AMRITA
VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

19CSE204 Object Oriented Paradigm 2-0-3-3

Java Collections
Generics

Java Generics

- The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.
- The Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).
- This helps us to reuse our code.
- Before generics, we store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

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

Advantage of Java Generics

- **Type-safety:** We can hold only a single type of objects in generics. It doesn?t allow to store other objects. Without Generics, we can store any type of objects.
- **Type casting is not required:** There is no need to typecast the object. Before Generics, we need to type cast.
- **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Advantage of Java Generics

```
List list = new ArrayList();
list.add(10);
list.add("10");
//With Generics, it is required to specify the type o
f object we need to store.
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.
Without Generics, we can store any type of objects.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Compile-Time Checking: It is checked at compile time so probl
will not occur at runtime. The good programming strategy says
far better to handle the problem at compile time than runtime.

Type casting is not required: There is no need to typecast the object.
Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
//After Generics, we don't need to typecast the object.
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

- **Java Generic Class**

```
class GenericsClass<T> {…}
// To create an instance of generic class
GenericsClass <Type> obj = new GenericsClass <Type>()
```

Note: In Parameter type we can not use primitives like 'int','char' or 'double'.

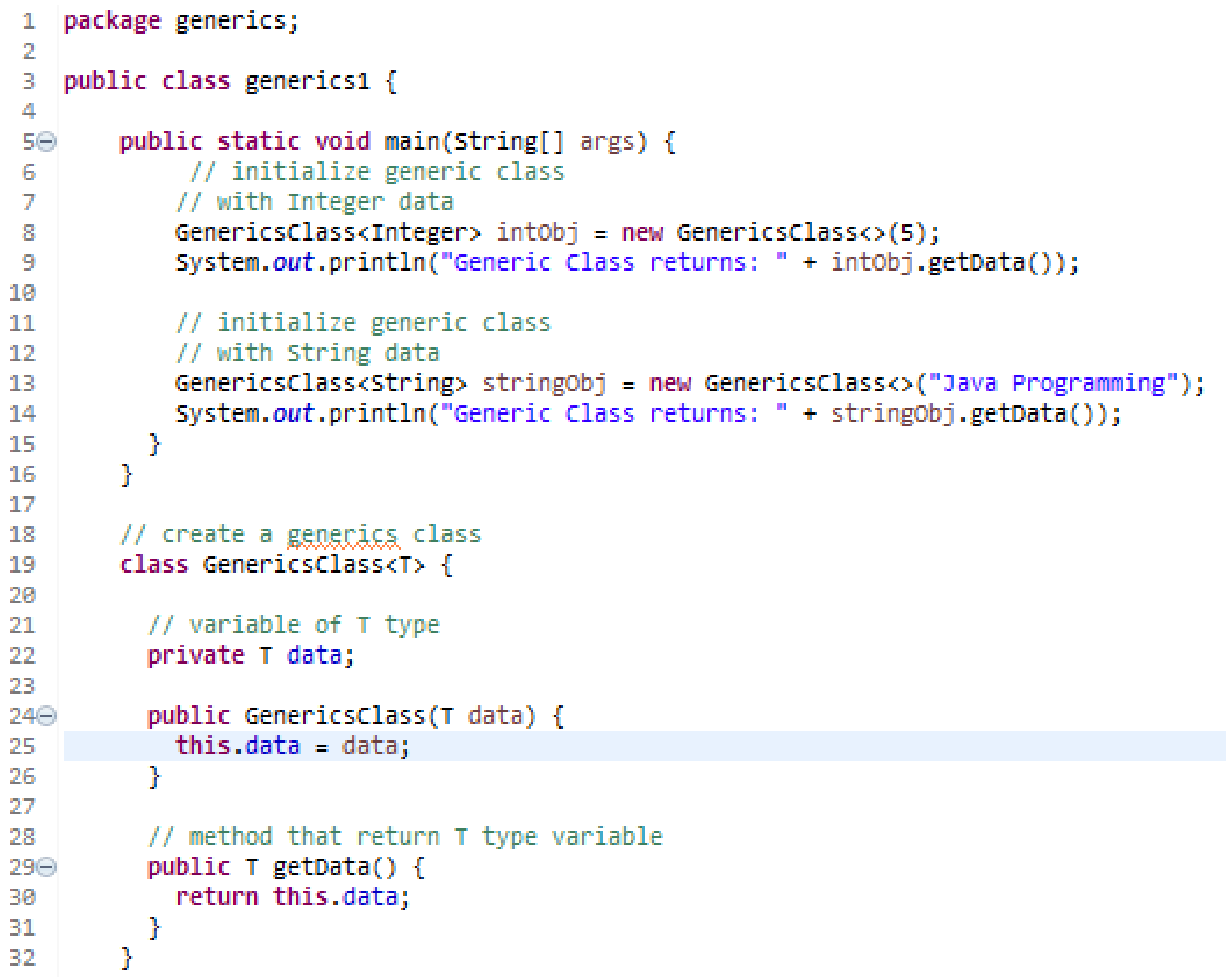
- **Java Generics Method**

