Data Structures and Algorithms Using Java

Debasis Samanta

Department of Computer Science & Engineering, IIT Kharagpur

Module 02: Generic Programming

Lecture 03 : Basics of Generic Class

- Concept of Generic Class
- Defining Generic Class
- Examples
  - Defining Generic Class
  - Generic Class with Arrays
  - Generic Class with Abstract Data Type

Concept of Generic Class

Why generic class?

Let us consider the case of processing of an array of any type of numbers using a Java program.

1. Initializing the array.
2. Printing the elements in the array.
3. Reversing the ordering of the elements in the array.

Why generic class?

Here, is the program structure which you should consider, in case the array stores integer numbers.

```
class SpecificArrayInt {
  // Declaring an array of integer values
  // Constructor to load the array.
  // Method to print the array elements.
  // Method to reverse the array elements.
}


class MainClassInt {
      /* This class utilize the class SpecificArrayInt to
manipulate some integer data */
}
```

Example 3.1 : Program to handle an array of integers

```
class SpecificArrayInt {
     // Declaring an array of integer numbers
     int a;

     // Constructor to load the array
     SpecificArrayInt(int a[]) {
         this.a = a;
     }

     // Method to print the array elements
     void printInt() {
         for(int x : a)
             System.out.println(x);
     }
```

// Continued to next page ...

Example 3.1 : Program to handle an array of integers

```
// Continued on ...

   // Method to reverse the array elements
   void reverseInt() {
        j = a.length;
        for (int i=0; i<j; i++)
           int temp;
           temp = a[i];
           a[i] = a[j];
           a[j] = temp;
           j--;
       } // End of for-loop
    } // end of method
} // end of class

                                       // Continued to next page ...
```

Example 3.1 : Program to handle an array of integers

```
// Continued on ...

class MainClassInt {
      //This class use the class SpecificArrayInt to manipulate data in it

      SpecificArrayInt a = {1, 2, 3, 4, 5};

      a.printInt();

      a.reverseInt();

      a.printInt();
}
```

Why generic class?

Now, consider the case of processing a set of double values stored in an array.

```
class SpecificArrayDouble {
  // Declaring an array of double values
  // Constructor to load the array.
  // Method to print the array elements.
  // Method to reverse the array elements.
}


class MainClassDouble {
      /*  This  class  utilize  the  class  SpecificArrayDouble  to
manipulate some integer data. */
}
```

Example 3.2 : Program to handle an array of doubles

```
class SpecificArrayDouble {
     // Declaring an array of double values
     double b;

     // Constructor to load the array
     SpecificArrayDouble(double b[]) {
         this.b = b;
     }

     // Method to print the array elements
     void printDouble() {
         for(double x : b)
             System.out.println(x);
     }
```

// Continued to next page ...

Example 3.2 : Program to handle an array of doubles

```
// Continued on ...

   // Method to reverse the array elements
   void reverseDouble() {
        j = b.length;
        for (int i=0; i<j; i++)
           double temp;
           temp = a[i];
           a[i] = a[j];
           a[j] = temp;
           j--;
       } // End of for-loop
    } // end of method
} // end of class

                                       // Continued to next page ...
```

Example 3.2 : Program to handle an array of doubles

```
// Continued on ...

class MainClassDouble {
      //This class use the class SpecificArrayInt to manipulate data in it

      SpecificArrayDouble b = {1.2, 2.3, 3.4, 4.5, 5.6};

      b.printDouble();

      b.reverseDouble();

      b.printDouble();
}
```

Why generic class?

Let us repeat the procedure, but a set of string elements stored in an array.

```
class SpecificArrayString {
  // Declaring an array of string items.
  // Constructor to load the array.
  // Method to print the array elements.
  // Method to reverse the array elements.
}


class MainClassString {
      /* This class utilize the class SpecificArrayString to
manipulate some integer data. */
}
```

Example 3.3 : Program to handle an array of Strings

```java
class SpecificArrayString {
    // Declaring an array of double values
    String c;

    // Constructor to load the array
    SpecificArrayDouble(String c[]) {
        this.c = c;
    }

    // Method to print the array elements
    void printString() {
        for(String x : c)
            System.out.println(x);
    }
```

// Continued to next page ...

Example 3.3 : Program to handle an array of Strings

```
// Continued on ...

   // Method to reverse the array elements
   void reverseString() {
        j = c.length;
        for (int i=0; i<j; i++)
           double temp;
           temp = c[i];
           c[i] = c[j];
           c[j] = temp;
           j--;
       } // End of for-loop
    } // end of method
} // end of class

                                                       // Continued to next page ...
```

Example 3.3 : Program to handle an array of Strings

```
// Continued on ...

class MainClassString {
      //This class use the class SpecificArrayInt to manipulate data in it

      SpecificArrayDouble b = {“A”, “B”, “C”, “D”, “E”};

      c.printString();

      c.reverseString();

      c.printString();
}
```

Why generic class?

- **Observations**
  - Data are different.
  - Algorithms (i.e., logic) are same for all methods.
  - Different programs.
  - Code duplications

Syntax of Defining Generic Class

Syntax for generic class definition

The syntax for declaring a generic class is as follows:

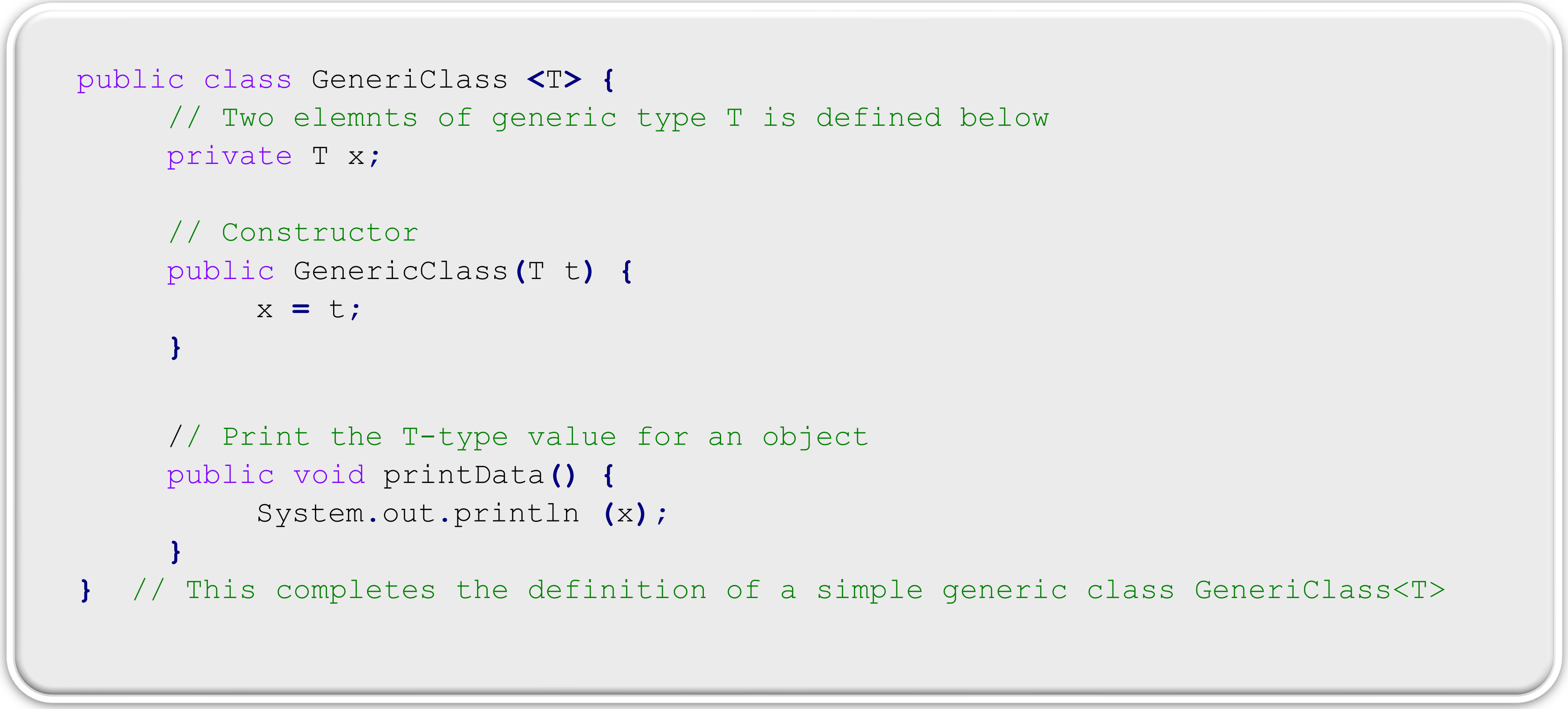
```
[<<Access] class <ClassName> <<Type1> [, <Type2>, …]> {
           … body of the class
}
```

Here, is the full syntax for declaring a reference to a generic class and instance creation:

```
<className><typeList> varName = new <className><typeList> (<InputArray>);
```

