COM2104: Advanced Programming

LECTURE 9: GENERICS & RECURSION

Objectives

- Understand what is generic type and how to use it for classes and methods.
- Understand what are wildcard in java generics and how to apply it.
- Understand what is recursion and how to use it.

Generics

Generics in Java

- Generics means parameterized types.
- The idea is to allow a type (like Integer, String, etc., or user-defined types) to be a parameter to methods, classes, and interfaces.
- Using Generics, it is possible to create classes that work with different data types.
- An entity such as a class, interface, or method that operates on a parameterized type is a generic entity.

Why Generics?

- The **Object** is the superclass of all other classes, and Object reference can refer to any object.
- These features lack type safety. Generics add that type of safety feature.

Types of Java Generics

- ❖ Generic Classes
- ❖ Generic Functions

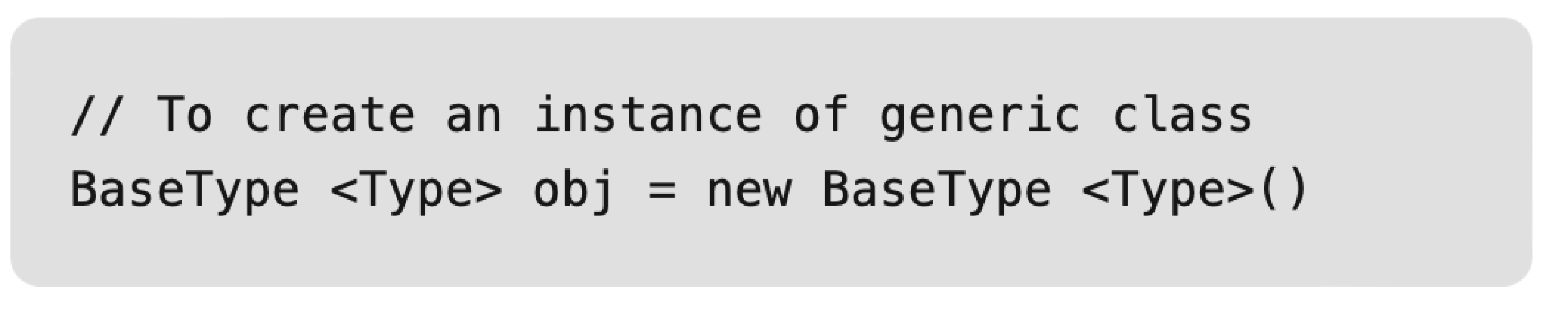
Generic class

- A generic class is implemented exactly like a non-generic class.
- The only difference is that it contains a type parameter section- <T>.
- There can be more than one type of parameter, separated by a comma (e.g., <T, U>).

Generic Class

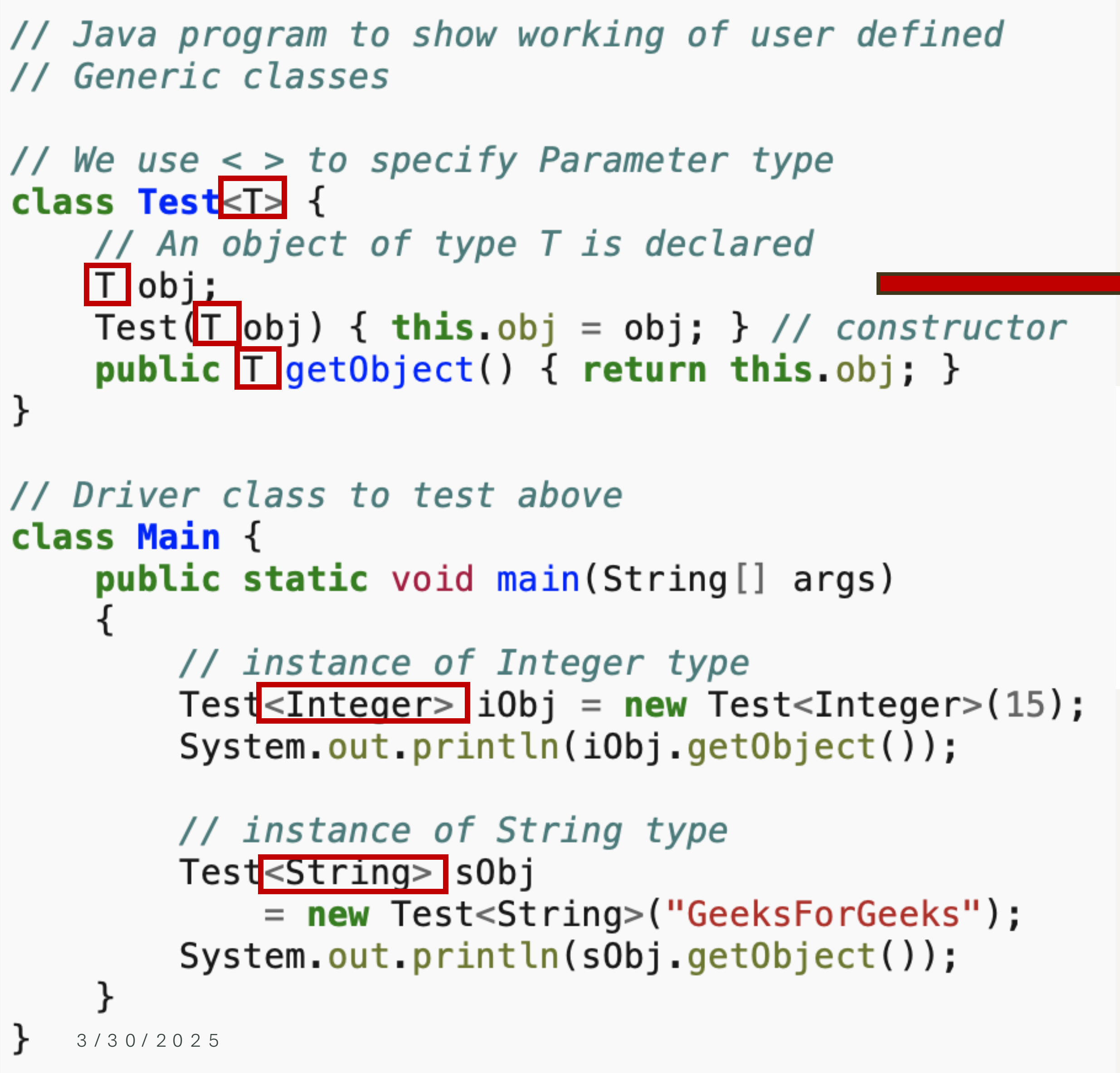
T is used to represent any data type, like String, Integer, Double, and so on.

- we use <> to specify parameter types in generic class creation, like <T>. To create objects of a generic class, we use the following syntax.



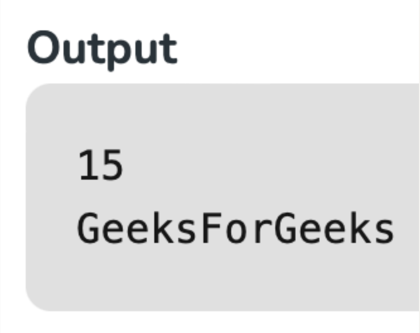
```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()
```

One type parameter in generic class



```java
// Java program to show working of user defined
// Generic classes

// We use < > to specify Parameter type
class Test<T> {
    // An object of type T is declared
    T obj;
    Test(T obj) { this.obj = obj; } // constructor
    public T getObject() { return this.obj; }
}

// Driver class to test above
class Main {
    public static void main(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj
            = new Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
    }
}
```

T in here represent any data type.