```
// create a public generics method and calling it
public <T> void genericsMethod(T data) {…}
object.<T>genericsMethod("Java Programming");
```

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

Create a Generics Class

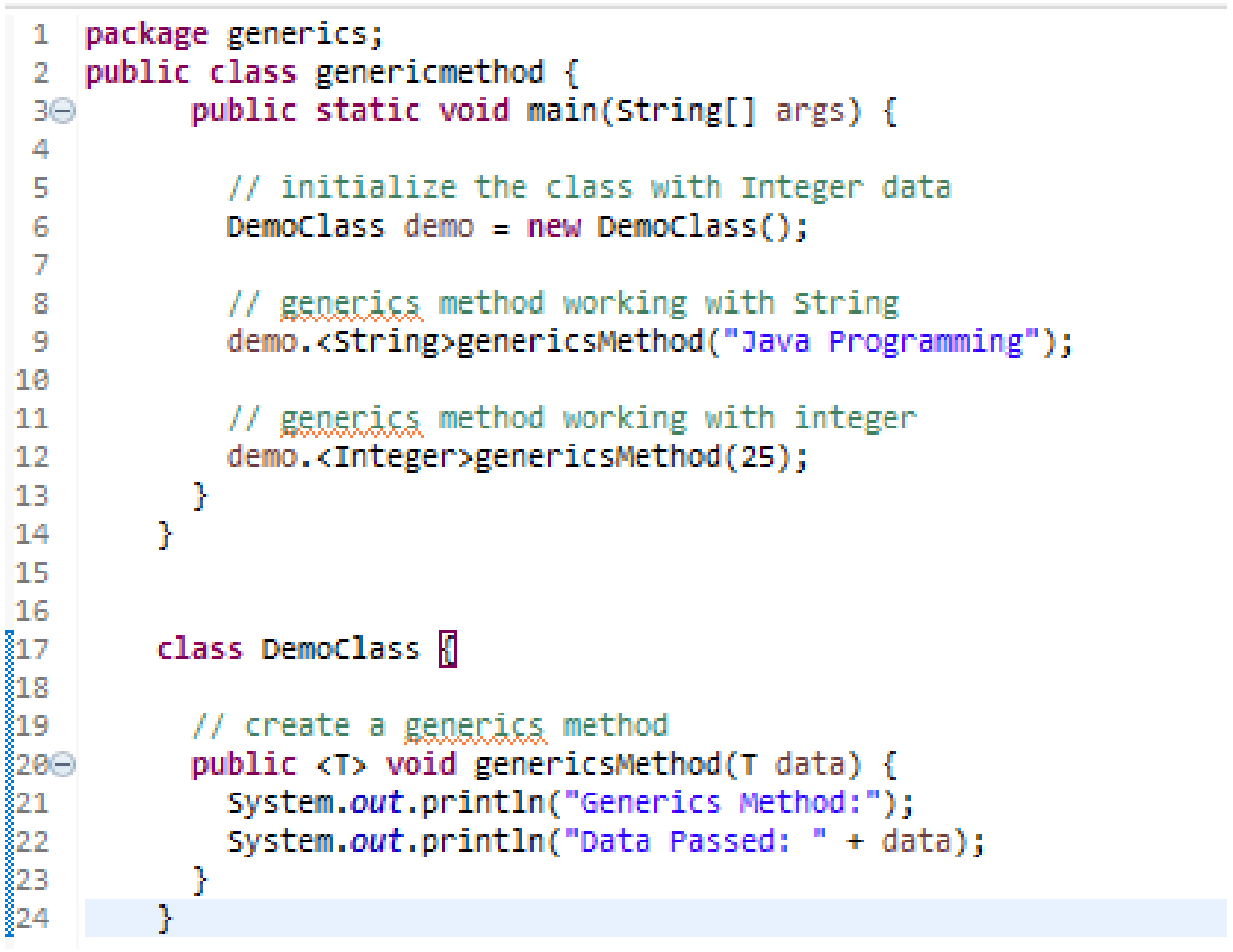


```java
package generics;

public class generics1 {

    public static void main(String[] args) {
         // initialize generic class
        // with Integer data
        GenericsClass<Integer> intObj = new GenericsClass<>(5);
        System.out.println("Generic Class returns: " + intObj.getData());

        // initialize generic class
        // with String data
        GenericsClass<String> stringObj = new GenericsClass<>("Java Programming");
        System.out.println("Generic Class returns: " + stringObj.getData());
      }
    }