Example: Defining a Generic Class

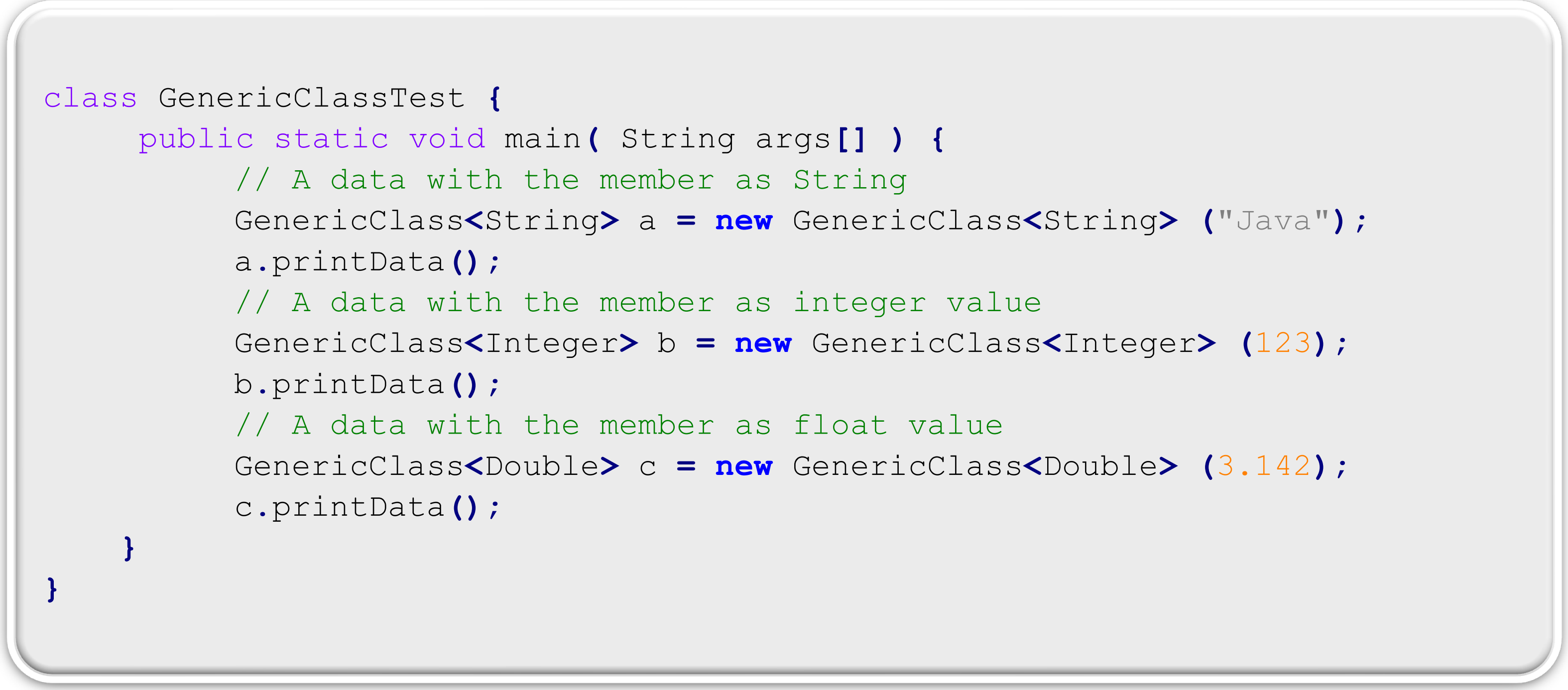
Example 3.4 : Defining a generic class



```
public class GeneriClass <T> {
     // Two elemnts of generic type T is defined below
     private T x;

     // Constructor
     public GenericClass(T t) {
          x = t;
     }

     // Print the T-type value for an object
     public void printData() {
          System.out.println (x);
     }
}  // This completes the definition of a simple generic class GeneriClass<T>
```

Example 3.4: Using the defined GeneriClass<T>

The driver class is programmed below, which creates different types of objects.



```java
class GenericClassTest {
     public static void main( String args[] ) {
          // A data with the member as String
          GenericClass<String> a = new GenericClass<String> ("Java");
          a.printData();
          // A data with the member as integer value
          GenericClass<Integer> b = new GenericClass<Integer> (123);
          b.printData();
          // A data with the member as float value
          GenericClass<Double> c = new GenericClass<Double> (3.142);
          c.printData();
    }
}
```

Example: Defining a Generic Class with Array of Any Data Type

Example 3.5: Processing arrays with any data type

```
class GenericArrayClass {
     // Declaring a generic array
     // Constructor to load the array.
     // Method to print the array elements
     // Method to reverse the array elements
}

class MainClass {
      //This class utilize the class GenericArrayClass to manipulate data of any type.
}
```

Example 3.5: Defining class to process arrays with any data type

```
class GenericArray<T> {
   //Declaring an array, which should store any type T of data
   T a[ ];

   GenericArray(T x) {        // Define a constructor
      a = x;
   }

   T getData(int i) {    // To return the element stored in the i-th place in the array
       return a[i];
   }

   void static printData (T b) {  // A generic method to print the elements in array b
      for(int i = 0; i < b.length(); i ++)
          System.out.print(b.getData(i) + “  ”); // Print the i-th element in b
      System.out.println();      // Print a new line
   }
```

Example 3.5: Defining class to process arrays with any data type

```
    void static reverseArray (T b) { // Generic method to reversed the order of elements in array b
        int front = 0, rear = b.length-1;
        T temp;
        while( front < rear)  {
            temp = b[rear];
            b[rear] = a[front];
            a[front] = temp;
            front++; rear--;
        }
    }
}
```

Example 3.5: Defining class to process arrays with any data type

```
class GenericClassArrayDemo {
     public void static main(String args a[]) {
        //Creating an array of integer data
        Integer x[ ] = {10, 20, 30, 40, 50};      // It is an array of Integer numbers

        // Store the data into generic array
        GenericArray<Integer> aInt = new GenericArray<Integer>(x);
        // Alternatively:
        // GenericArray<Integer> aInt = new GenericArray<Integer>(new Integer x[ ] {10, 20, 30, 40, 50});

       // Printing the data ...
       printData(aInt);          // Printing the array of integer objects

       //Reverse ordering of data ...
       reverseArray(aInt);

       // Printing the data after reverse ordering ...
       printData(aInt);          // Printing the array of integer objects after reversing

                                                                             // Continued to next page ...
```

Example 3.5: Defining class to process arrays with any data type

```
// Continued on ...

       //Creating an array of integer data
       String y[ ] = {“A”, “B”, “C”, “D”, “E”};      // It is an array of String objects

       // Store the data into a generic array
       GenericArray<String> bString = new GenericArray<String>(y);


       // Printing the data ...
       printData(bString);          // Printing the array of string objects

       //Reverse ordering of data ...
       reverseArray(bString);

       // Printing the data after reverse ordering ...
       printData(bString);          // Printing the array of string objects after reversing

                                                                            // Continued to next page ...
```

Example 3.5: Defining class to process arrays with any data type

```
// Continued on ...

       //Creating an array of double data
       Double z[ ] = {1.2, 2.3, 3.4, 4.5, 5.6};      // It is an array of double values

       // Store the data into a generic array
       GenericArray<Double> cDouble = new GenericArray<Double>(z);


       // Printing the data ...
       printData(cDouble);          // Printing the array of double values

       //Reverse ordering of data ...
       reverseArray(cDouble);

       // Printing the data after reverse ordering ...
       printData(cDouble);          // Printing the array of double values after reversing
   }  // End of main method
} // End of GenericArrayClassDemo class
```

Example: Defining a Generic Class with User Defined Data Type

Example 3.6: Working with user defined class

```
//Define a class Student
class Student {
   String name;
   float marks;

   Student(String name, float marks) {
      this.name = name;
      this.marks = marks;
}
```

```
class GenericClass<T>  {      // Use < > to specify class type
    T obj;                    // An object of type T is declared
    GenericClass(T obj) {     // Constructor of the generic class
        this.obj = obj;
    }
    public T getObject()  {   // A Method in the class
        return this.obj;
    }
}
```

Example 3.6: Working with user defined class

```
class GenericClassTest  {     // Driver class to test generic class facility
    public static void main (String[] args) {
        GenericClass <Integer> iObj = new GenericClass <Integer>(15);
                              // A class with Integer type
        System.out.println(iObj.getObject());

        GenericClass <String> sObj  = new GenericClass <String>("Java");
                             // Another class with String type
        System.out.println(sObj.getObject());

        GenericClass <Student> tObj = new GenericClass <Student>(”Debasis”, 99.9);
                             // Another class with Student type
        System.out.println(tObj.getObject());
    }
}
```

https://cse.iitkgp.ac.in/~dsamanta/javads/index.html

https://nptel.ac.in/noc/faqnew.php

Data Structures and Algorithms Using Java

Debasis Samanta

Department of Computer Science & Engineering, IIT Kharagpur

Module 02: Generic Programming

Lecture 04 : Parameterized Generic Class

- Important Points
- Generic Class with Multiple Type Parameters
  - Syntax of Defining Advanced Generic Class
  - Examples

Important Notes

Features in generic programming

- In the previous lectures, we have shared our programing experience, when we want to process data using same logic, but are of different types.

- The generic feature in Java is a right solution to get rid-off code duplicities.

Important Notes

1. You cannot instantiate an array whose element type is a type parameter. That is following is invalid.

   ```
   T a = new T[5];
   ```

   The reason is that you can’t create an array of T is that there is no way for the compiler to know what type of array to actually create.