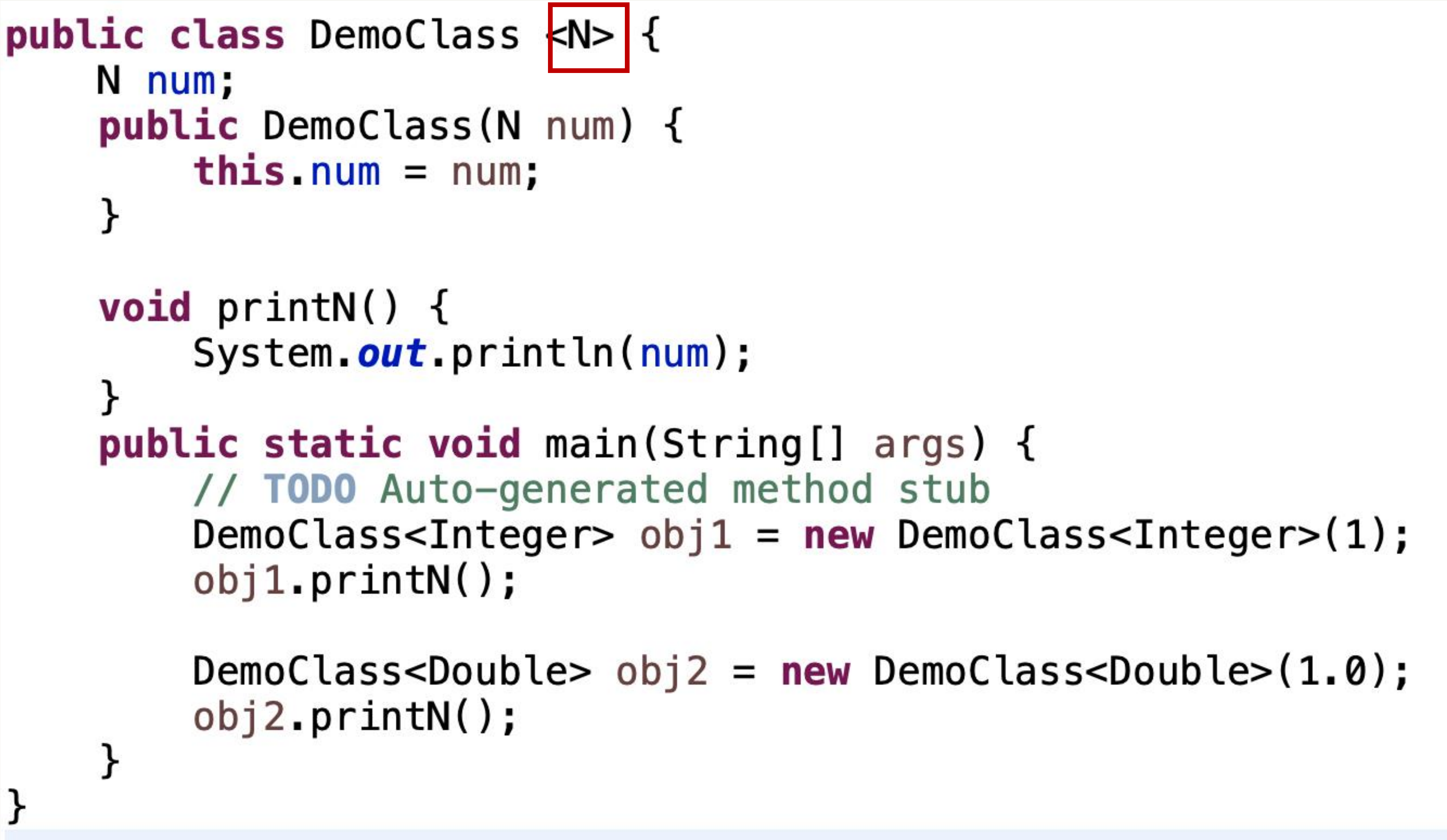


Output

```
15
GeeksForGeeks
```

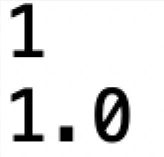
Just like ArrayList, we could provide different data types to instantiate different instances of class Test.

One type parameter in generic class



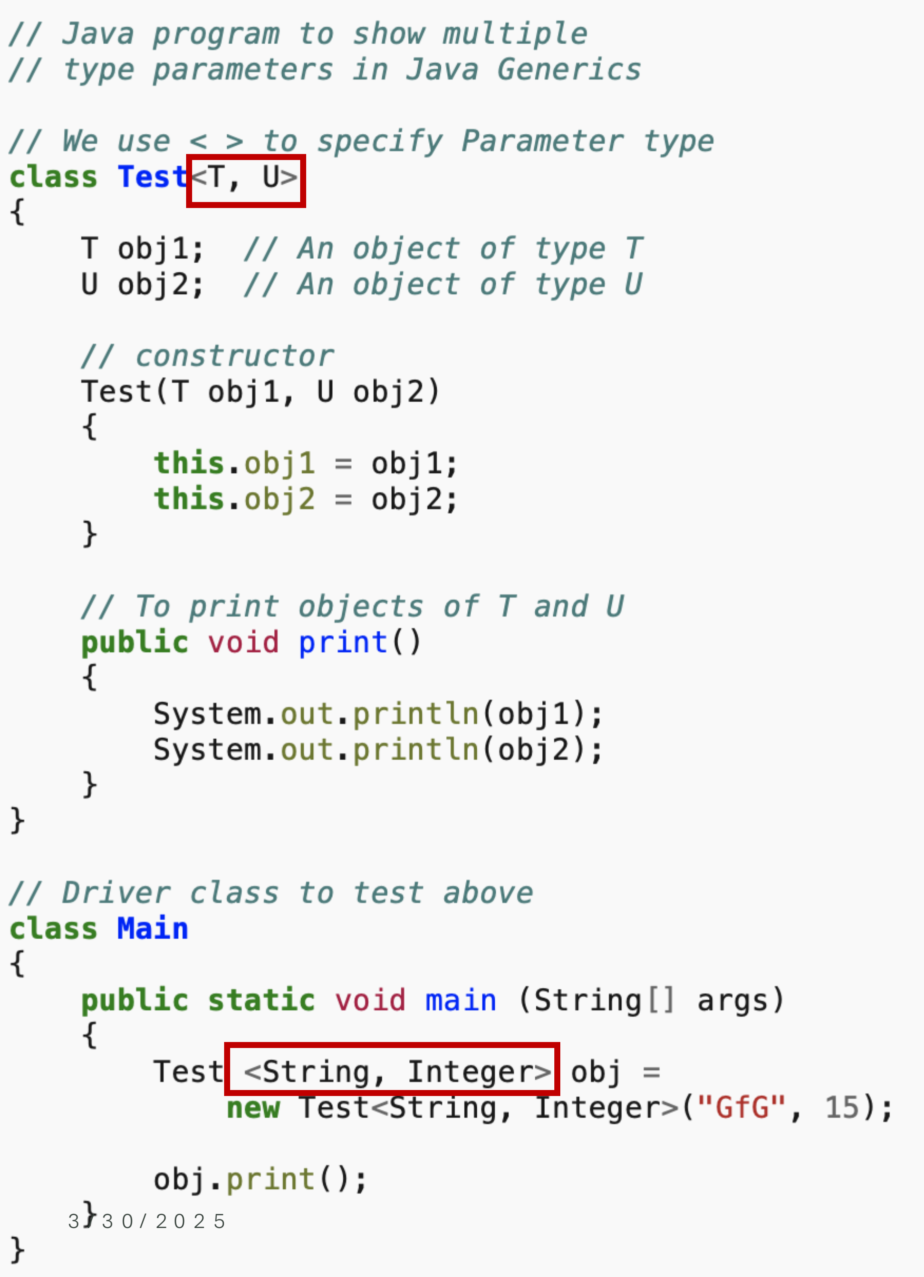
```java
public class DemoClass <N> {
    N num;
    public DemoClass(N num) {
        this.num = num;
    }

    void printN() {
        System.out.println(num);
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        DemoClass<Integer> obj1 = new DemoClass<Integer>(1);
        obj1.printN();

        DemoClass<Double> obj2 = new DemoClass<Double>(1.0);
        obj2.printN();
    }
}
```

Output

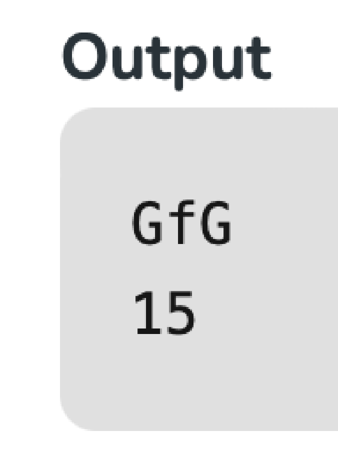


```
1
1.0
```

Multiple type parameters in generic class



```java
// Java program to show multiple
// type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1;  // An object of type T
    U obj2;  // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =
            new Test<String, Integer>("GfG", 15);

        obj.print();
    }
}
```



Output

```
GfG
15
```

What's the meaning of U in here?

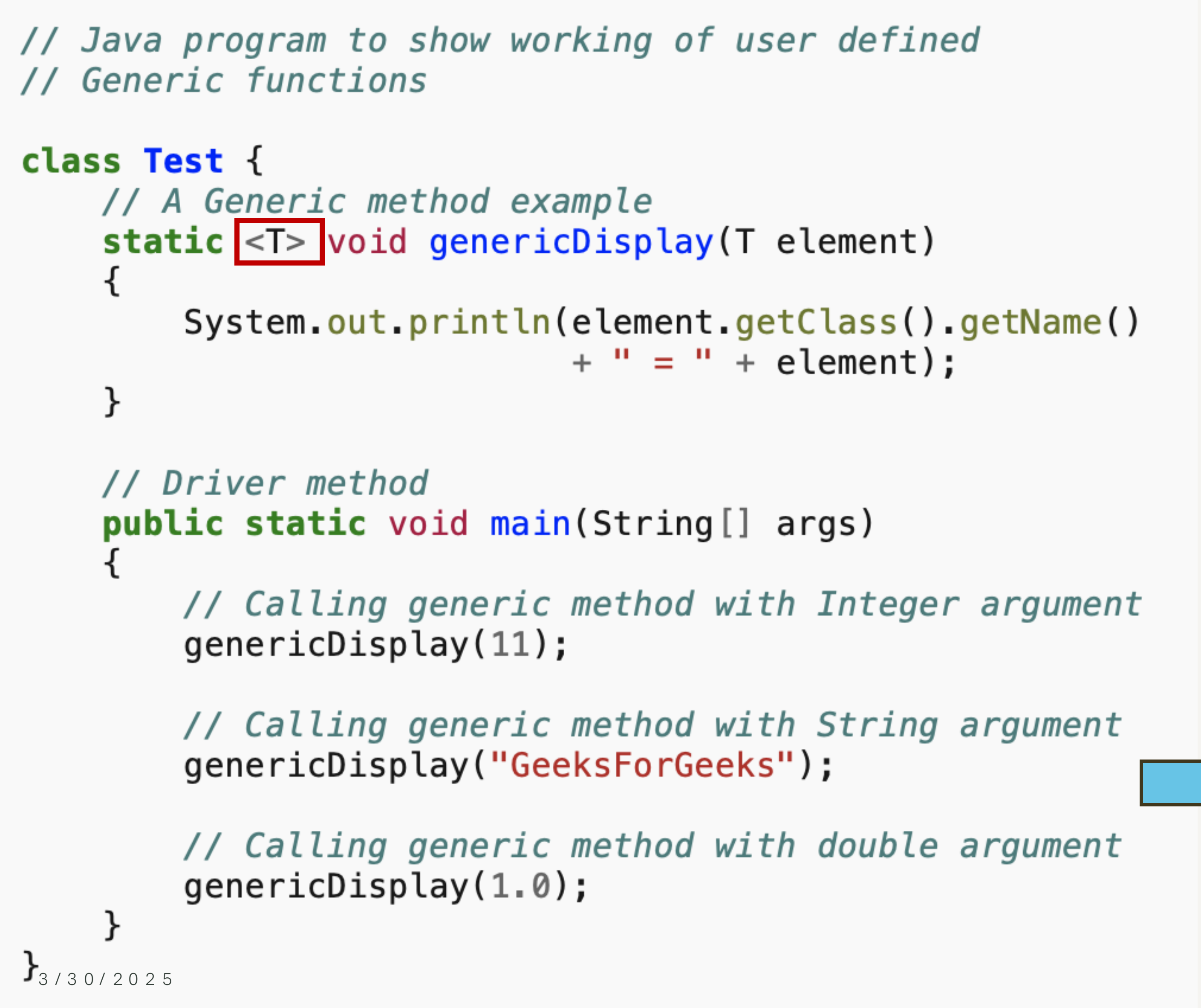
Type Parameters in Java Generics

Type Parameter	Description
T	Type. Referring to any data types.
E	Element. Referring to elements with different data types in a list.
N	Number. Referring to numbers.
K	Key (Used in Map).
V	Value (Used in Map).
S, U, V	2nd, 3rd, 4th types. Also referring to any data types. But used after we have used T.