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

      // variable of T type
      private T data;

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

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

Output

```
Generic Class returns: 5
Generic Class returns: Java Programming
```

Create a Generics Method

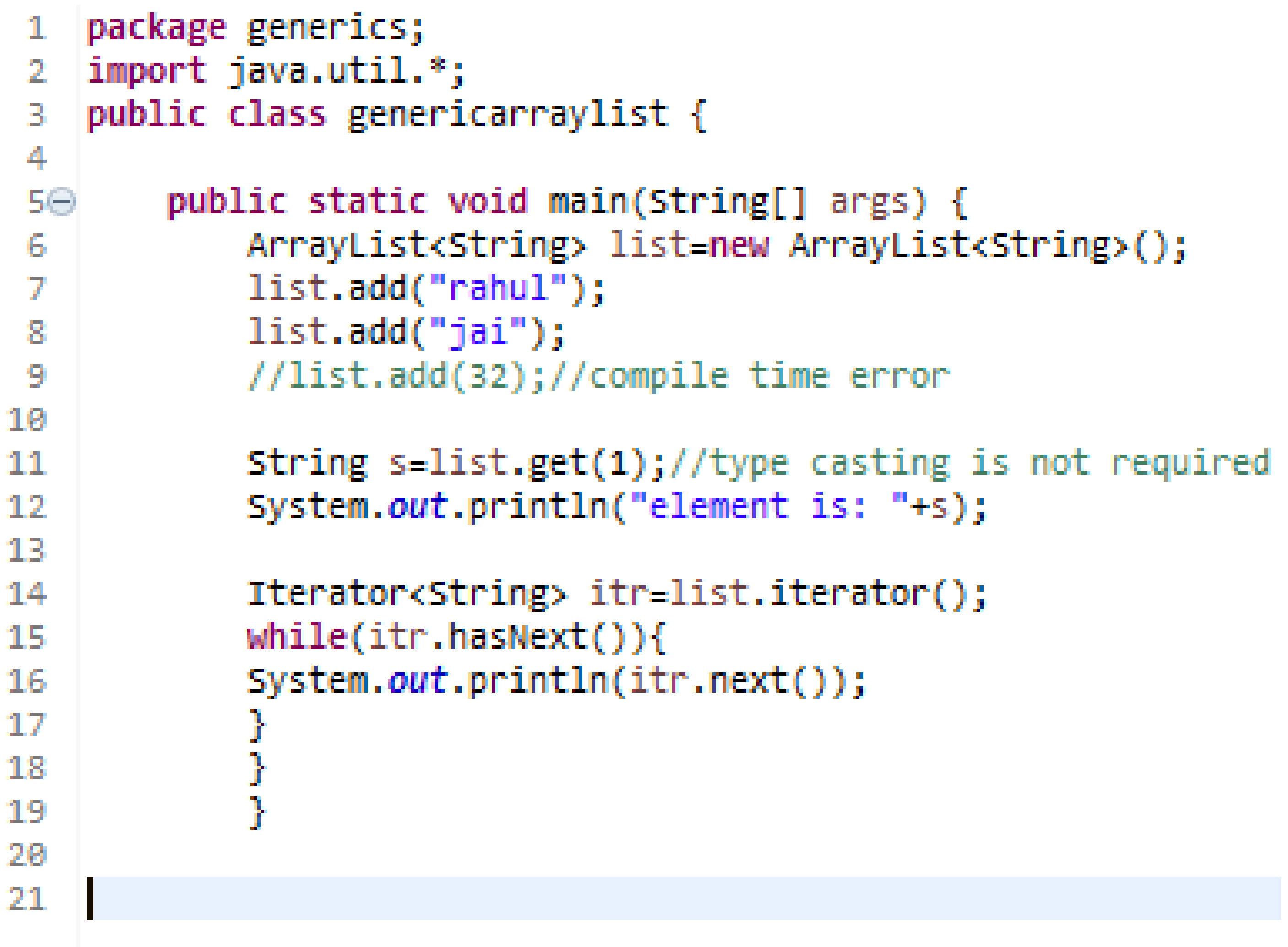


```java
package generics;
public class genericmethod {
	public static void main(String[] args) {

		// initialize the class with Integer data
		DemoClass demo = new DemoClass();

		// generics method working with String
		demo.<String>genericsMethod("Java Programming");

		// generics method working with integer
		demo.<Integer>genericsMethod(25);
	}
    }


    class DemoClass {

	// create a generics method
	public <T> void genericsMethod(T data) {
		System.out.println("Generics Method:");
		System.out.println("Data Passed: " + data);
	}
    }
```

Output

```
Generics Method:
Data Passed: Java Programming
Generics Method:
Data Passed: 25
```