Important Notes

2. A generic method or any method in a generic class can be declared as static.

Example 4.1

```
class GenericStaticDemo  {
   // Defining a static generic method to print any data type
    static void gPrint (T t)  {
         System.out.println (t);
   }

   public static void main(String[] args)    {
       gPrint(101);              // Calling generic method with Integer argument

       gPrint("Joy with Java"); // Calling generic method with String argument

       gPrint(3.1412343);       // Calling generic method with double argument
   }
}
```

Important Notes

3. A generic method or any method in a generic class can be declared as static.

Example 4.2

```
 class GenericClass<T>  {       // Use < > to specify class type
    T obj;                     // An object of type T is declared
    GenericClass(T obj) {      // Constructor of the generic class
        this.obj = obj;
    }
    public void static print(T obj)  {    // A method in the class
        System.out.println(obj);
    }
}

class GenericStaticDemo2 {
     public void static main(String args a[]) {
          GenericClass<Integer> a = new GenericClass<Integer>(new Integer x[ ] {10, 20, 30, 40, 50});

          GenericClass<String> s = new GenericClass<String>("Joy with Java");

          GenericClass<Double> d = new GenericClass<Double>(1.23);

          // Printing the data ...
          print(a);          // Printing the array a
          print(s);          // Printing the string
          print(d);          // Printing the value
    }
}
```

Important Notes

4. In parameter type, you can not use primitives type like `int`, `char`, `double`, etc. Only class can be referred as the template data.

Example 4.3

```
 class GenericClass<T>  {      // Use < > to specify class type
    T obj;                    // An object of type T is declared
    GenericClass(T obj) {     // Constructor of the generic class
        this.obj = obj;
    }
}
class GenericClassDemo3 {
     public void static main(String args a[]) {
          GenericClass<Integer> a = new GenericClass<Integer>(123);  // Okay
          GenericClass<int> a = new GenericClass<int>(234);          // ERROR!

          GenericClass<String> s = new GenericClass<String>("Joy with Java"); // Okay

          GenericClass<double> d = new GenericClass<double>(9.87); // ERROR!
          GenericClass<Double> d = new GenericClass<Double>(1.23); // Okay

    }
}
```

Important Notes

5. A generic class can be defined with multiple type parameters.

Example 4.4

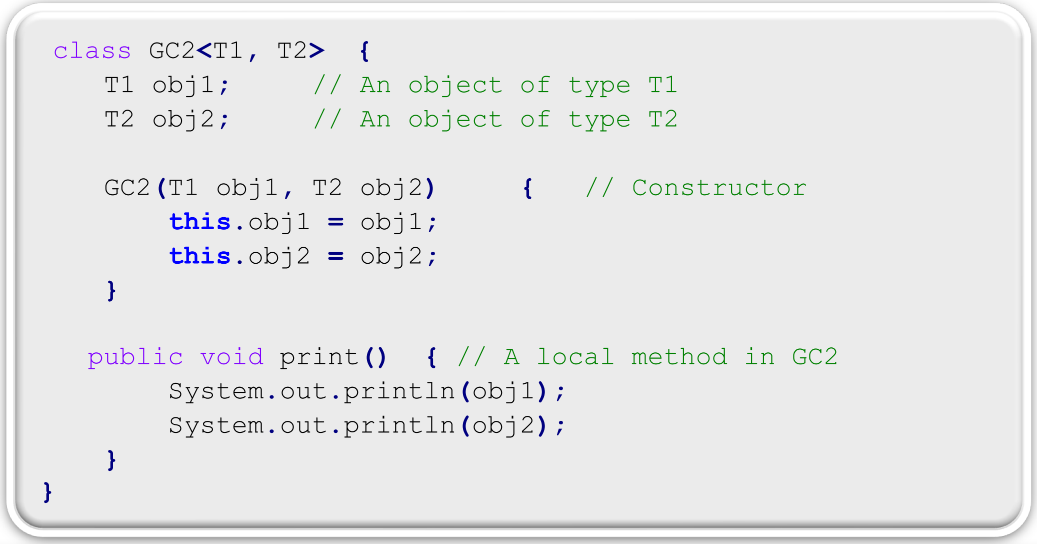
```
class GenericClass<T, V>  {      // Use < > to specify class type
    T obj1;                     // An object of type T is declared
    V obj2;                     // An object of type V is declared

    // Constructor of the generic class
    GenericClass(T obj1, V obj2) {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }
}
```

Generic Class with Multiple Type Parameters

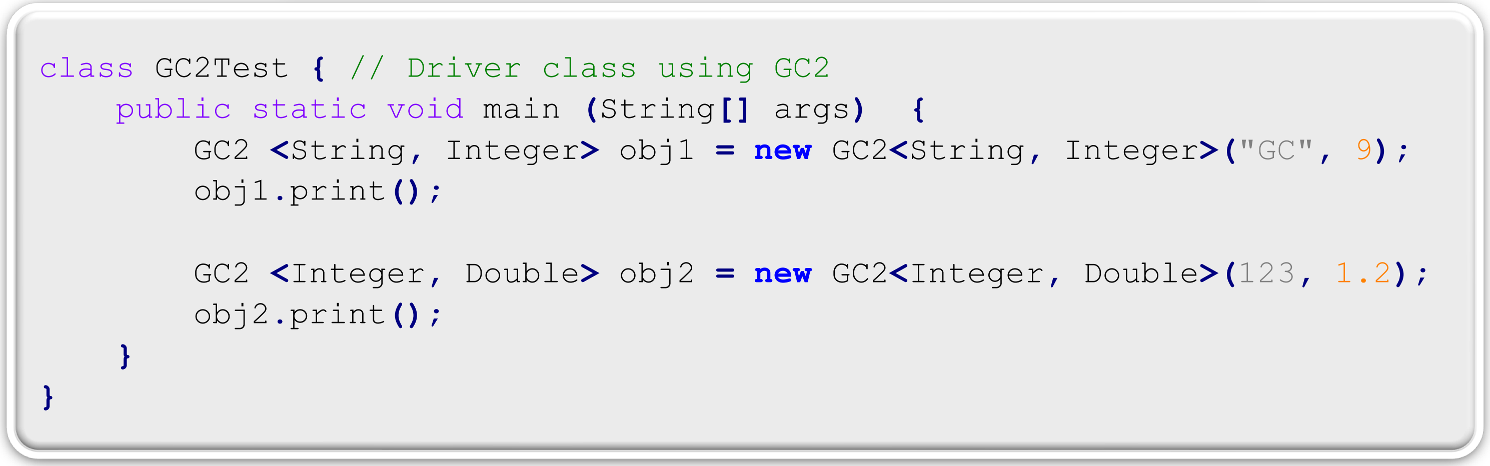
Concept

- We can create a generic class with one or more type parameters so that more than one types of data to be manipulated in a generic program.

Example 4.5: Generic class with two parameters



```java
class GC2<T1, T2>  {
   T1 obj1;     // An object of type T1
   T2 obj2;     // An object of type T2

   GC2(T1 obj1, T2 obj2)     {   // Constructor
       this.obj1 = obj1;
       this.obj2 = obj2;
   }

  public void print()  { // A local method in GC2
       System.out.println(obj1);
       System.out.println(obj2);
   }
}
```



```java
class GC2Test { // Driver class using GC2
    public static void main (String[] args)  {
        GC2 <String, Integer> obj1 = new GC2<String, Integer>("GC", 9);
        obj1.print();

        GC2 <Integer, Double> obj2 = new GC2<Integer, Double>(123, 1.2);
        obj2.print();
    }
}
```

Example 4.6: Another generic class with two parameters

```
public class PairData <T, V> {
     // Two fields of generic type T and V
      private T x;
      private V y;        // Note: How a field can be defined generically.

   // Constructor
      public PairData(T a, V b) {
                x = a;
                y = b;
     }

      // Get the T-type value for a pair-data
      public T getTvalue() {
          return x;
     }

      // Get the V-type value for a pair-data
      public V getVvalue() {
          return y;
     }

    // To print the data member in an object
     public void printData() {
             System.out.println (getTvalue + “,” getVvalue);
    }
}  // This completes the definition of the class PairData<T, V>

                                                         // Continued to next ...
```

Example 4.6: Another generic class with two parameters

```
// Continued on ...

 // The driver class is programmed below.
 class  MultiParamtereGenericClassTest {
       public static void main( String args[] ) {
            // A pair data with both members as String
            PairData<String, String> a = new PairData<String, String> (“Debasis”, “Samanta”);
            a.printData();

            // A pair data with the first member as String and other as Integer
            PairData<String, Integer> b = new PairData<String, Integer> (“Debasis”, 789);
            b.printData();

            // A pair data with the first member as Integer and other as String
            PairData<Integer, String> c = new PairData<Integer, String> (943, “Samanta”);
            c.printData();

            // A pair data with the first member as Integer and other as Double
            PairData<Integer, Double> d = new PairData<Integer, Double> (555, 12.34);
            d.printData();
      }
}
```

Example 4.7: Generic class with method overloading

```
 // Define the user defined Student class
class Student {
   String name;   // Name of the students
   int marks[3];   // Stores the marks in three subjects

   // Constructor for the class Student
   Student(String s, int m[ ]) {
       name = s;
       marks = m;
   }

   //Defining a method to print student’s record
    void printStudent() {
        System.out.println(“Name : ‟ + name);
        System.out.println(“Scores : ‟ + marks[0] + ‟  ‟ + marks[1] + ‟  ‟ + marks[2] );
    }
 }     // End of the class Student

                                                                                   // Continued to next ...
```

Example 4.7: Generic class with method overloading

```
// Continued on...

 // Defining a generic array with two type parameters
 class GenericMultiArrays<T, S> {
      //Declaring an array, which should store any type T of data
      T a[ ];    // Define that the array a[ ] can store one type of data
      S b[];     // Define that the array b[ ] can store another type of data
      GenericArrays(T x, S y) {         // Define a constructor
          a = x;
          b = y;
      }

      T getDataT(int i) {// To return the element stored in i-th place in the array
           return a[i];
      }

      S getDataS(int i) { //To return the element stored in i-th place in the array
           return b[i];
      }

                                                                   // Continued to next ...
```