Comments on Generic Type Usage

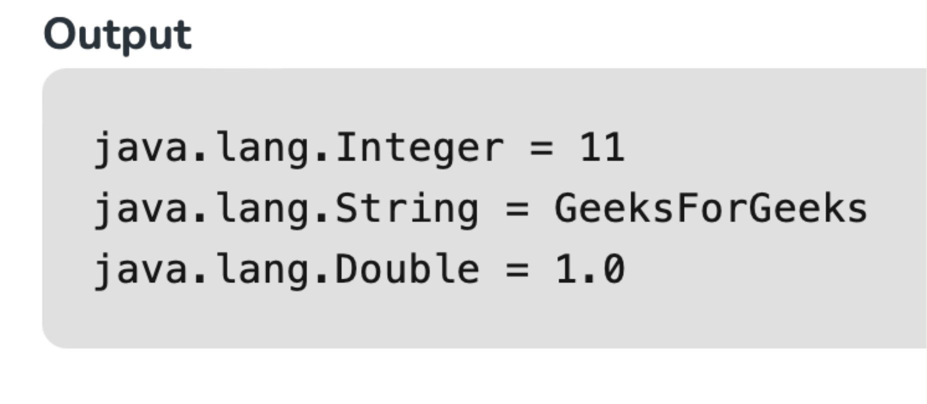
- T for General Purpose: Use T when the type can be any object and the method or class operates generically on this type.
- E for Collections: Use E for collection elements to indicate that the type is used for items in collections like List, Set, or Queue.
- K and V for Maps: Use K and V for keys and values in maps, making it clear what types are expected for map operations.
- S, U, V for Relationships: Use these when there is a need to represent multiple related types within the same method or class, especially when there are interactions between these types.

Generics Functions



```
// Java program to show working of user defined
// Generic functions

class Test {
    // A Generic method example
    static <T> void genericDisplay(T element)
    {
        System.out.println(element.getClass().getName()
                           + " = " + element);
    }

    // Driver method
    public static void main(String[] args)
    {
        // Calling generic method with Integer argument
        genericDisplay(11);

        // Calling generic method with String argument
        genericDisplay("GeeksForGeeks");

        // Calling generic method with double argument
        genericDisplay(1.0);
    }
}
```

We have to add <T> for the header of one method.

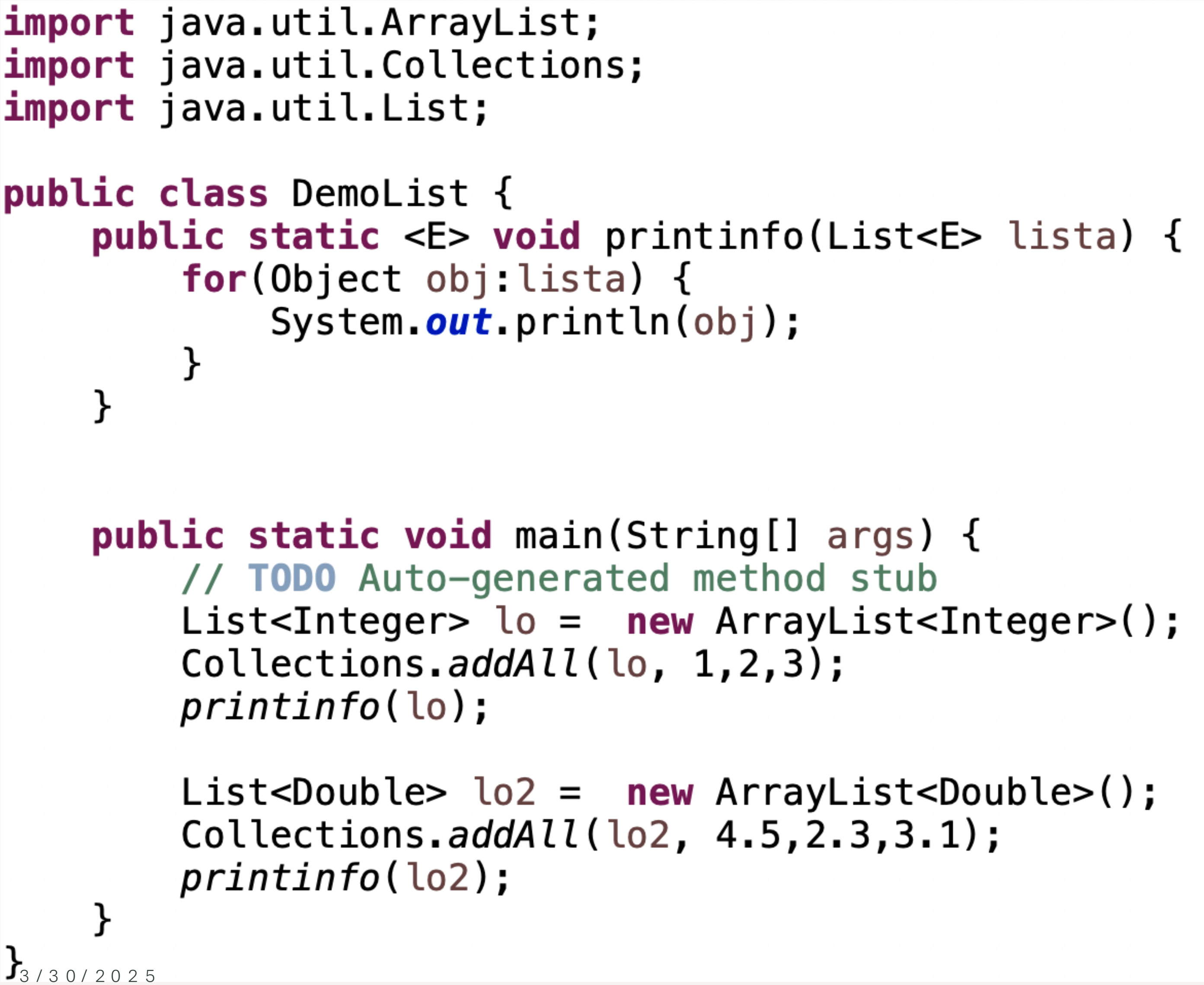


Output

```
java.lang.Integer = 11
java.lang.String = GeeksForGeeks
java.lang.Double = 1.0
```

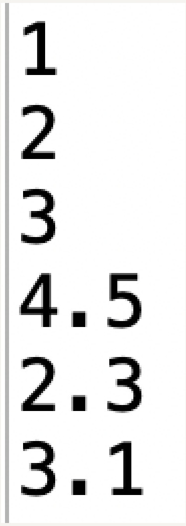
When implementation, we could pass data with different types to the function.

Generics Functions



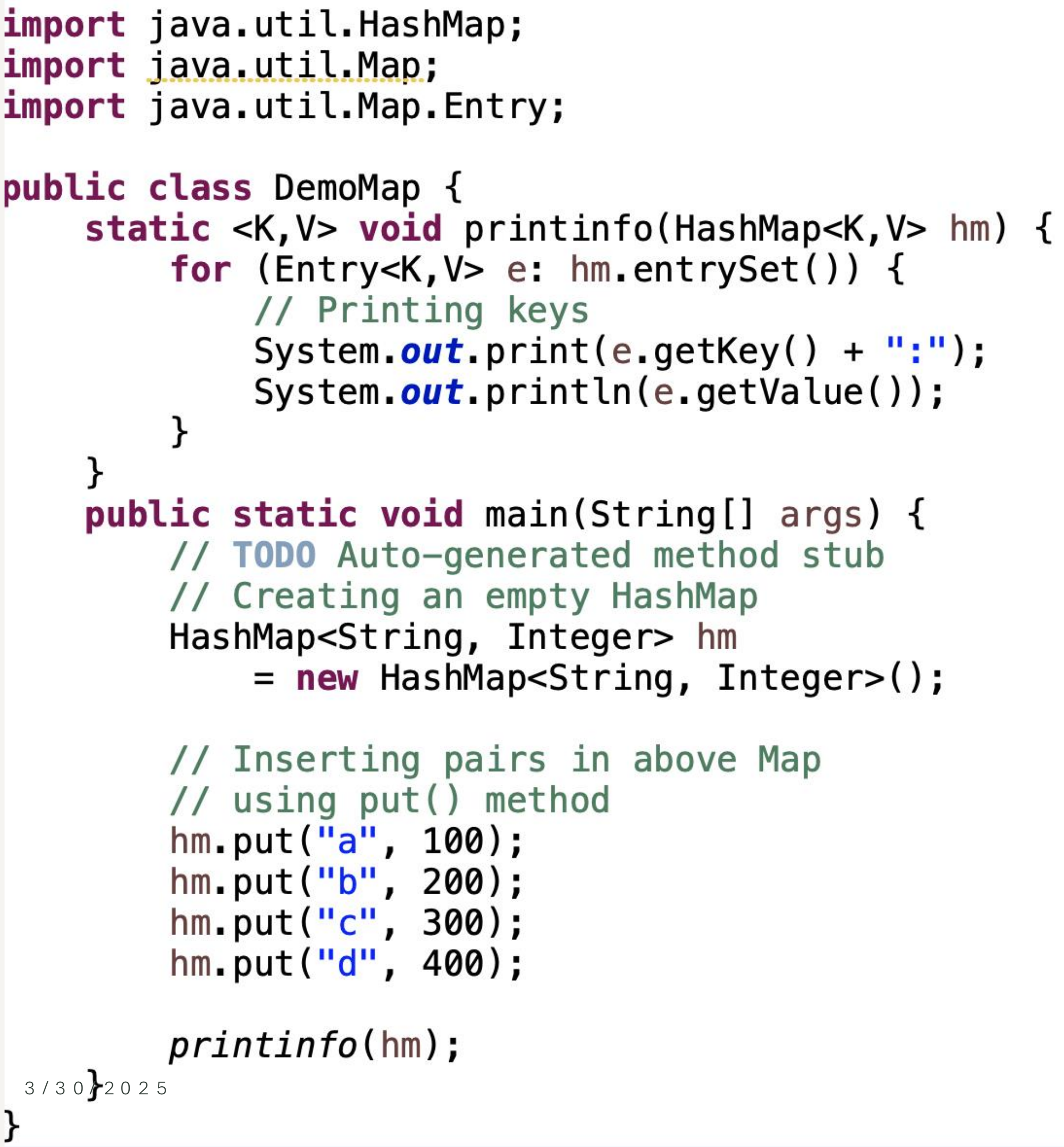
```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class DemoList {
    public static <E> void printinfo(List<E> lista) {
        for(Object obj:lista) {
            System.out.println(obj);
        }
    }


    public static void main(String[] args) {
        // TODO Auto-generated method stub
        List<Integer> lo =  new ArrayList<Integer>();
        Collections.addAll(lo, 1,2,3);
        printinfo(lo);

        List<Double> lo2 =  new ArrayList<Double>();
        Collections.addAll(lo2, 4.5,2.3,3.1);
        printinfo(lo2);
    }
}
```