Generics with ArrayList

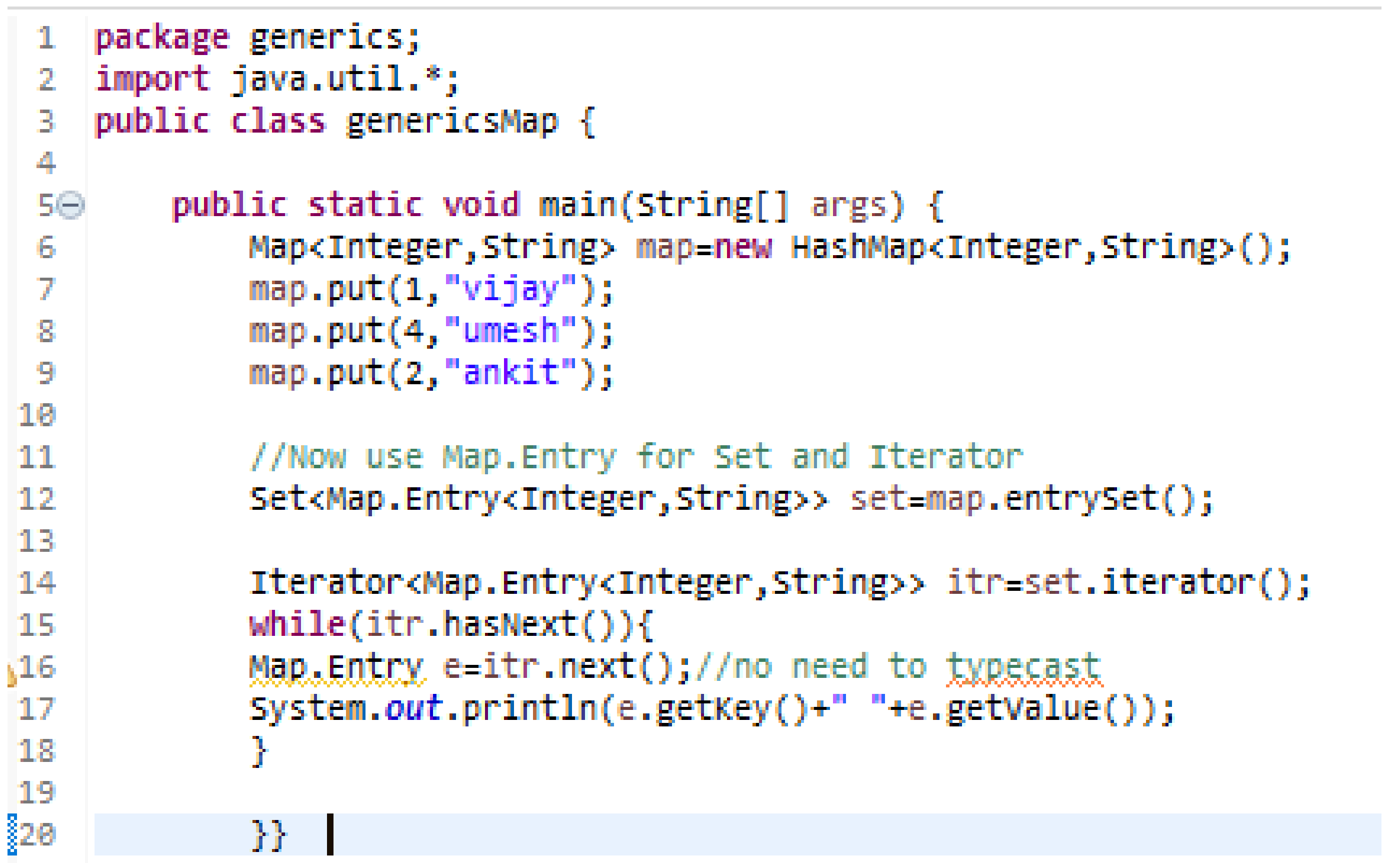


```java
package generics;
import java.util.*;
public class genericarraylist {

    public static void main(String[] args) {
        ArrayList<String> list=new ArrayList<String>();
        list.add("rahul");
        list.add("jai");
        //list.add(32);//compile time error

        String s=list.get(1);//type casting is not required
        System.out.println("element is: "+s);

        Iterator<String> itr=list.iterator();
        while(itr.hasNext()){
        System.out.println(itr.next());
        }
        }
        }