Example 4.7: Generic class with method overloading

```
// Continued on...

       // Overloaded methods in the generic class
       void printData (T t) {          // A generic method to print the elements in array t
            for(int i = 0; i < t.length(); i ++)
                   System.out.print(t.getData(i) + “  ”); //Print the i-th element in t
            System.out.println();          // Print a new line
       }

       void printData(S s){          //An overloaded generic method to print elements in s
           for(int i = 0; i < s.length(); i ++)
                 s[i].printStudent()           // Print the i-th student in s
           System.out.println();              // Print a new line
       }

                                                                            // Continued to next ...
```

Example 4.7: Generic class with method overloading

```
// Continued on...

   // Few additional methods
   void reverseArray(T t) {    //Generic method to reverse the order of elements in t
     int front = 0, rear = t.length-1; T temp;
     while( front < rear)  {
       temp = t[rear];
       t[rear] = t[front];
       t[front] = temp;
       front++; rear--;
     }
   }
   void reverseArray(S s){//Generic method to reverse the order of elements in s
   int front = 0, rear = s.length-1; S temp;
   while( front < rear)  {
     temp = s[rear]
     s[rear] = s[front];
     s[front] = temp;
     front++; rear--;
   }
  }
}      // End of the definition of class GenericMultiArrays

                                                                // Continued to next ...
```

Example 4.7: Generic class with method overloading

```
// Continued on...

      // Driver class is programmed below
      Class GenericMultiArraysDemo {
           public void static main(String args a[]) {
        //Creating an array of String data
        String t[ ] = {“A”, “B”, “C”};      // It is an array of String data

       //Creating an array of Students’ data
        Student s[3];     // It is an array of String data
        s[0] = new Student(“Ram”, 86, 66, 96);
        s[1] = new Student(“Rahim”, 88, 99, 77);
        s[2] = new Student(“John”, 75, 85, 95);

       // Store the data into generic arrays
       GenericArrays<String, Student> arrayData = new GenericArrays<String,
                                                             Student>(t, s);

                                                                   // Continued to next ...
```

Example 4.7: Generic class with method overloading

```
// Continued on...


      // Printing the data ...
       arrayData.printData(t);          // Printing the array of strings

       //Reverse ordering of data ...
       arrayData.reverseArray(t);

       // Printing the data ...
       arrayData.printData(s);          // Printing the student’s data

       //Reverse ordering of data ...
       arrayData.reverseArray(s);

       // Printing the data after reverse ordering ...
       arrayData.printData(t);          // Printing the array of strings

       arrayData.printData(s);          // Printing the array of students
   }
}
```

Important Notes

6. If a class A is declared as generic with type parameter <T>, then object of class can be created any type. This is fine, but it may causes in several situation error during execution.

Example 4.8

```
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 a compiler error!
        return sum / array.length;
    }
}
```

Important Notes

7. If a class A is declared as generic with type parameter <T>, then object of class can be created any type. This is fine!. But, in several situations, it may cause errors during execution.

    - Here, you know that the method `doubleValue( )` is well defined for all numeric classes, such as `Integer`, `Float` and `Double`, which are the sub classes of `Number`, and `Number` defines the `doubleValue( )` method. Hence, this method is available to all numeric wrapper classes.

    - Further, you note that you can create object of the class `GenericError` with some type parameter for which there is no method defined `doubleValue()`. In other words, the compiler does not have any knowledge about that you are only interested to create objects of numeric types. Thus, the program reports compile-time error showing that the `doubleValue( )` method is unknown.

    - To solve this problem, you need some way to tell the compiler that you intend to pass only numeric types to `T`. Furthermore, you need some way to ensure that only numeric types are actually passed.

https://cse.iitkgp.ac.in/~dsamanta/javads/index.html

https://nptel.ac.in/noc/faqnew.php

Data Structures and Algorithms Using Java

Debasis Samanta

Department of Computer Science & Engineering, IIT Kharagpur

Module 02: Generic Programming

Lecture 05 : Bounded Argument Generic Class

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

Bounded Types in Generic Class

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.

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.

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 ...
```

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);                 */
      }
}
```

Wildcard in Java Generics

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.

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 ...
```

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 ...
```

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 ...

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.");
      }
}
```

Example 5.3: Generic class with another limitation

Notes:

1. There is no error when s1 is compared with s2;

2. The same is not true for s2 and s3 or s3 and s4. The reason is that the `si.compareMarks (sj)` method works only when the type of the object `sj` is same as the invoking object `si`.

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)`.

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 ...
```

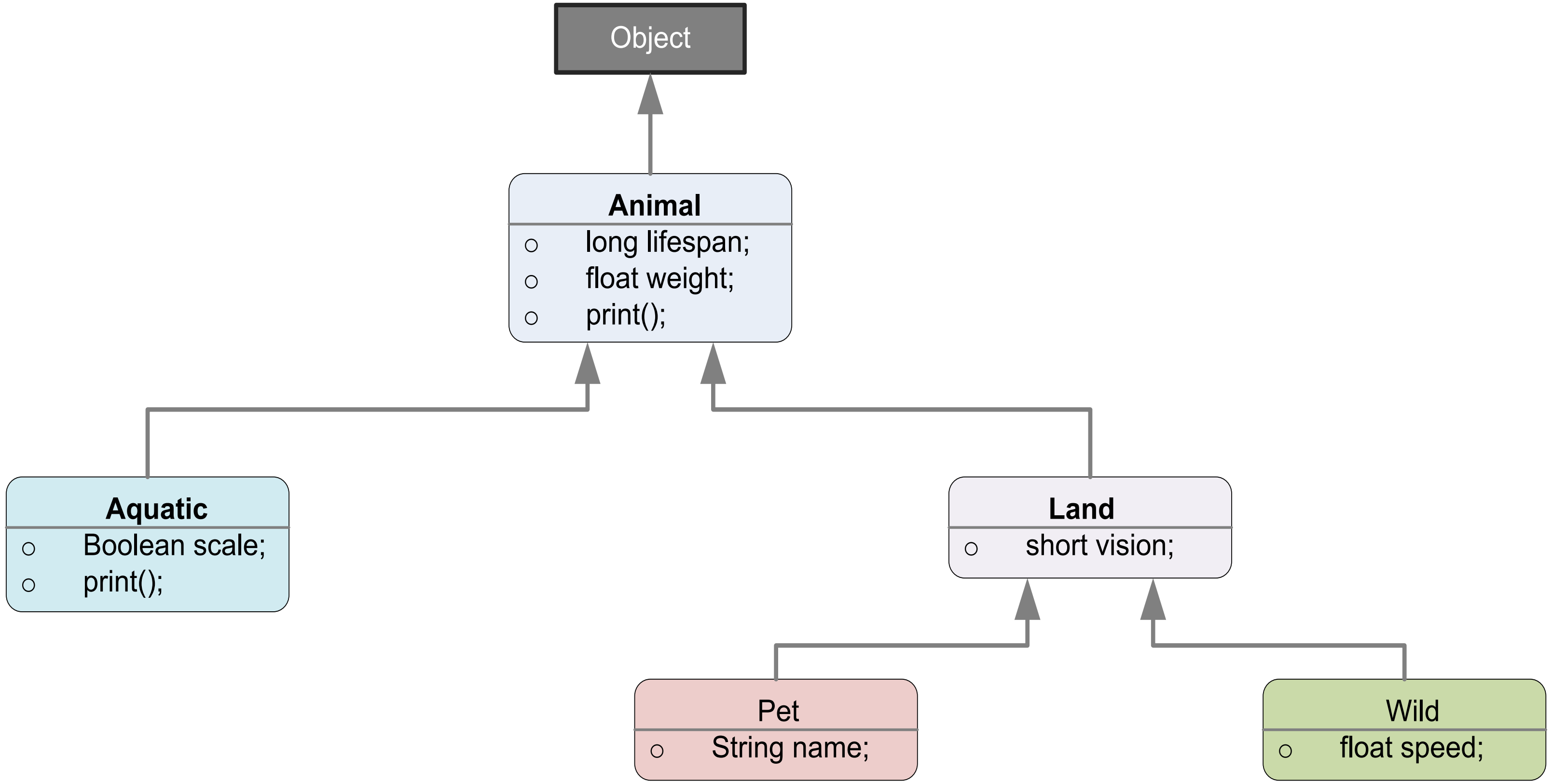
Example 5.4: Solution of the limitation with wildcard

```
// 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.

Bounded Wildcard Arguments

The concept



Object

Animal

- ○ long lifespan;
- ○ float weight;
- ○ print();

Aquatic

- ○ Boolean scale;
- ○ print();

Land

- ○ short vision;

Pet

- ○ String name;

Wild

- ○ float speed;

Bounded wildcard arguments

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

1. Upper bound wildcard

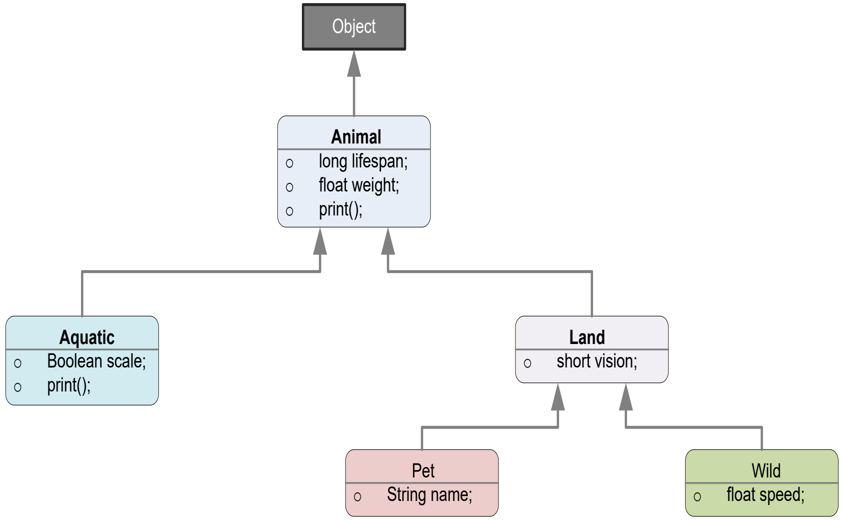
These wildcards can be used when you want to write a method that works on the class where it is defined or any of its sub class.

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.



Bounded wildcard arguments

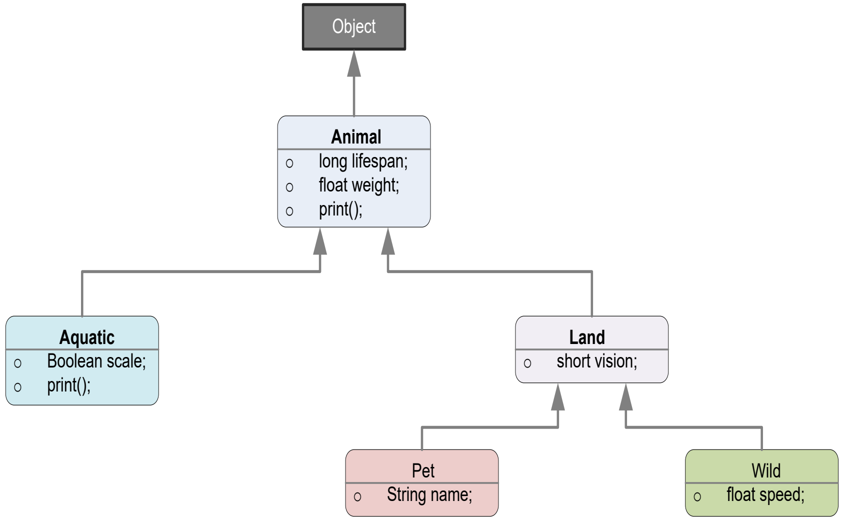
2. Lower bound wildcard

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.

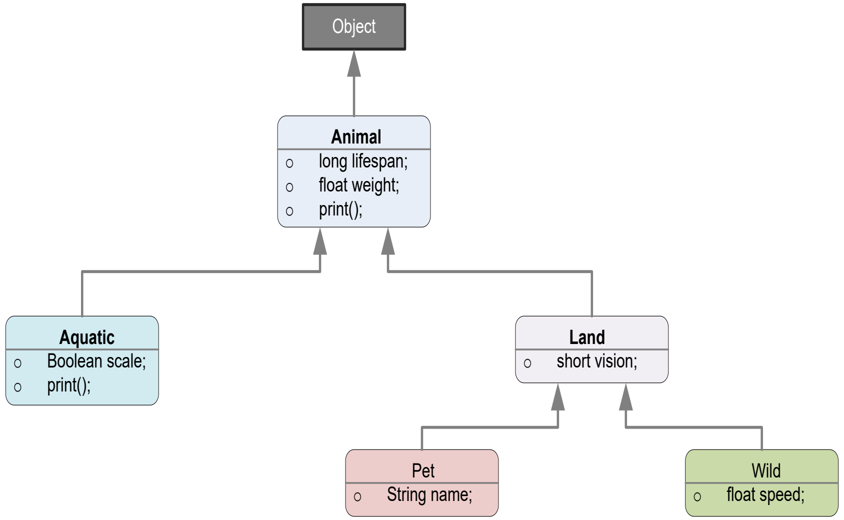
Syntax:

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

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



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(?) { … }
```