output



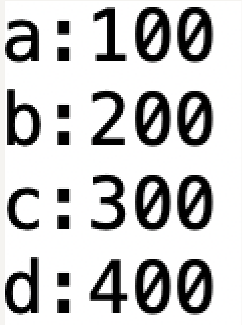
```
1
2
3
4.5
2.3
3.1
```

Generics Functions



```java
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

public class DemoMap {
    static <K,V> void printinfo(HashMap<K,V> hm) {
        for (Entry<K,V> e: hm.entrySet()) {
            // Printing keys
            System.out.print(e.getKey() + ":");
            System.out.println(e.getValue());
        }
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // Creating an empty HashMap
        HashMap<String, Integer> hm
            = new HashMap<String, Integer>();

        // Inserting pairs in above Map
        // using put() method
        hm.put("a", 100);
        hm.put("b", 200);
        hm.put("c", 300);
        hm.put("d", 400);

        printinfo(hm);
    }
}
```

output



```
a:100
b:200
c:300
d:400
```

Remarks about generic functions

- If the method has generic type parameters, we should name the function with the same generic type in the header.

```
public <T> void printinfo(T obj) {...}
```

```
public <T, U> void printinfo(T obj, U obj2) {...}
```

- If the method has no parameters of a generic type, the generic type can be removed when naming the method.

```
public void printinfo() {...}
```

Generics Wildcards

Wildcards in Java

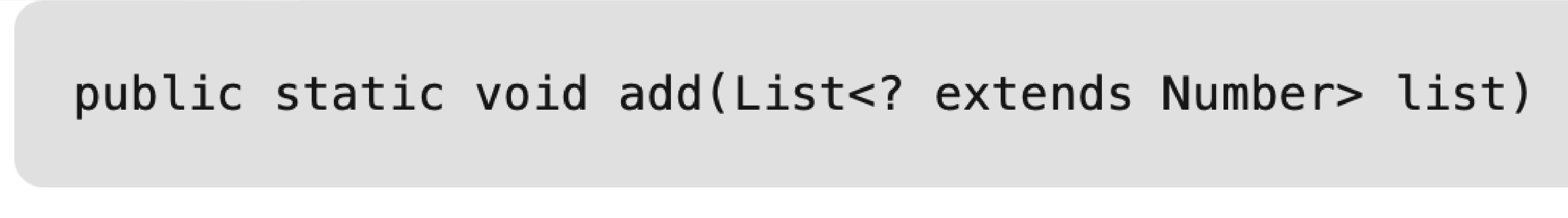
- The question mark (?) is known as the wildcard in generic programming. It represents an unknown type.
- The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type.
- Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly. This incompatibility may be softened by the wildcard if ? is used as an actual type parameter.

Types of wildcards in Java

- Upper Bounded Wildcards
- Lower Bounded Wildcards
- Unbounded Wildcard

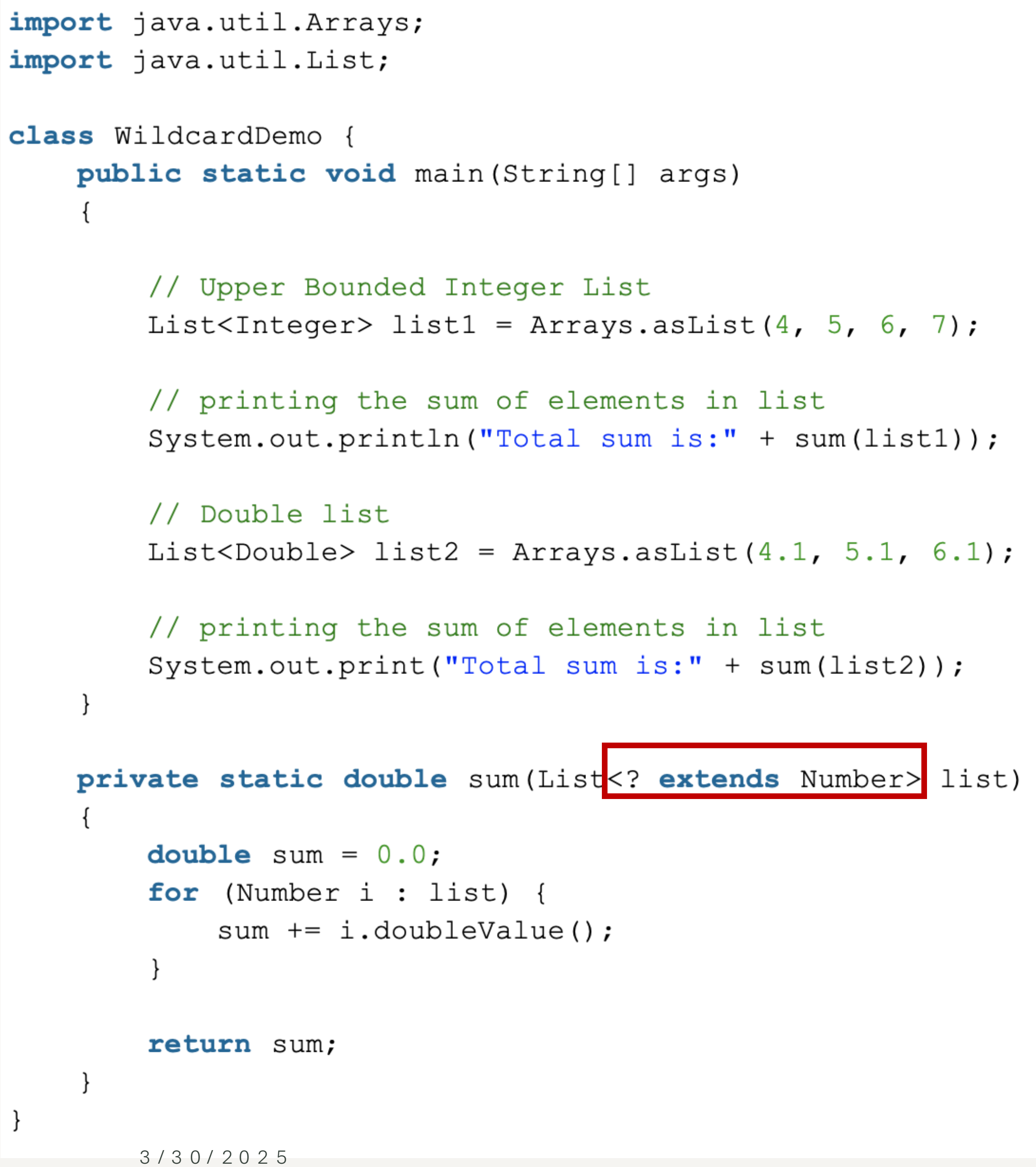
Upper Bounded Wildcards

- These wildcards can be used when you want to relax the restrictions on a variable.
  - For example, say you want to write a method that works on List < Integer >, List < Double >, and List < Number >, you can do this using an upper bounded wildcard.
- To declare an upper-bounded wildcard, use the wildcard character (‘?’), followed by the **extends** keyword, followed by its upper bound.