```

Output

```
element is: jai
rahul
jai
```

Java Generics using Map

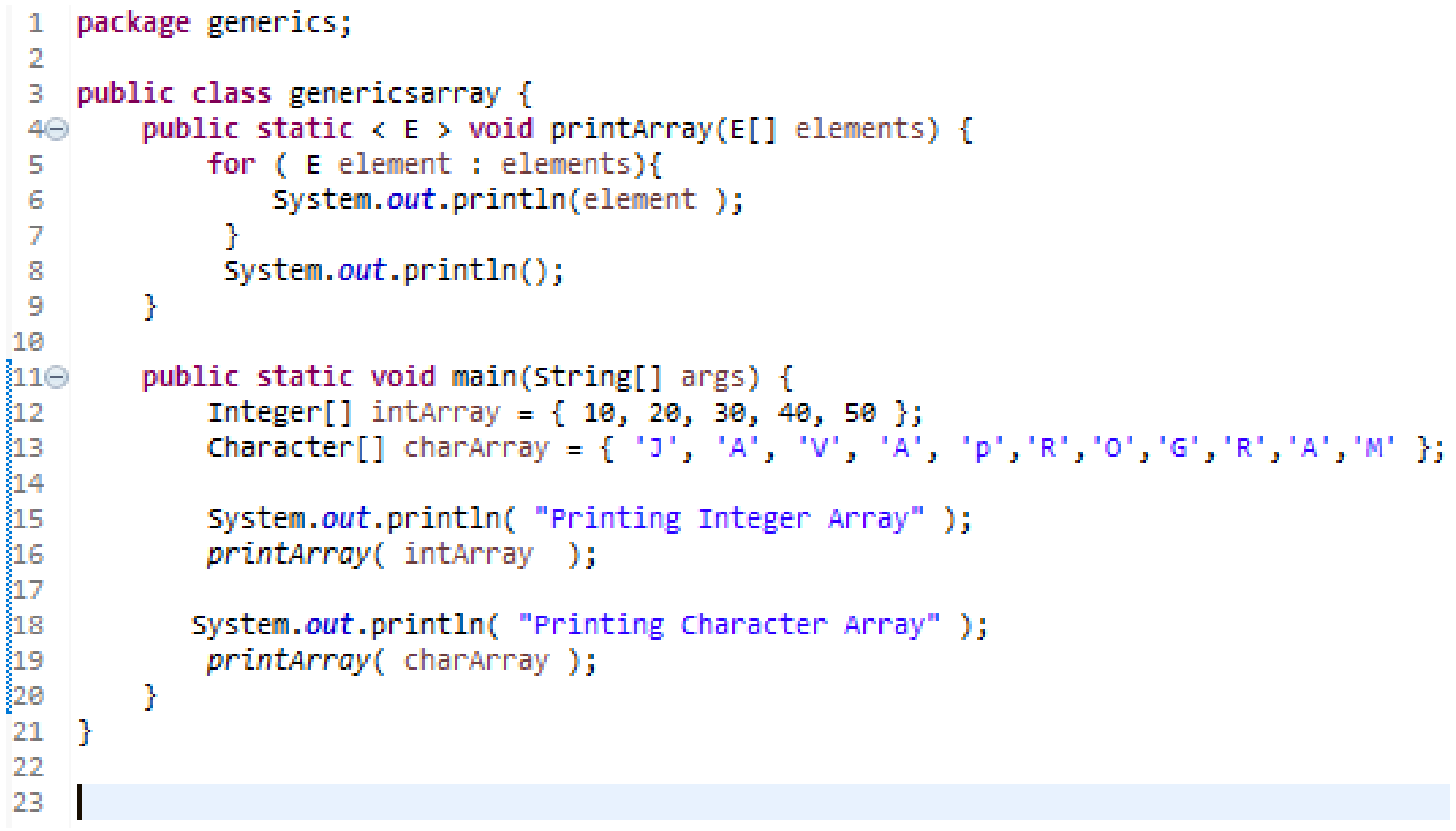


```java
package generics;
import java.util.*;
public class genericsMap {

    public static void main(String[] args) {
        Map<Integer,String> map=new HashMap<Integer,String>();
        map.put(1,"vijay");
        map.put(4,"umesh");
        map.put(2,"ankit");

        //Now use Map.Entry for Set and Iterator
        Set<Map.Entry<Integer,String>> set=map.entrySet();

        Iterator<Map.Entry<Integer,String>> itr=set.iterator();
        while(itr.hasNext()){
        Map.Entry e=itr.next();//no need to typecast
        System.out.println(e.getKey()+" "+e.getValue());
        }

        }}
```

Output

```
1 vijay
2 ankit
4 umesh
```

Java generic method to print array elements.



```java
package generics;

public class genericsarray {
    public static < E > void printArray(E[] elements) {
        for ( E element : elements){
            System.out.println(element );
         }
         System.out.println();
    }

    public static void main(String[] args) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', 'p','R','O','G','R','A','M' };

        System.out.println( "Printing Integer Array" );
        printArray( intArray  );

       System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}


```

Output

```
Printing Integer Array
10
20
30
40
50

Printing Character Array
J
A
V
A
p
R
O
G
R
A
M
```

