



Vietnam National University of HCMC
International University
School of Computer Science and Engineering



Generic Classes and Methods

(IT069IU)

Nguyen Trung Ky

 ntky@hcmiu.edu.vn

 it.hcmiu.edu.vn/user/ntky

Previously,



- **Polymorphism**
 - Method overriding and overloading in Inheritance
 - Zoo Example
 - Company Payroll Example
- **Abstraction**
 - Abstract Class
 - Abstract Method
 - Why we need abstract class
 - Examples:
 - Zoo Example
 - Company Payroll Example
- **Interface**
 - Interface in real life examples
 - Upgrade Company Payroll with Invoices Example
- Abstract vs Interface

Agenda's today

- Generic
 - Generic Method
 - Generic Class
 - Bounded type parameters



When does it start?

GENERIC JAVA (1998)



Philip Wadler

Gilad Bracha

Martin Odersky

David Stoutamire

When does it start?



Adding Generics to the Java Programming Language: Participant Draft Specification

Gilad Bracha, Sun Microsystems
Norman Cohen, IBM
Christian Kemper, Inprise
Steve Marx, WebGain
Martin Odersky, EPFL
Sven-Eric Panitz, Software AG
David Stoutamire, Sun Microsystems
Kresten Thorup, Aarhus University
Philip Wadler, Lucent Technologies

April 27, 2001

1 Summary

We propose to add generic types and methods to the Java programming language.

The main benefit of adding genericity to the Java programming language lies in the added expressiveness and safety that stems from making type parameters explicit and making type casts implicit. This is crucial for using libraries such as collections in a flexible, yet safe way.

The proposed extension is designed to be fully backwards compatible with the current language, making the transition from non-generic to generic programming very easy. In particular, one can retrofit existing library classes with generic interfaces without changing their code.

The present specification evolved from the GJ proposal which has been presented and motivated in a previous paper [BOSW98].

The rest of this specification is organized as follows. Section 2 explains how parameterized types are declared and used. Section 3 explains polymorphic methods. Section 4 explains how parameterized types integrate with exceptions. Section 5 explains what changes in Java's expression constructs. Section 6 explains how the extended language translated into the class file format of the Java Virtual Machine. Section 7 explains how generic type information is stored in classfiles. Where possible, we follow the format and conventions the Java Language Specification (JLS) [GJSB96].

Generic



What problem does it solve?

The Problem



Generic

- Introduce type parameters for **classes and methods**, which are symbols can be substituted for any concrete type.
- The benefit is to **eliminate the need to create multiple versions of methods or classes** for **various data types**.
 - > Use one version for all reference data types.

Generic Methods





```
1 // Fig. 21.1: OverloadedMethods.java
2 // Printing array elements using overloaded methods.
3 public class OverloadedMethods
4 {
5     public static void main( String[] args )
6     {
7         // create arrays of Integer, Double and Character
8         Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
9         Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
10        Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
11
12        System.out.println( "Array integerArray contains:" );
13        printArray( integerArray ); // pass an Integer array
14        System.out.println( "\nArray doubleArray contains:" );
15        printArray( doubleArray ); // pass a Double array
16        System.out.println( "\nArray characterArray contains:" );
17        printArray( characterArray ); // pass a Character array
18    } // end main
19
20    // method printArray to print Integer array
21    public static void printArray( Integer[] inputArray )
22    {
23        // display array elements
24        for ( Integer element : inputArray )
25            System.out.printf( "%s ", element );
26
27        System.out.println();
28    } // end method printArray
29
30    // method printArray to print Double array
31    public static void printArray( Double[] inputArray )
32    {
33        // display array elements
34        for ( Double element : inputArray )
35            System.out.printf( "%s ", element );
36
37        System.out.println();
38    } // end method printArray
39
```

```
40 // method printArray to print Character array
41 public static void printArray( Character[] inputArray )
42 {
43     // display array elements
44     for ( Character element : inputArray )
45         System.out.printf( "%s ", element );
46
47     System.out.println();
48 } // end method printArray
49 } // end class OverloadedMethods
```

Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O

This is the old way where we haven't learn about generic method where we have to create a different overloading methods with similar bodies to handle different data types. This is beyond ugly and repetitive!

Motivation for Generic Methods

- Overloaded methods are often used to perform similar operations on different types of data.
- Study each `printArray` method.
 - Note that the array element type appears in each method's header and for-statement header.
 - If we were to replace the element types in each method with a generic name—`T` (type parameter) by convention—then all three methods would look like the one.