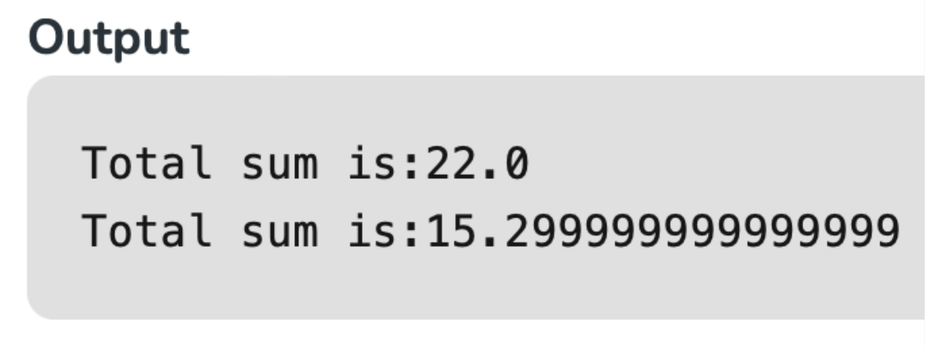
```
public static void add(List<? extends Number> list)
```

One example



```java
import java.util.Arrays;
import java.util.List;

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

        // Upper Bounded Integer List
        List<Integer> list1 = Arrays.asList(4, 5, 6, 7);

        // printing the sum of elements in list
        System.out.println("Total sum is:" + sum(list1));

        // Double list
        List<Double> list2 = Arrays.asList(4.1, 5.1, 6.1);

        // printing the sum of elements in list
        System.out.print("Total sum is:" + sum(list2));
    }

    private static double sum(List<? extends Number> list)
    {
        double sum = 0.0;
        for (Number i : list) {
            sum += i.doubleValue();
        }

        return sum;
    }
}
```

Number class is the super class of Integer, Double, Long, Float and etc.

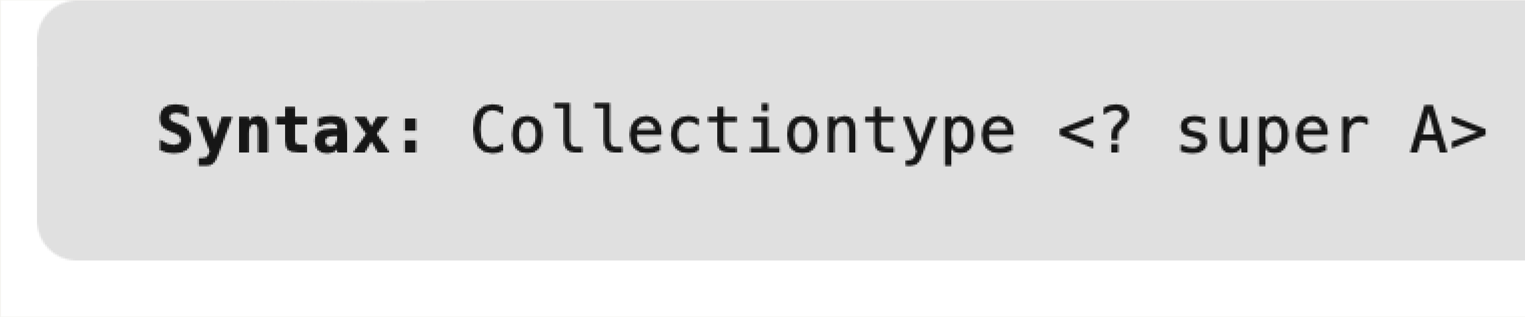


Output

```
Total sum is:22.0
Total sum is:15.299999999999999
```

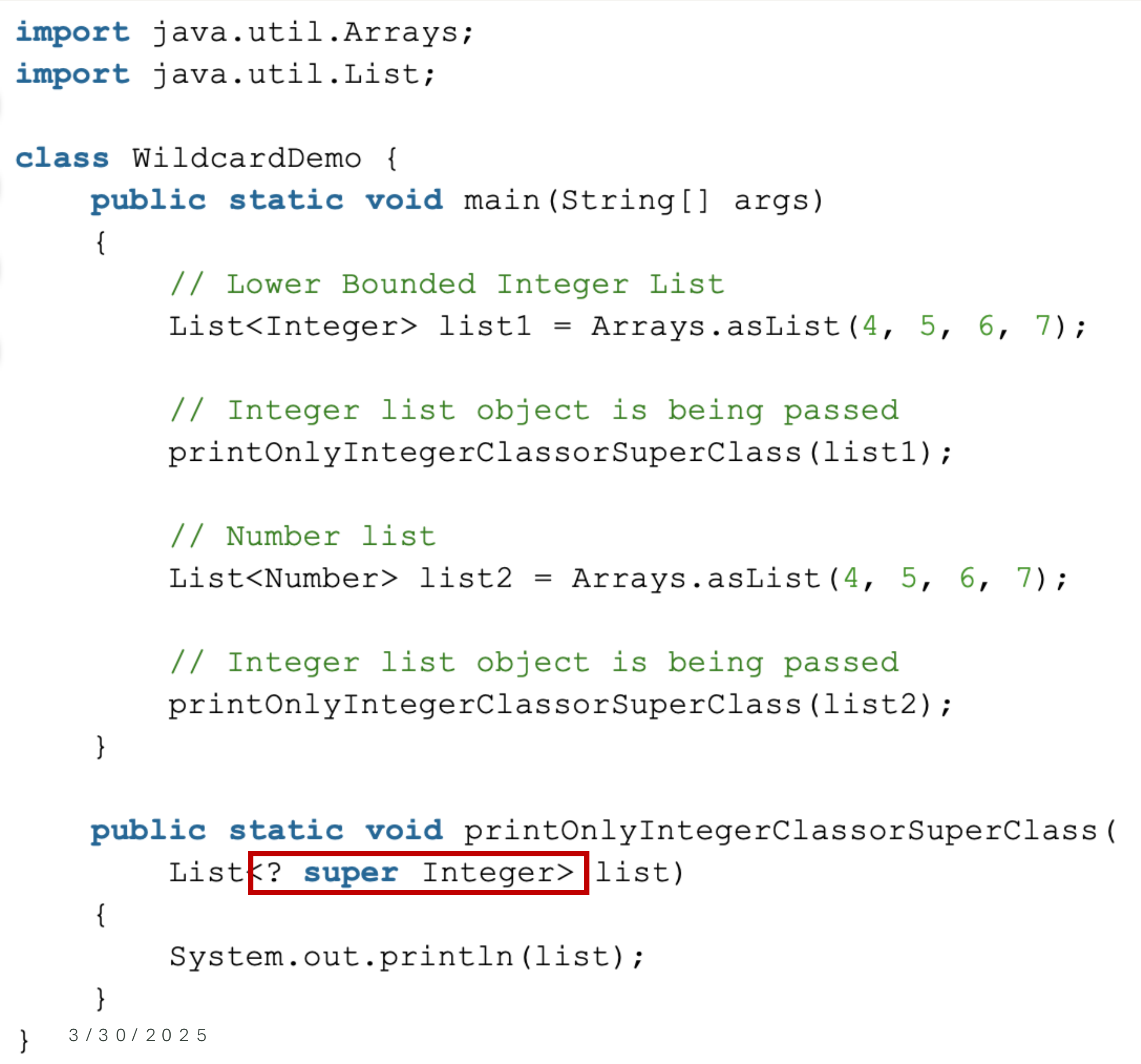
Lower Bounded Wildcards

- It is expressed using the wildcard character (‘?’), followed by the super keyword, followed by its lower bound: <? super A>.



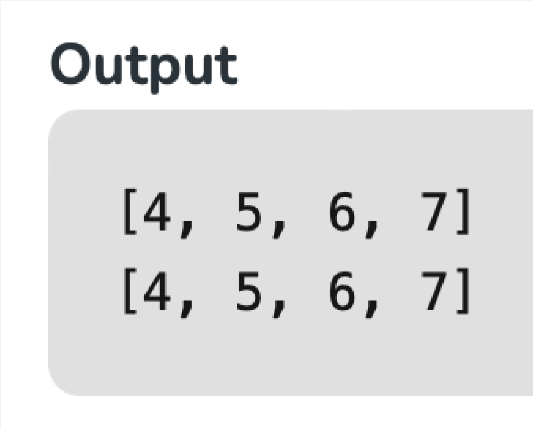
```
Syntax: Collectiontype <? super A>
```

One example



```java
import java.util.Arrays;
import java.util.List;

class WildcardDemo {
    public static void main(String[] args)
    {
        // Lower Bounded Integer List
        List<Integer> list1 = Arrays.asList(4, 5, 6, 7);

        // Integer list object is being passed
        printOnlyIntegerClassorSuperClass(list1);

        // Number list
        List<Number> list2 = Arrays.asList(4, 5, 6, 7);

        // Integer list object is being passed
        printOnlyIntegerClassorSuperClass(list2);
    }

    public static void printOnlyIntegerClassorSuperClass(
        List<? super Integer> list)
    {
        System.out.println(list);
    }
}
```

- Here arguments can be Integer or superclass of Integer(which is Number).
- The method printOnlyIntegerClassorSuperClass will **only take Integer or its superclass objects**.
- However, if we pass a list of types **Double** then we will get a **compilation error**. It is because only the Integer field or its superclass can be passed.
- Double is not the superclass of Integer.