Wildcard in Java Generics

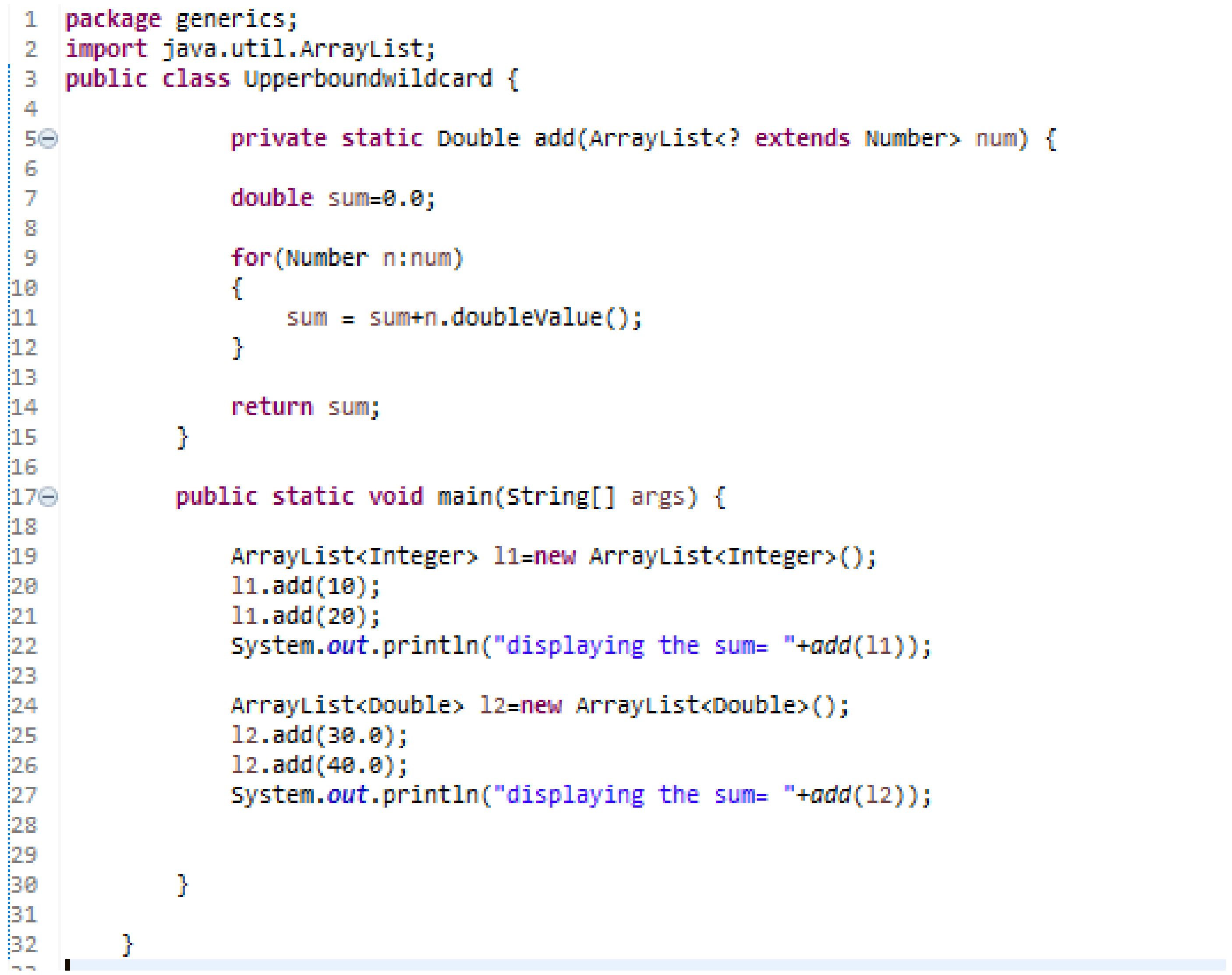
- The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.
  - We can use a wildcard as a **type of 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 supertype**.
- **Upper Bounded Wildcards**
  - The purpose of upper bounded wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.

```
List<? extends Number>
```

- **Unbounded Wildcards**
  - The unbounded wildcard type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -
  - When the given method is implemented by using the functionality provided in the Object class.
  - When the generic class contains the methods that don't depend on the type parameter.
- **Lower Bounded Wildcards**
  - The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.

```
List<? super Integer>
```