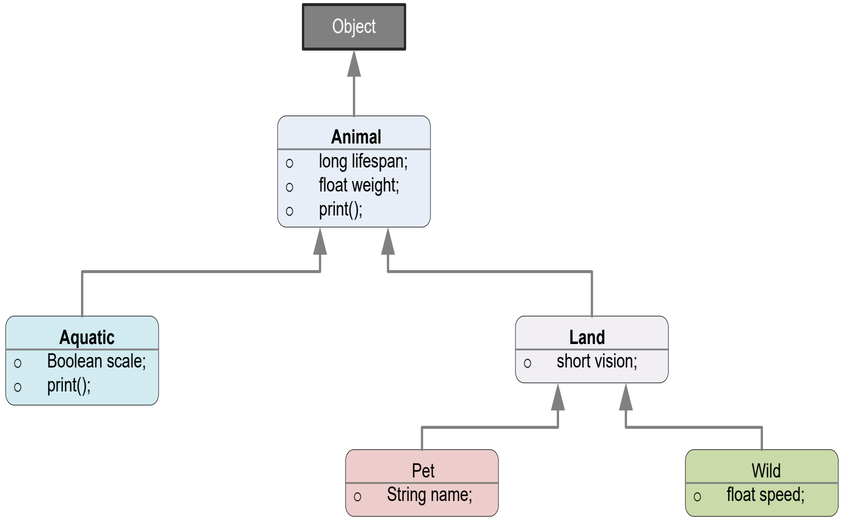
Examples of Bounded Wildcard

Example 5.4: Bounded wildcard arguments

The objective of this example is to illustrate how different methods can be defined with different bounder wildcard arguments.

A program is given which consists of the following parts.

1. Definition of all the classes as shown in the figure.
2. Declaration of the generic class, which can be used to store different lists of animals.
3. Definitions of different methods to handle objects of different classes in the class hierarchy.
4. Driver class to manipulate the objects of different types.

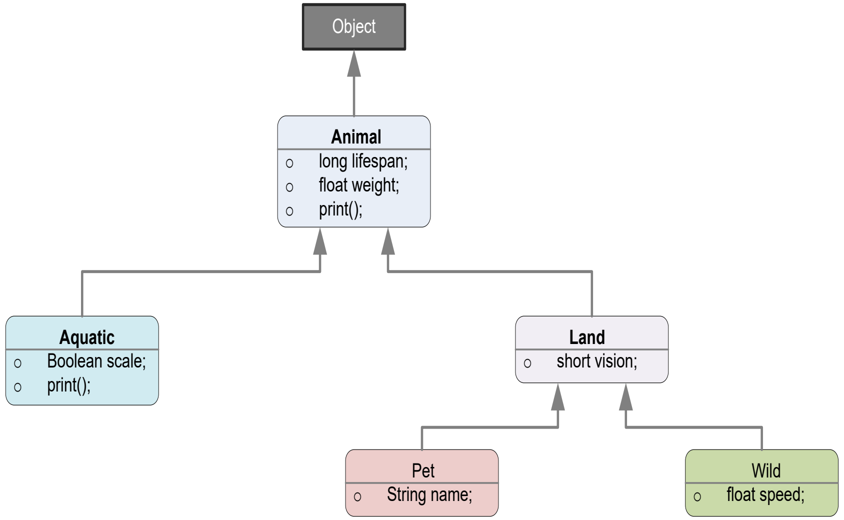


Example 5.4: Definition of all the classes in animals

The objective of this example is to illustrate how different methods can be defined with different bounder wildcard arguments.

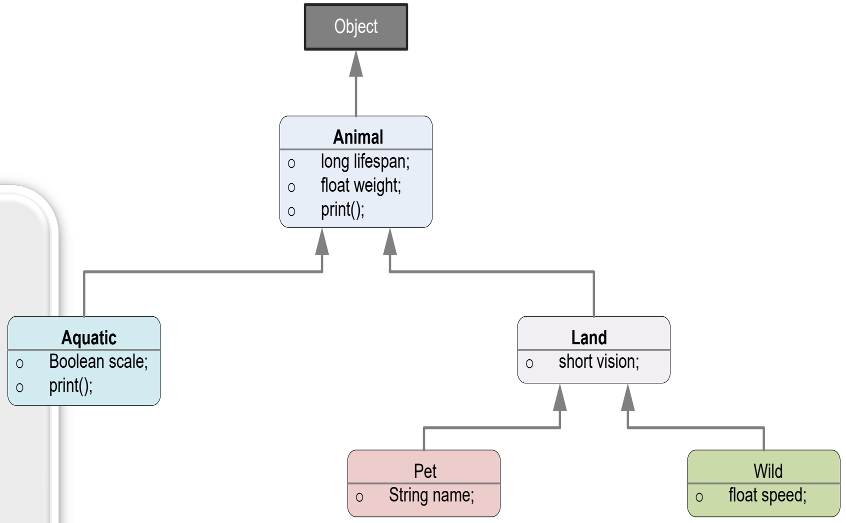
A program is given which consists of the following parts.

1. Definition of all the classes as shown in the figure.
2. Declaration of the generic class, which can be used to store different lists of animals.
3. Definitions of different methods to handle objects of different classes in the class hierarchy.
4. Driver class to manipulate the objects of different types.



