src acts as in variable being data to be copied.
- out variable: An out variable holds data updated by the code. For example, copy(src, dest). Here dest acts as in variable having copied data.

1. Upper bound wildcard: If a variable is of in category, use extends keyword with wildcard.
2. Lower bound wildcard: If a variable is of out category, use super keyword with wildcard.
3. Unbounded wildcard: If a variable can be accessed using Object class method then use an unbound wildcard.
4. No wildcard: If code is accessing variable in both in and out category then do not use wildcards.

NPTEL Online Certification Courses
IIT Kharagpur

And with this discussion, I just want to conclude this right there are some it and tip hints and tips you can go through it and then have the clear understanding of it.

(Refer Slide Time: 39:04)



REFERENCES

- https://docs.oracle.com/javase/tutorial/
- https://cse.iitkgp.ac.in/~dsamanta/javads/index.html

And more about the concept that we have discussed in the topic generic programming you can get details material that is there in this, all programs that have been used in this videos lectures you can find it there and also a good documentation about each type of concept it is also discussed in details.

So, that you can go through and then study of your own and if you want to have more discussion about all these things, then definitely this is the one or highly referred materials it is from the Oracle.com you can go through it.

So, this is the idea about generic programming and I would advise you to study little because this concept is bit different and advanced, so a single one round reading may not be sufficient I would advise you to go through several round of reading and all the programs that has been given there you have to just check it try to learn and then run the program. Thank you very much.