Generics : Upper Bounded Wildcards

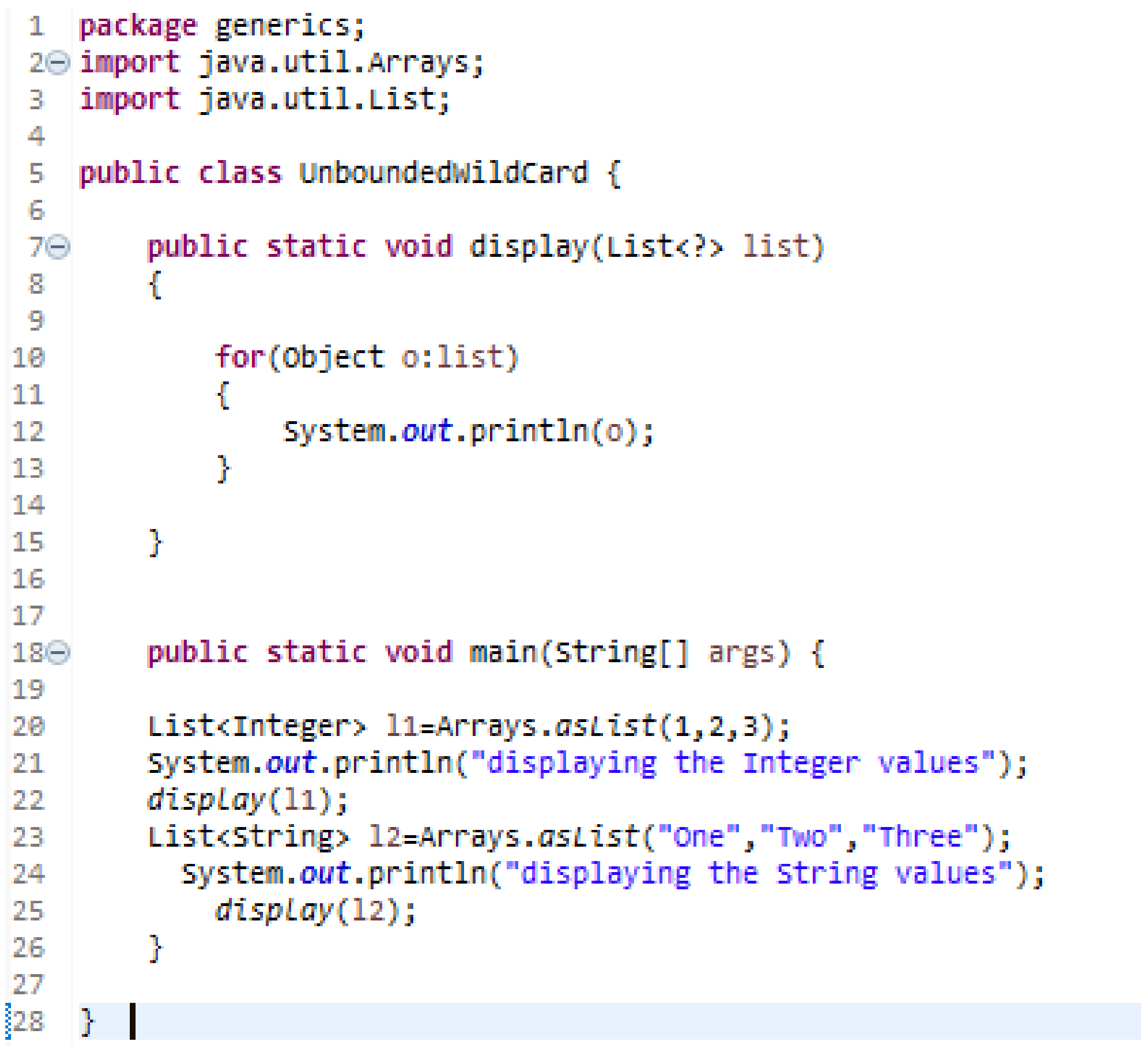


```java
package generics;
import java.util.ArrayList;
public class Upperboundwildcard {

            private static Double add(ArrayList<? extends Number> num) {

            double sum=0.0;

            for(Number n:num)
            {
                sum = sum+n.doubleValue();
            }

            return sum;
        }

        public static void main(String[] args) {

            ArrayList<Integer> l1=new ArrayList<Integer>();
            l1.add(10);
            l1.add(20);
            System.out.println("displaying the sum= "+add(l1));

            ArrayList<Double> l2=new ArrayList<Double>();
            l2.add(30.0);
            l2.add(40.0);
            System.out.println("displaying the sum= "+add(l2));


        }

    }
```