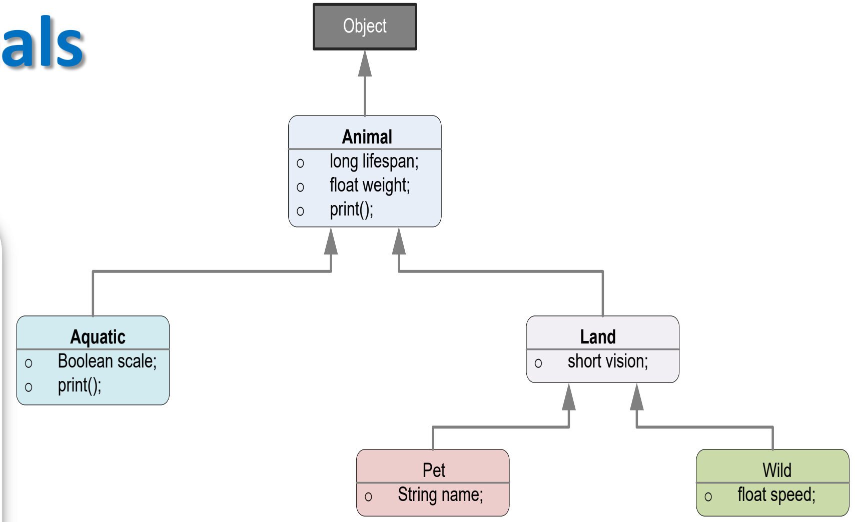
Example 5.5: Definition of all the classes in animals

```
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...
```



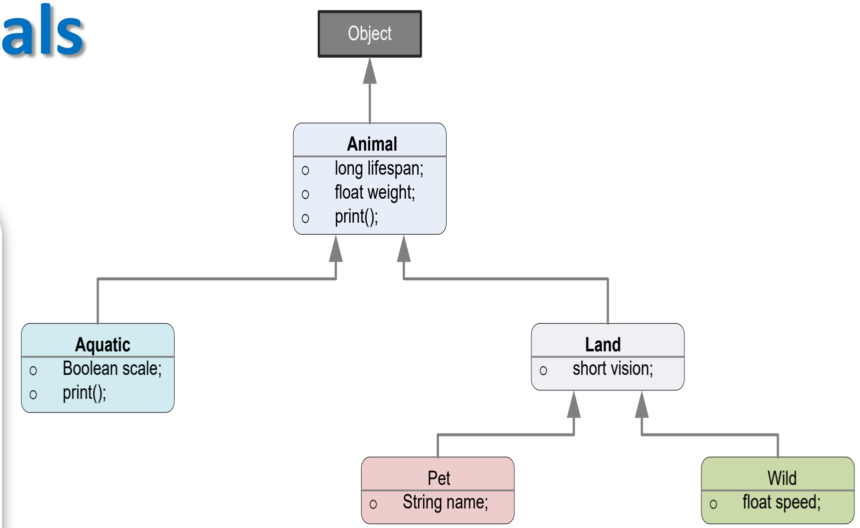
Example 5.5: Definition of all the classes in animals

```
// 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...
```



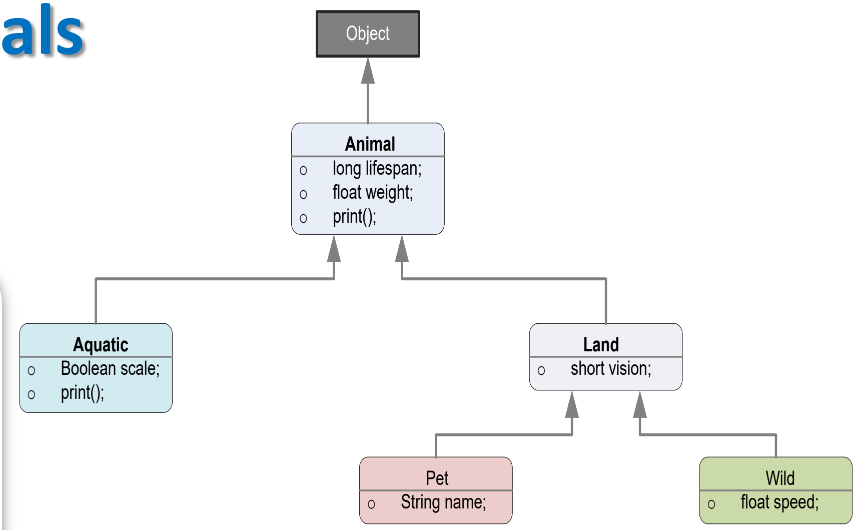
Example 5.5: Definition of all the classes in animals

```java
// 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...
```



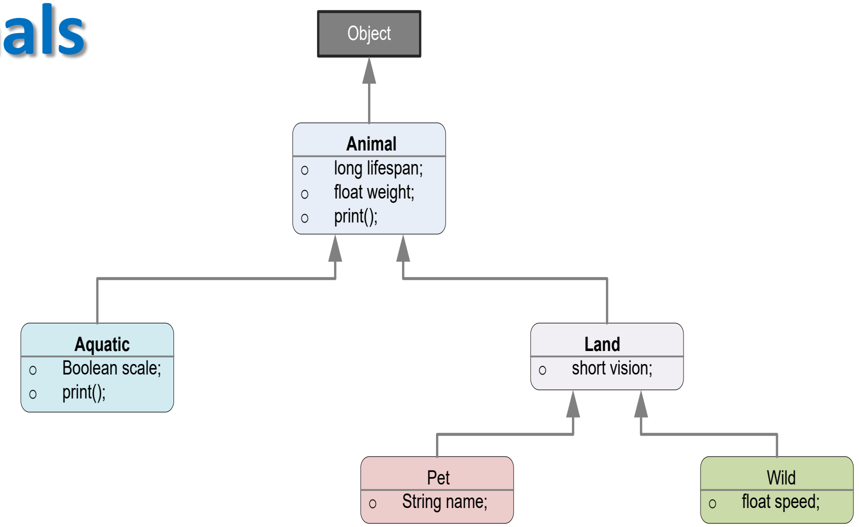
Example 5.5: Definition of all the classes in animals

```
// Continued on...

class Pet extends Land {
      String name;
      Pet(long years, float kg, short vision, String name) {
          super(years, kgs, vision, name);   // Super class constructor
             this.name = name;
      }
}   // End of class Pet




                                                        // Continued to next...
```



Example 5.5: Definition of all the classes in animals

```java
// 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...
```



Object

Animal
- long lifespan;
- float weight;
- print();

Aquatic
- Boolean scale;
- print();

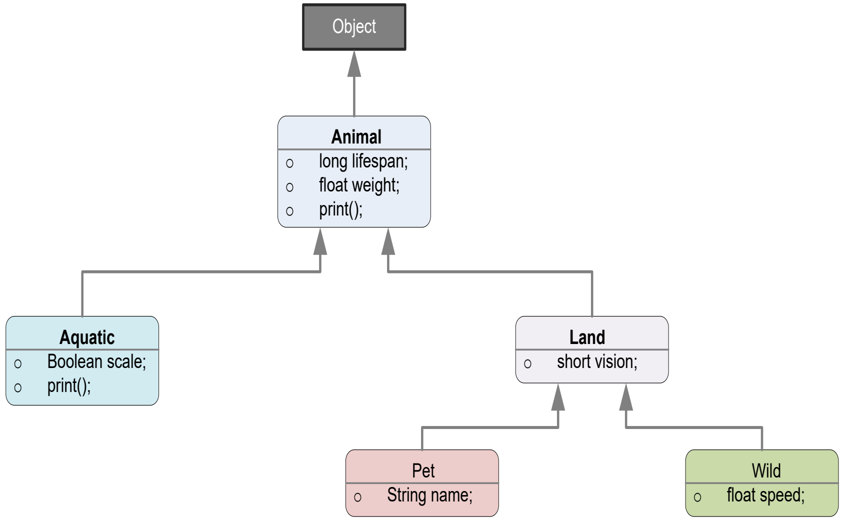
Land
- short vision;

Pet
- String name;

Wild
- float speed;

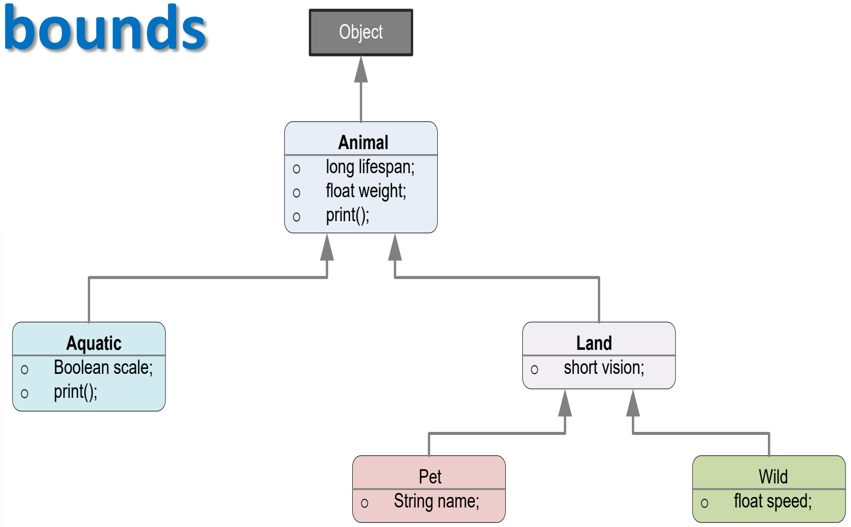
Example 5.5: Defining generic class to maintain lists of animals

```
// 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...
```