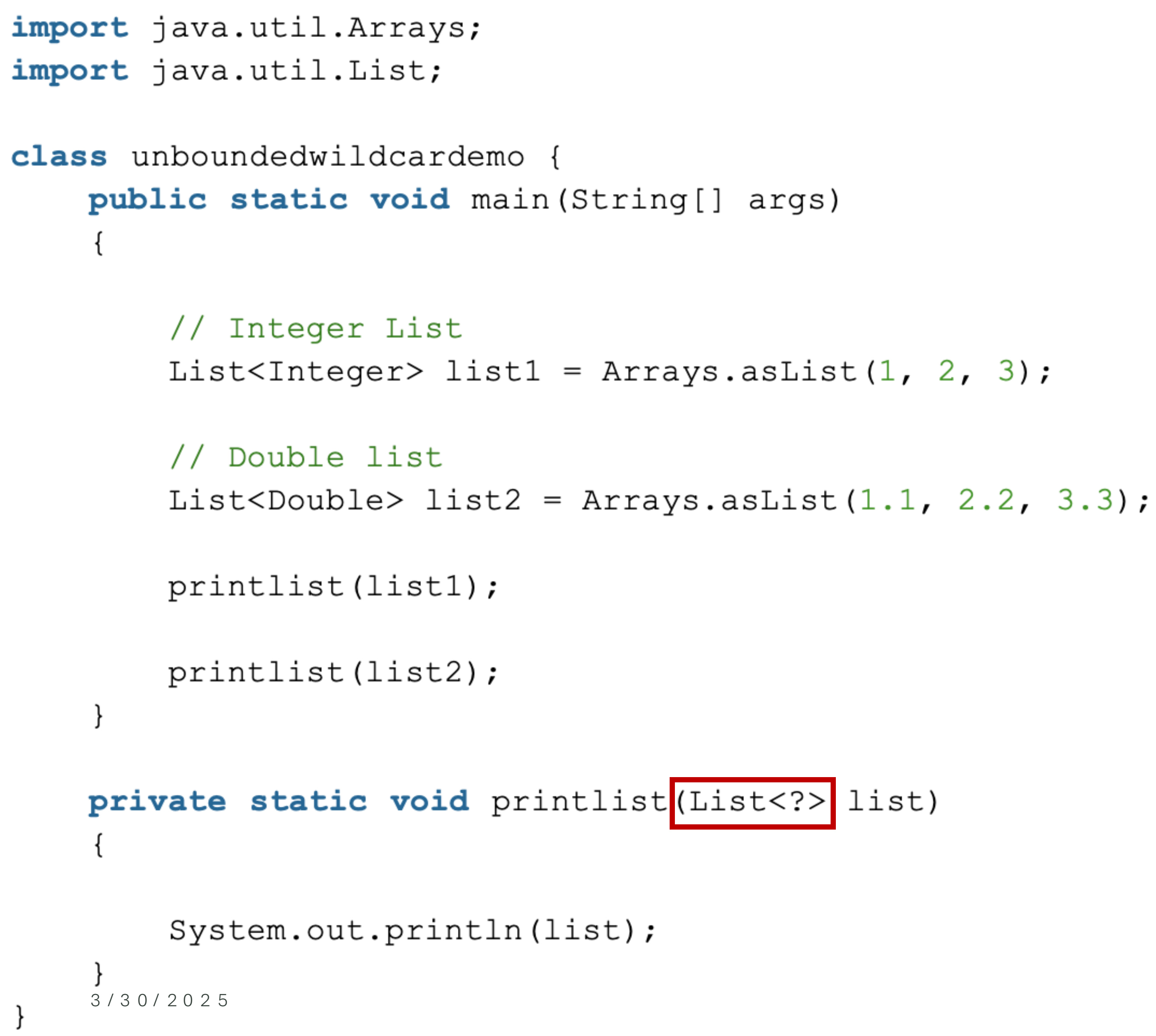
Output

```
[4, 5, 6, 7]
[4, 5, 6, 7]
```

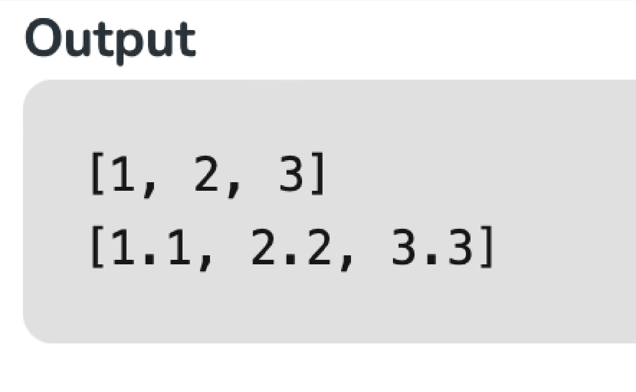
Unbounded Wildcard

- This wildcard type is specified using the wildcard character (?), for example, List. This is called a list of unknown types.

One example



```
import java.util.Arrays;
import java.util.List;

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

        // Integer List
        List<Integer> list1 = Arrays.asList(1, 2, 3);

        // Double list
        List<Double> list2 = Arrays.asList(1.1, 2.2, 3.3);

        printlist(list1);

        printlist(list2);
    }

    private static void printlist(List<?> list)
    {

        System.out.println(list);
    }
}
```

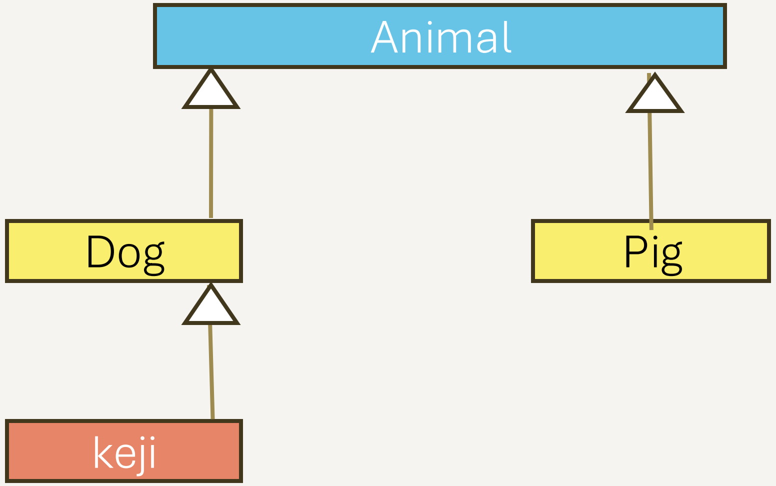


Output

```
[1, 2, 3]
[1.1, 2.2, 3.3]
```

Apply wildcards to our own classes with inherited relationship

- Suppose we have four classes with the following relationship. Dog and Pig inherit Animal. keji inherits Dog.