Conventions

- Type parameter meaning
 - **E**: Element type in a collection
 - **K**: Key type in a map
 - **V**: Value type in a map
 - **T**: General type
 - **S, U**: Additional general types

Implementing a Generic Methods

- *Generic methods* are methods that introduce their own type parameters.
- The type parameter's scope is limited to the method where it is declared.
- The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type.

```
public <T> void printArray( T[] inputArray )  
{  
    //body  
} // end method printArray
```

Quiz 1



- Try to solve this Generic problem in Java on HackerRank website:

<https://www.hackerrank.com/challenges/java-generics/problem?isFullScreen=true>

Generic methods are a very efficient way to handle multiple datatypes using a single method. This problem will test your knowledge on Java Generic methods.

Let's say you have an integer array and a string array. You have to write a **single** method `printArray` that can print all the elements of both arrays. The method should be able to accept both integer arrays or string arrays.

You are given code in the editor. Complete the code so that it prints the following lines:

```
1
2
3
Hello
World
```

Do not use method overloading because your answer will not be accepted.

```
1 > ...
4
5 class Printer
6 {
7     //Write your code here
8
9 }
10
11 v public class Solution {
12
13 |
14     public static void main( String args[] ) {
15         Printer myPrinter = new Printer();
16         Integer[] intArray = { 1, 2, 3 };
17         String[] stringArray = {"Hello", "World"};
18         myPrinter.printArray(intArray);
19         myPrinter.printArray(stringArray);
20         int count = 0;
21
22         for (Method method : Printer.class.getDeclaredMethods()) {
23             String name = method.getName();
24
25             if(name.equals("printArray"))
26                 count++;
27         }
28
29         if(count > 1)System.out.println("Method overloading is not allowed!");
30
31     }
32 }
```

```
1 // Fig. 21.3: GenericMethodTest.java
2 // Printing array elements using generic method printArray.
3
4 public class GenericMethodTest
5 {
6     public static void main( String[] args )
7     {
8         // create arrays of Integer, Double and Character
9         Integer[] intArray = { 1, 2, 3, 4, 5 };
10        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
11        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
12
13        System.out.println( "Array integerArray contains:" );
14        printArray( integerArray ); // pass an Integer array
15        System.out.println( "\nArray doubleArray contains:" );
16        printArray( doubleArray ); // pass a Double array
17        System.out.println( "\nArray characterArray contains:" );
18        printArray( characterArray ); // pass a Character array
19    } // end main
20
21    // generic method printArray
22    public static < T > void printArray( T[] inputArray )
23    {
24        // display array elements
25        for ( T element : inputArray )
26            System.out.printf( "%s ", element );
27
28        System.out.println();
29    } // end method printArray
30 } // end class GenericMethodTest
```

Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O

With generic methods, we can use only method to handle different type of data types instead of creating many overloaded method! Amazing!

Generic Class



Generic Classes



A Generic class simply means that the items or functions in that class can be generalized with the parameter (example T) to specify that we can add any type as a parameter in place of T like Integer, Character, String, Double or any other user-defined type.

```
class Solution<T>
{
    T data;
    public static T getData(){
        return data;
    }
}
```

To create object/instance of Generic class

- **Create a new generic class** with a type placeholder T:

```
public class ClassName <T>{  
...  
}
```

- **Use that generic class** to create an object from it and specific what datatype will be replace T.

```
ClassName <Type> objectName = new ClassName <Type>();
```