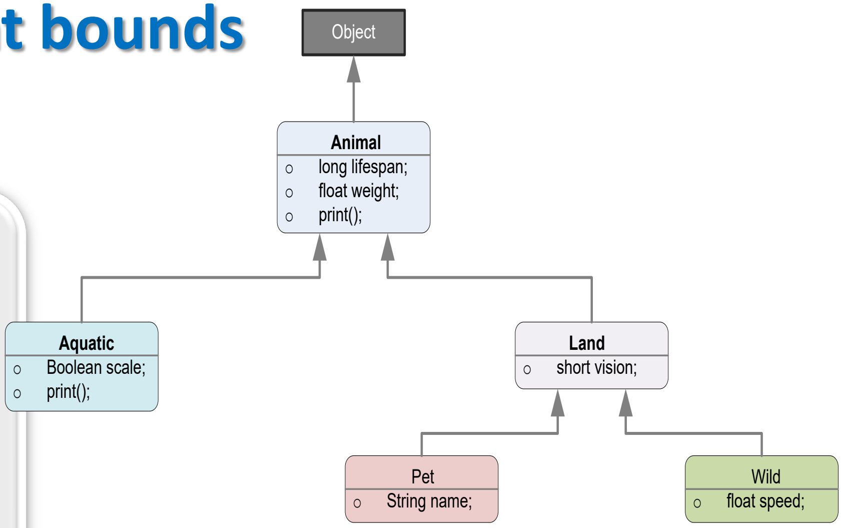
Example 5.5: Defining different methods with different bounds of arguments

```
// 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...
```



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...
```



Object

Animal
○ long lifespan;
○ float weight;
○ print();

Aquatic
○ Boolean scale;
○ print();

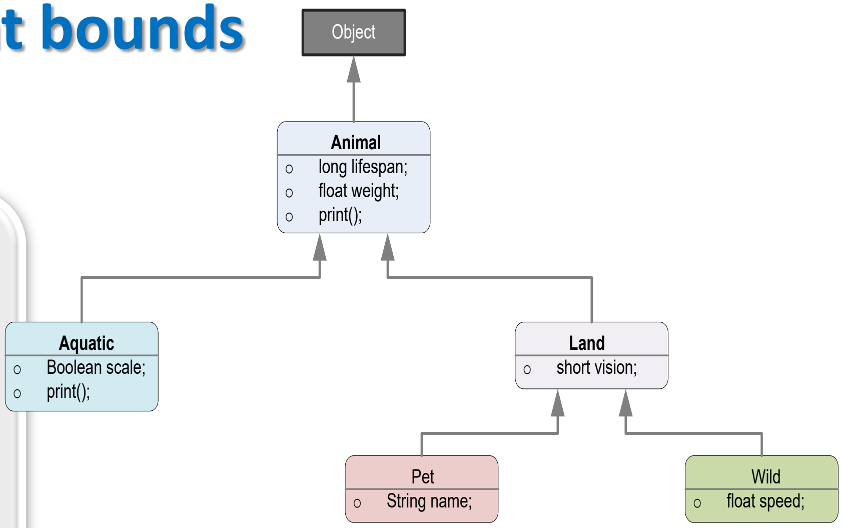
Land
○ short vision;

Pet
○ String name;

Wild
○ float speed;

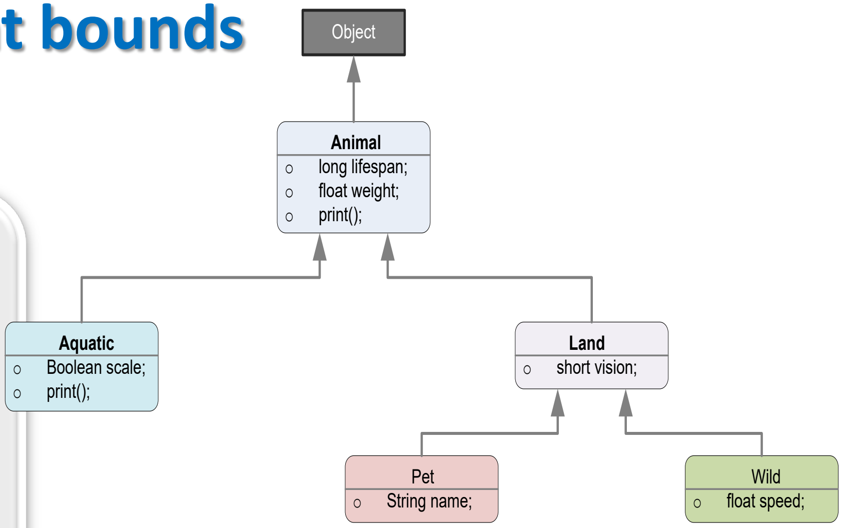
Example 5.5: Defining different methods with different bounds of arguments

```
// 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...
```



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...
```



Object

Animal
- long lifespan;
- float weight;
- print();

Aquatic
- Boolean scale;
- print();

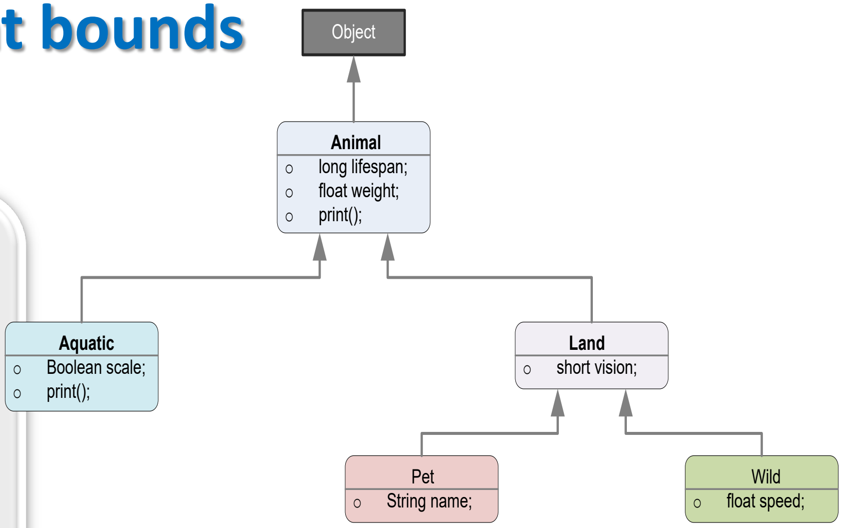
Land
- short vision;

Pet
- String name;

Wild
- float speed;

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...
```



Object

Animal
- ○ long lifespan;
- ○ float weight;
- ○ print();

Aquatic
- ○ Boolean scale;
- ○ print();

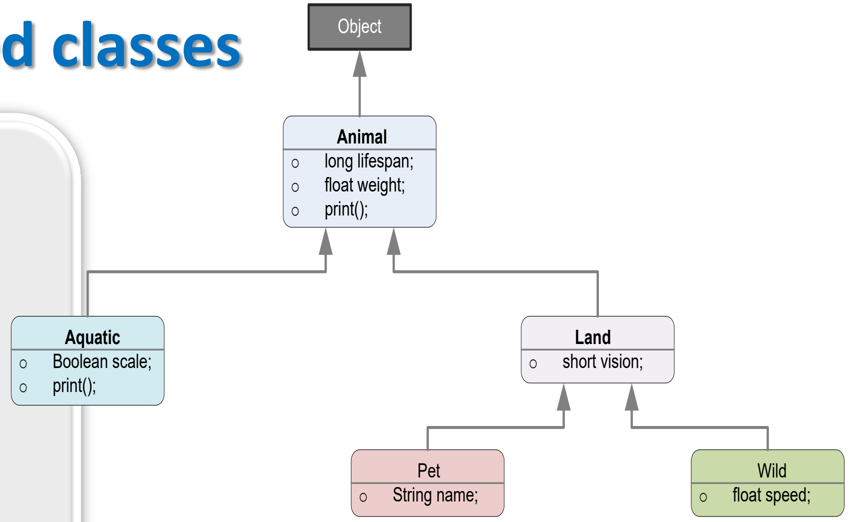
Land
- ○ short vision;

Pet
- ○ String name;

Wild
- ○ float speed;

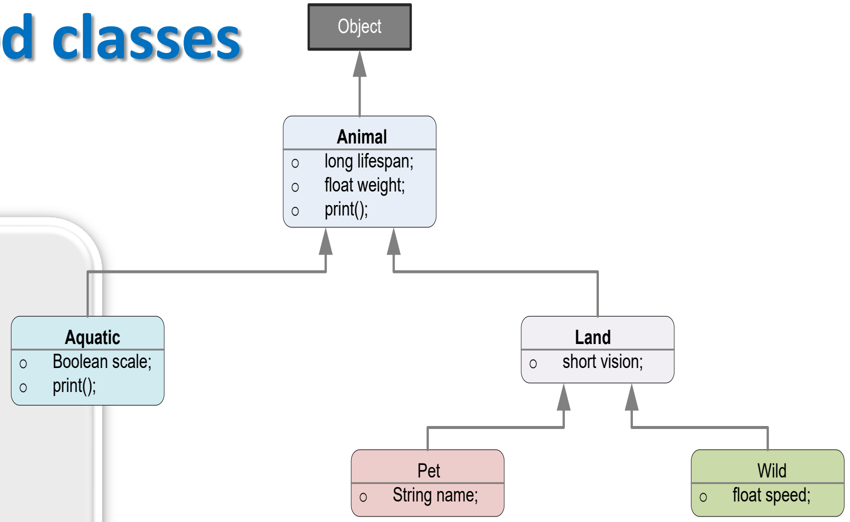
Example 5.5: Main program utilizing the above-defined classes

```
// 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...
```



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...
```



Object

Animal
- long lifespan;
- float weight;
- print();

Aquatic
- Boolean scale;
- print();