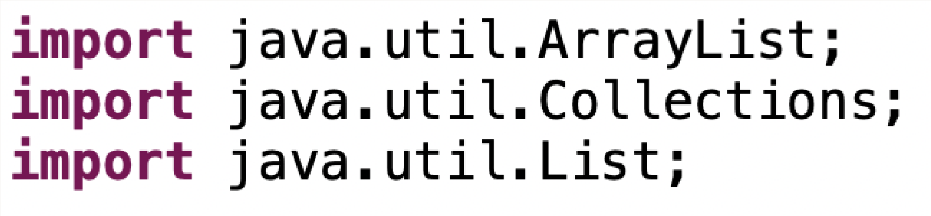
Animal

Dog

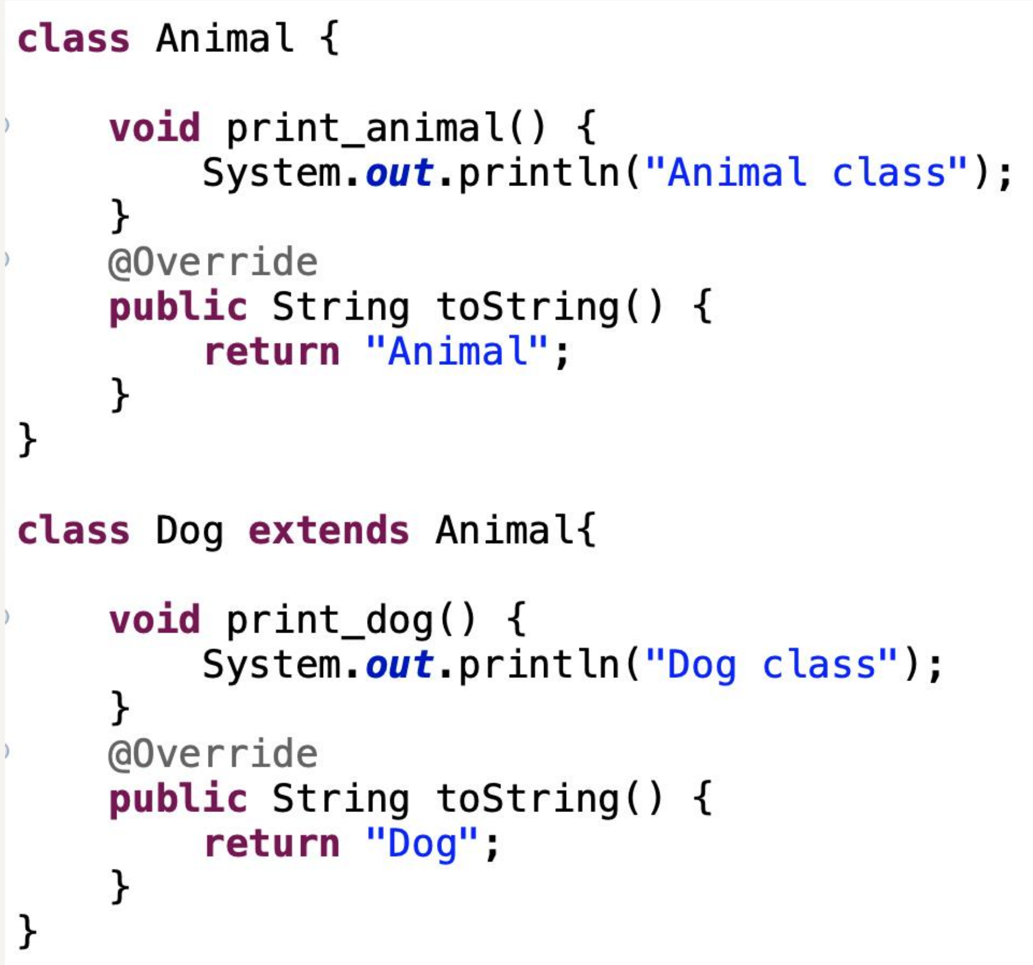
Pig

keji

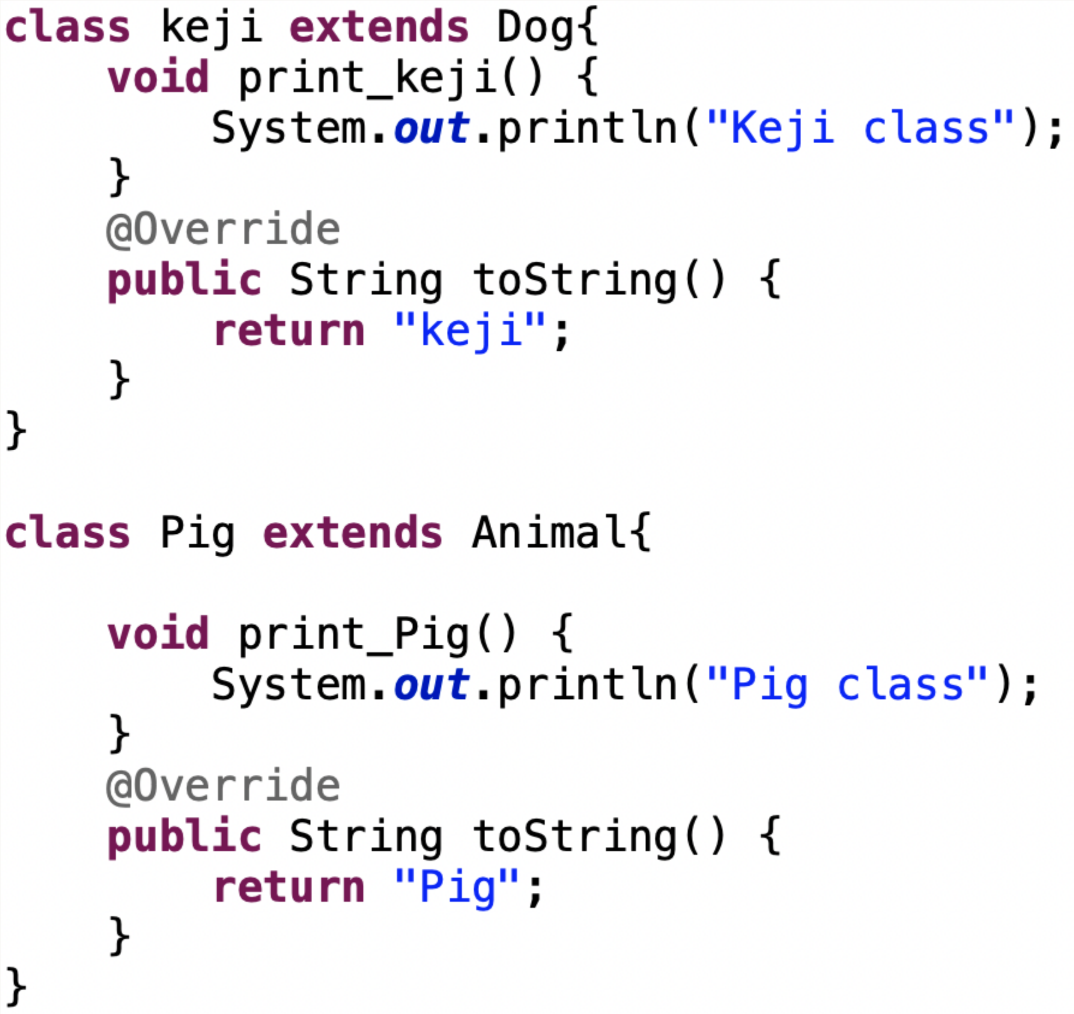
The above example in code



```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
```

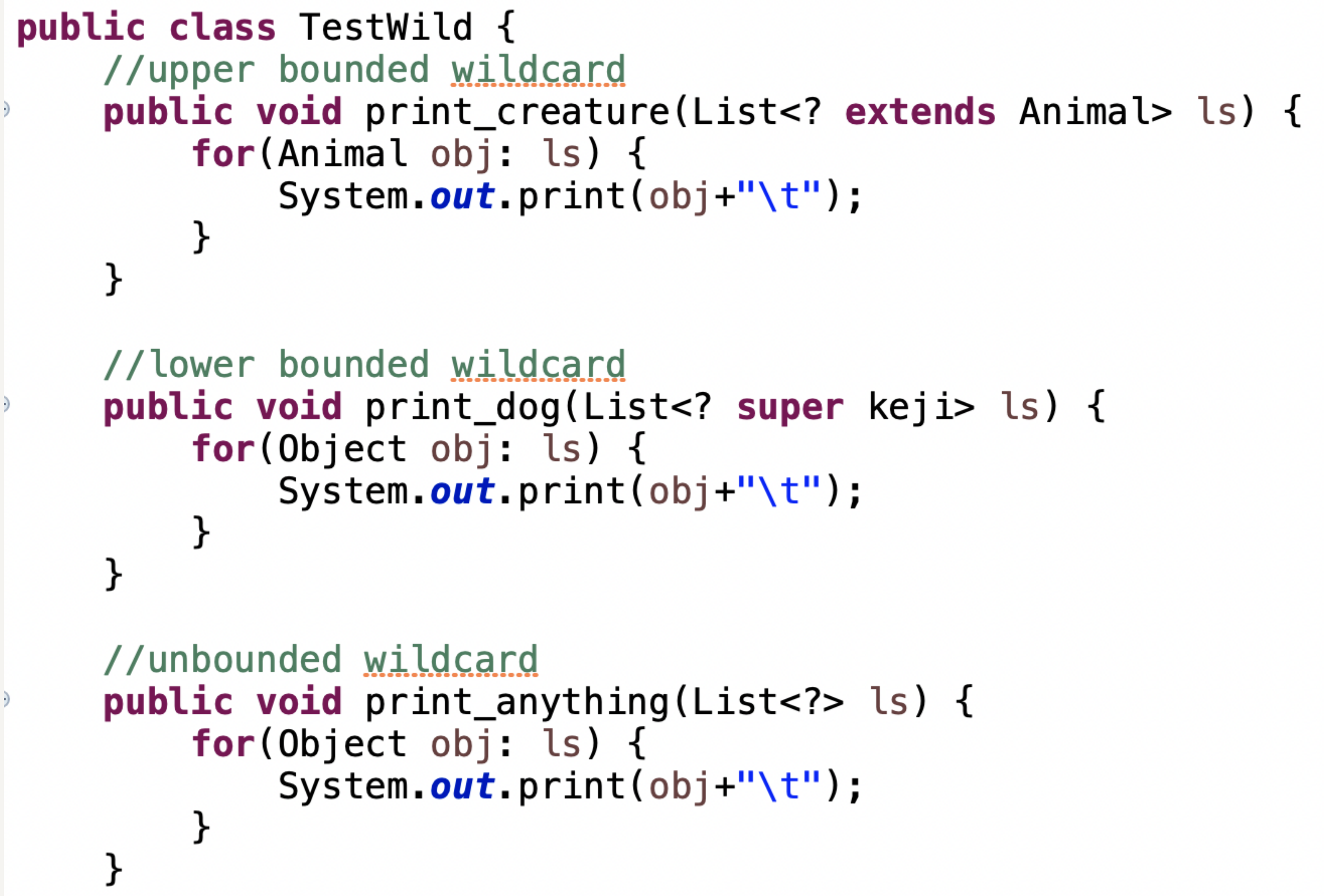


```
class Animal {

    void print_animal() {
        System.out.println("Animal class");
    }
    @Override
    public String toString() {
        return "Animal";
    }
}

class Dog extends Animal{

    void print_dog() {
        System.out.println("Dog class");
    }
    @Override
    public String toString() {
        return "Dog";
    }
}
```



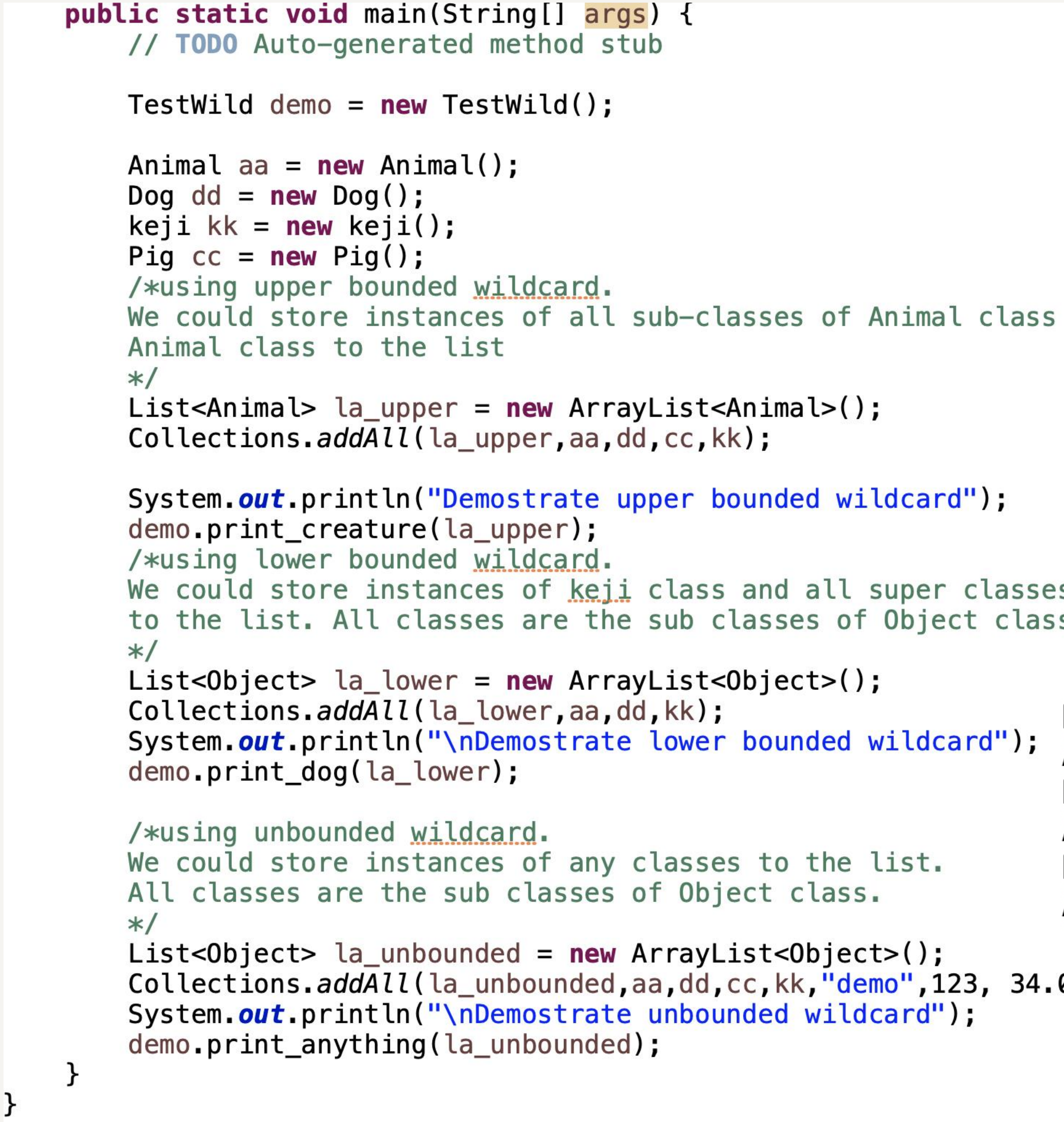
```
class keji extends Dog{
    void print_keji() {
        System.out.println("Keji class");
    }
    @Override
    public String toString() {
        return "keji";
    }
}

class Pig extends Animal{

    void print_Pig() {
        System.out.println("Pig class");
    }
    @Override
    public String toString() {
        return "Pig";
    }
}
```