Output

```
displaying the sum= 30.0
displaying the sum= 70.0
```

Generics : UnBounded Wildcards

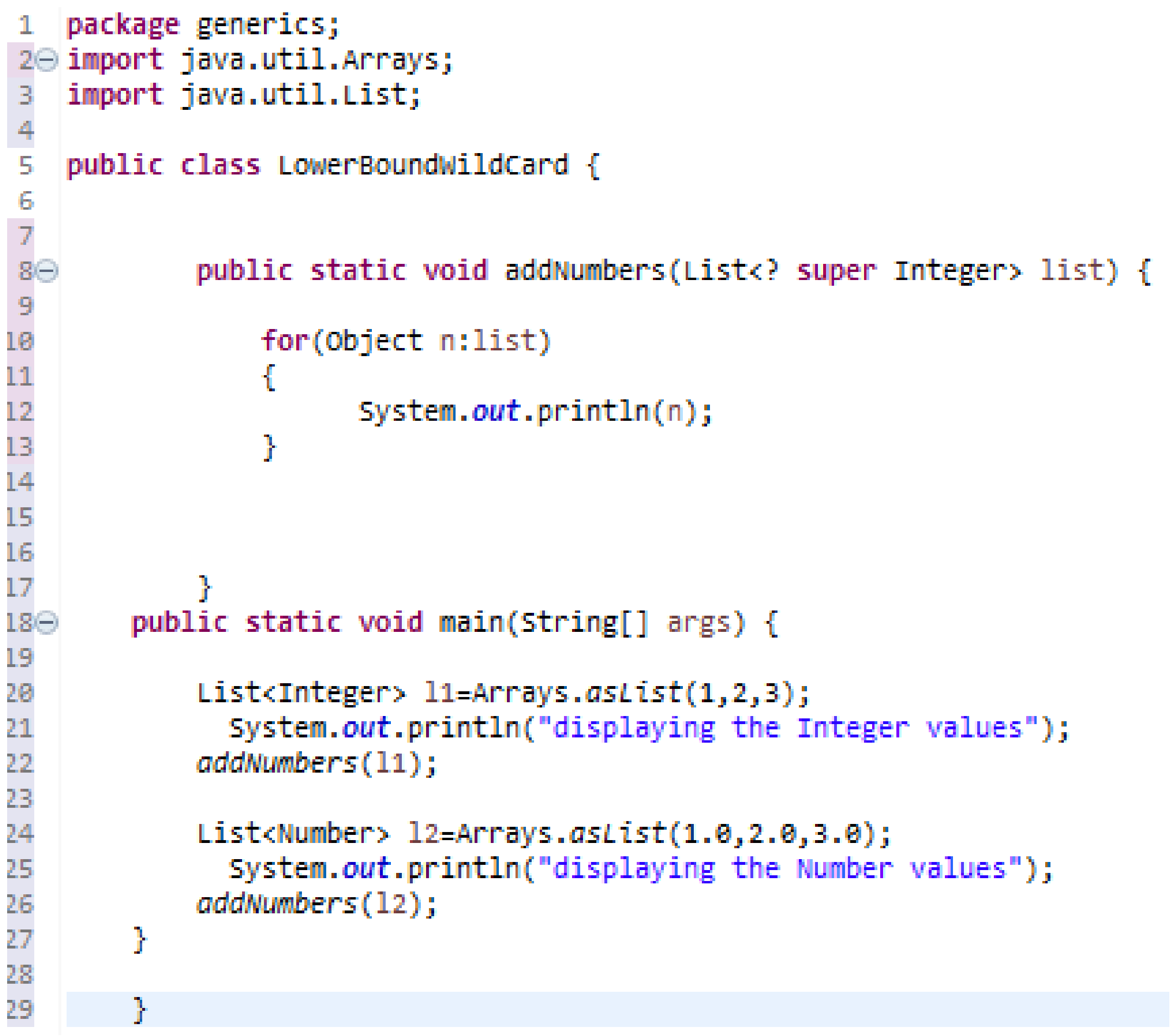


```
package generics;
import java.util.Arrays;
import java.util.List;

public class UnboundedWildCard {

    public static void display(List<?> list)
    {

        for(Object o:list)
        {
            System.out.println(o);
        }

    }


    public static void main(String[] args) {

    List<Integer> l1=Arrays.asList(1,2,3);
    System.out.println("displaying the Integer values");
    display(l1);
    List<String> l2=Arrays.asList("One","Two","Three");
      System.out.println("displaying the String values");
        display(l2);
    }

}
```

Output

```
displaying the Integer values
1
2
3
displaying the String values
One
Two
Three
```

Generics : lower Bounded Wildcards



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

public class LowerBoundWildCard {


        public static void addNumbers(List<? super Integer> list) {

            for(Object n:list)
            {
                  System.out.println(n);
            }



        }
    public static void main(String[] args) {

        List<Integer> l1=Arrays.asList(1,2,3);
          System.out.println("displaying the Integer values");
        addNumbers(l1);

        List<Number> l2=Arrays.asList(1.0,2.0,3.0);
          System.out.println("displaying the Number values");
        addNumbers(l2);
    }

    }
```

Output

```
displaying the Integer values
1
2
3
displaying the Number values
1.0
2.0
3.0
```

Namah Shivaya