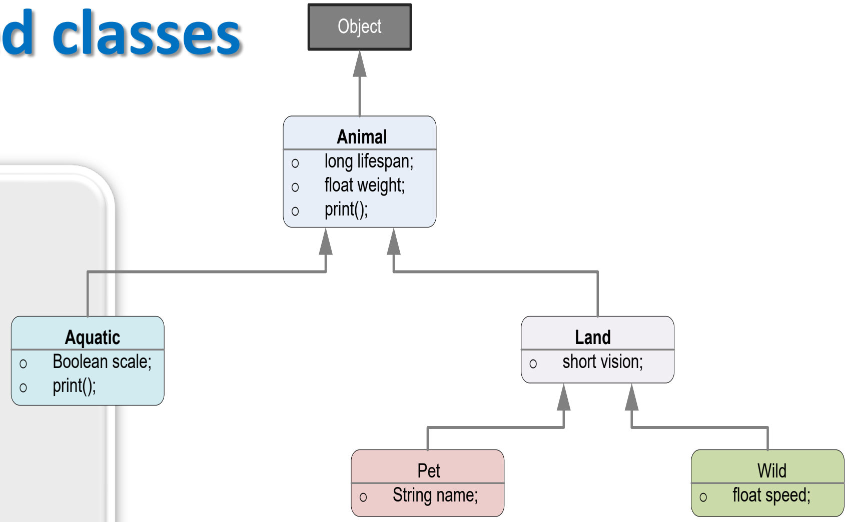
Land
- short vision;

Pet
- String name;

Wild
- float speed;

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...
```



Object

Animal
- long lifespan;
- float weight;
- print();

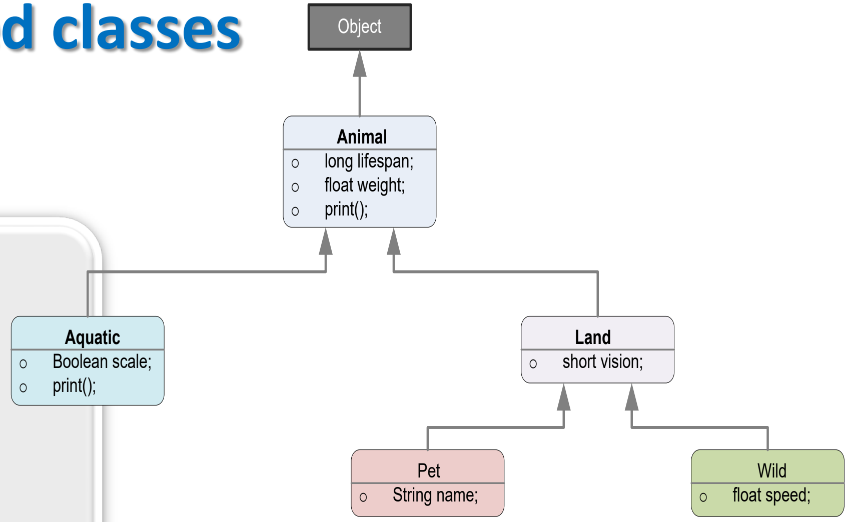
Aquatic
- Boolean scale;
- print();

Land
- short vision;

Pet
- String name;

Wild
- float speed;

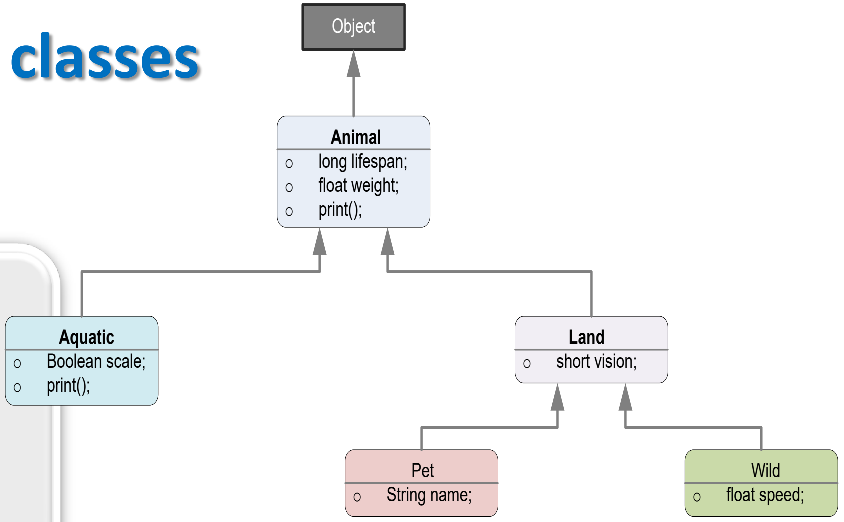
Example 5.5: Main program utilizing the above-defined classes



```
// 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...
```

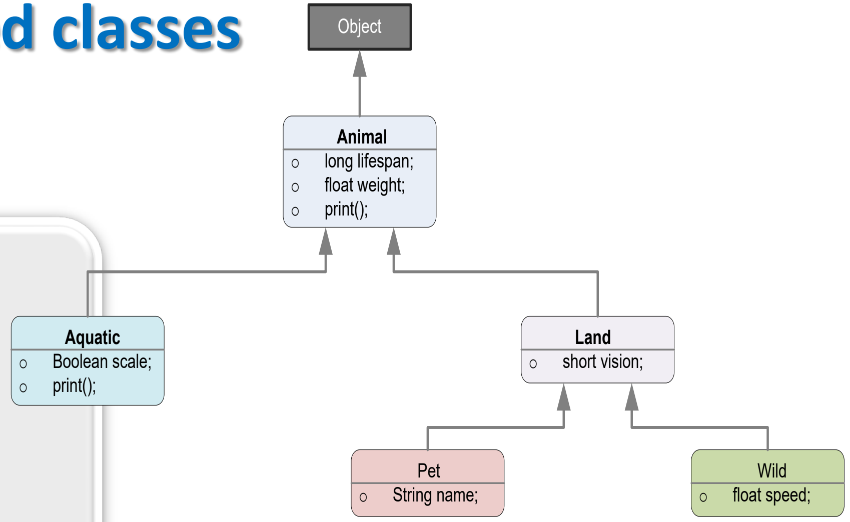
Example 5.5: Main program utilizing the above-defined classes

```
// 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...
```



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...
```


Object

Animal
- long lifespan;
- float weight;
- print();

Aquatic
- Boolean scale;
- print();

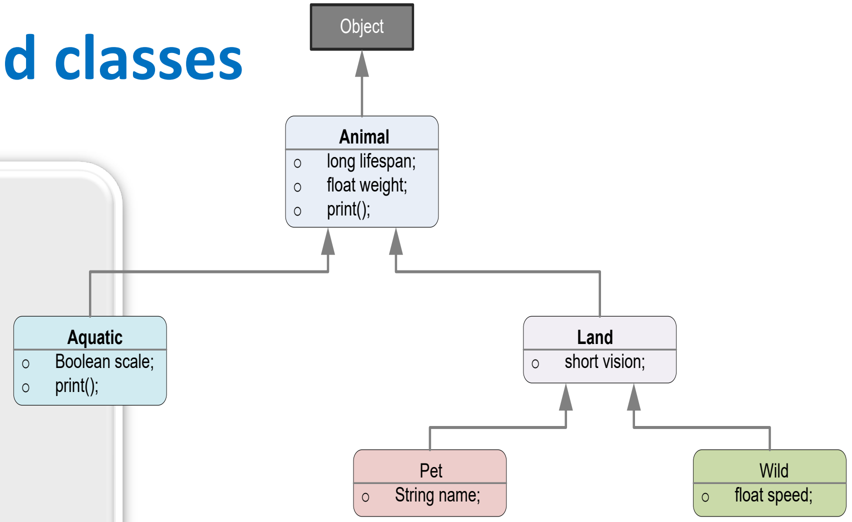
Land
- short vision;

Pet
- String name;

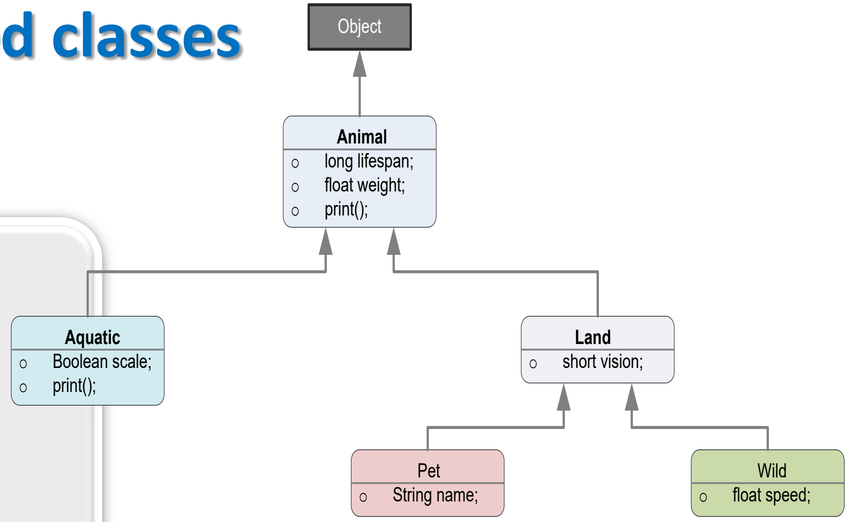
Wild
- float speed;

Example 5.4: Main program utilizing the above-defined classes

```
// Continued on...

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


                                                     // Continued to next...
```



Example 5.5: Main program utilizing the above-defined classes



Object

Animal
○ long lifespan;
○ float weight;
○ print();

Aquatic
○ Boolean scale;
○ print();

Land
○ short vision;

Pet
○ String name;

Wild
○ float speed;

```
// Continued on...

       // showWild(…) is with upper bound wildcard argument with
           // class Wild and its sub classes
            showWild (uList);      // Compile-time error
            showWild (qList);     // Compile-time error
            showWild (lList);    // Compile-time error
            showWild (pList);   // Compile-time error
            showWild (wList);  // OK
      }
}
```

Guideline for Wildcards

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.

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.

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