Using Test class to demonstrate wildcards



```java
public class TestWild {
    //upper bounded wildcard
    public void print_creature(List<? extends Animal> ls) {
        for(Animal obj: ls) {
            System.out.print(obj+"\t");
        }
    }

    //lower bounded wildcard
    public void print_dog(List<? super keji> ls) {
        for(Object obj: ls) {
            System.out.print(obj+"\t");
        }
    }

    //unbounded wildcard
    public void print_anything(List<?> ls) {
        for(Object obj: ls) {
            System.out.print(obj+"\t");
        }
    }
```

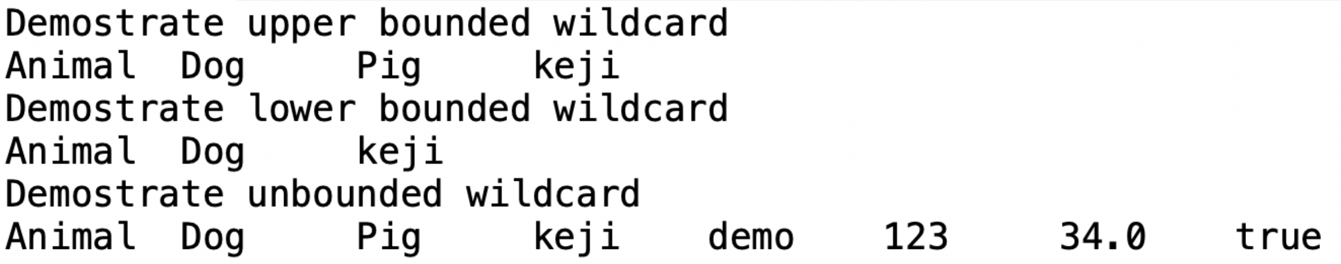
The main function in Test class



```java
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        TestWild demo = new TestWild();

        Animal aa = new Animal();
        Dog dd = new Dog();
        keji kk = new keji();
        Pig cc = new Pig();
        /*using upper bounded wildcard.
        We could store instances of all sub-classes of Animal class and those of
        Animal class to the list
        */
        List<Animal> la_upper = new ArrayList<Animal>();
        Collections.addAll(la_upper,aa,dd,cc,kk);

        System.out.println("Demostrate upper bounded wildcard");
        demo.print_creature(la_upper);
        /*using lower bounded wildcard.
        We could store instances of keji class and all super classes of it
        to the list. All classes are the sub classes of Object class.
        */
        List<Object> la_lower = new ArrayList<Object>();
        Collections.addAll(la_lower,aa,dd,kk);
        System.out.println("\nDemostrate lower bounded wildcard");
        demo.print_dog(la_lower);

        /*using unbounded wildcard.
        We could store instances of any classes to the list.
        All classes are the sub classes of Object class.
        */
        List<Object> la_unbounded = new ArrayList<Object>();
        Collections.addAll(la_unbounded,aa,dd,cc,kk,"demo",123, 34.0,true);
        System.out.println("\nDemostrate unbounded wildcard");
        demo.print_anything(la_unbounded);
    }
}
```

The whole jave file is on Moodle, TestWild.java

Output



```
Demostrate upper bounded wildcard
Animal  Dog     Pig     keji
Demostrate lower bounded wildcard
Animal  Dog     keji
Demostrate unbounded wildcard
Animal  Dog     Pig     keji    demo    123     34.0    true
```

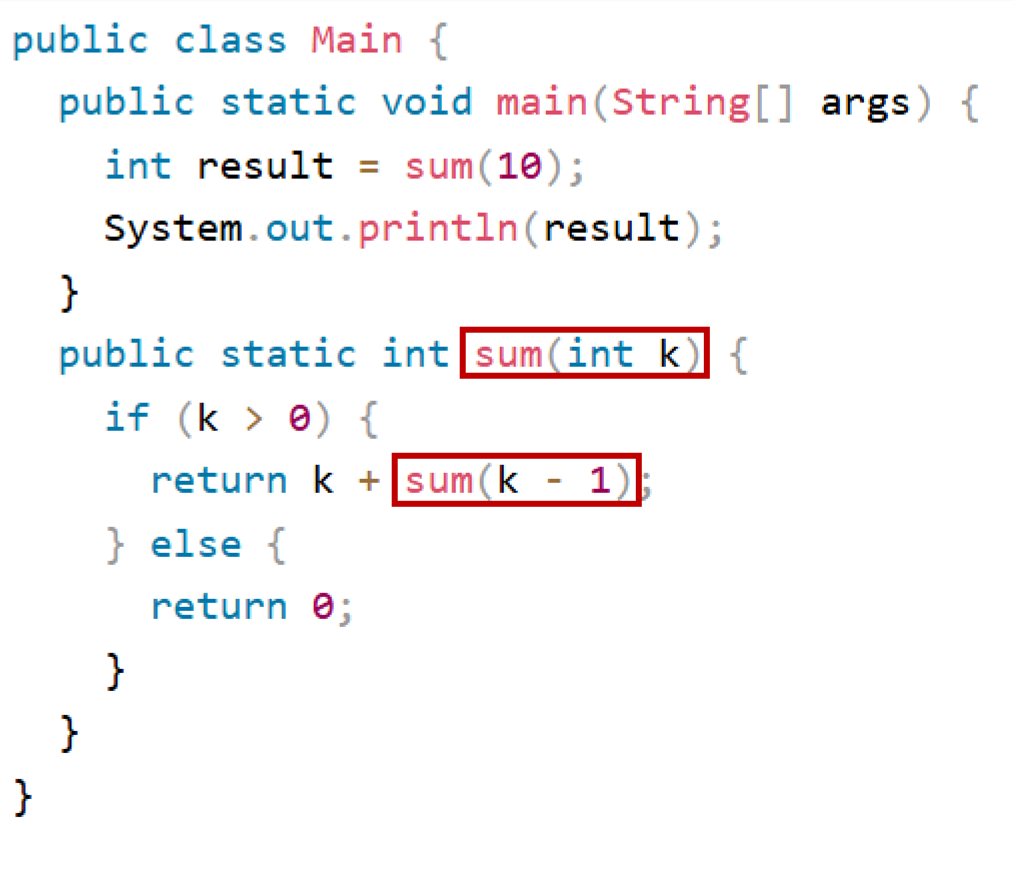
Recursion

Java Recursion

- Recursion is the technique of making a function call itself.
- This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion Example

- recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

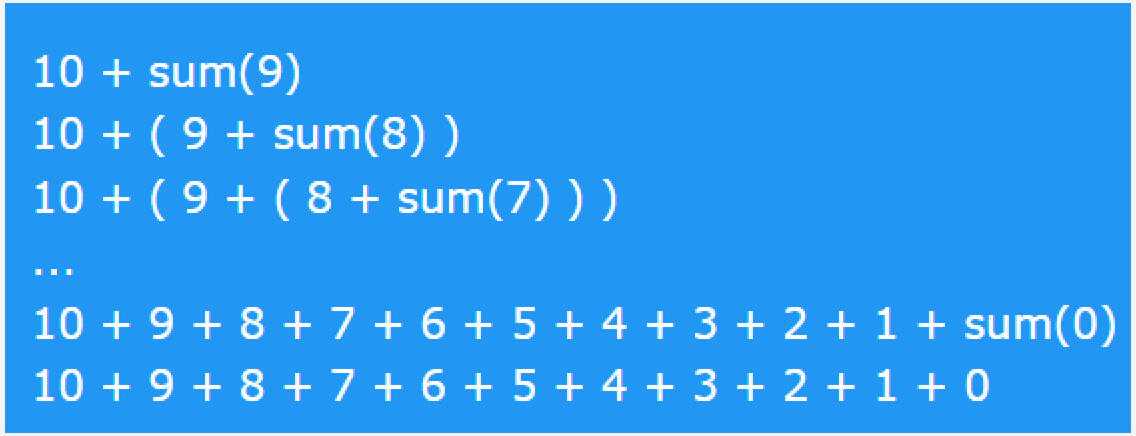


```java
public class Main {
  public static void main(String[] args) {
    int result = sum(10);
    System.out.println(result);
  }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k - 1);
    } else {
      return 0;
    }
  }
}
```

Output: 55

Example Explained

- When the sum() function is called, it adds parameter k to the sum of all numbers smaller than k and returns the result. When k becomes 0, the function just returns 0. When running, the program follows these steps:



```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + ( 9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Halting Condition

- Every recursive function should have a halting condition, which is the condition where the function stops calling itself.
- In this example, the halting condition is when the parameter k becomes 0.

End