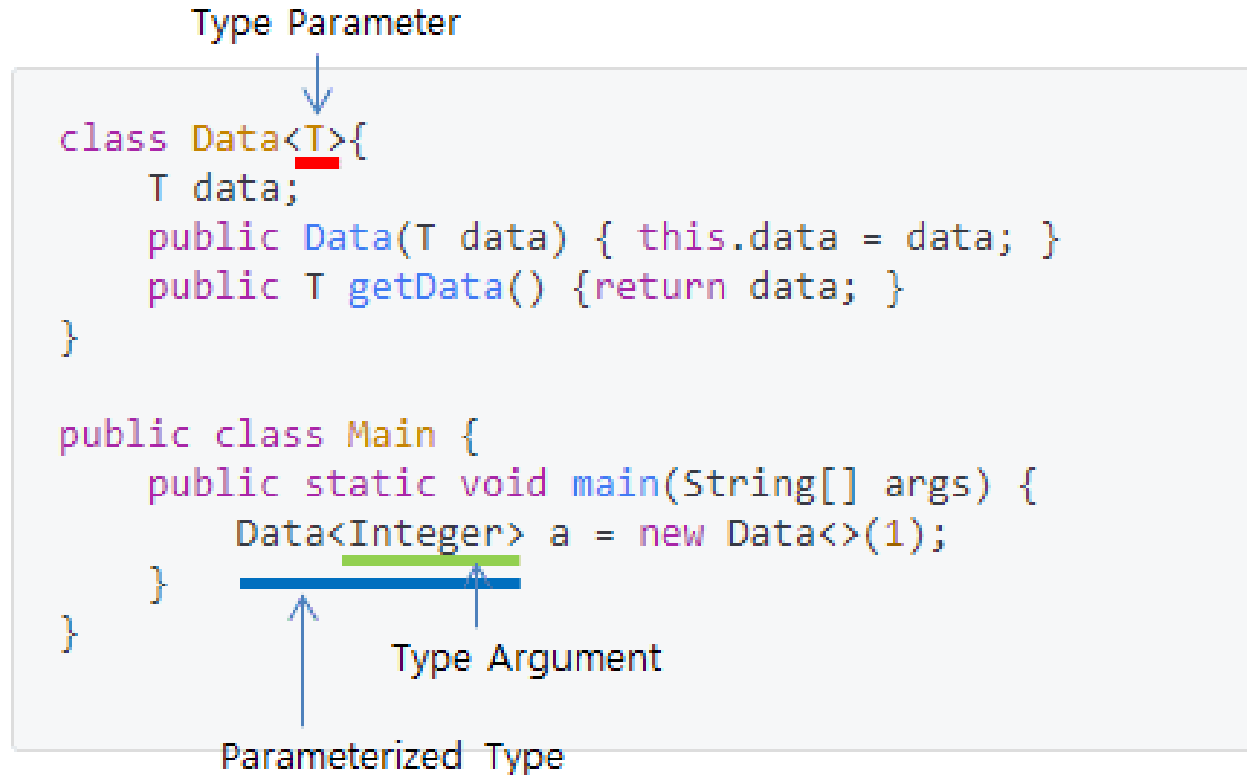
Simple Generic Class Example

Type Parameter

```
class Data<T>{  
    T data;  
    public Data(T data) { this.data = data; }  
    public T getData() {return data; }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Data<Integer> a = new Data<>(1);  
    }  
}
```

Type Argument

Parameterized Type



Another Example of Generic Class

```
public class Wallet<T> {  
    private T balance;  
  
    Wallet(T amount) {  
        this.balance = amount;  
    }  
  
    public void setBalance(T amount)  
    {  
        this.balance = amount;  
    }  
  
    public T getBalance() {  
        return this.balance;  
    }  
}
```

```
public class TestWallet {  
    public static void main(String[] args) {  
        Wallet<Integer> Tom = new Wallet<Integer>(0);  
        Wallet<Double> Jerry = new Wallet<Double>(0.0);  
        Tom.setBalance(10);  
        Jerry.setBalance(2.5);  
        System.out.println(Tom.getBalance());  
        System.out.println(Jerry.getBalance());  
    }  
}
```

10

2.5

Single Type Vs Multiple Type Parameters



```
class Solution<T>
{
    T data;
    public static T getData(){
        return data;
    }
}
```

Single Type Parameter

```
public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()    { return key; }
    public V getValue() { return value; }
}
```

Multiple Type Parameter

Bounded Type Parameters

Limit the type parameters to specific group of subclass!

Problem



```
public class MyNumberClass <T>{  
    T myNumber;  
  
    public MyNumberClass(T myNumber) {  
        this.myNumber = myNumber;  
    }  
  
    public double square(){  
        return myNumber * myNumber;  
    }  
}
```

! Operator '*' cannot be applied to 'T', 'T'

```
public class Main {  
    public static void main(String[] args) {  
        MyNumberClass<Integer> myNumber = new MyNumberClass<>(4);  
        System.out.println(myNumber.square());  
    }  
}
```

Solve it with Bound Type Parameters



```
public class MyNumberClass <T extends Number>{  
    T myNumber;  
  
    public MyNumberClass(T myNumber) {  
        this.myNumber = myNumber;  
    }  
  
    public double square(){  
        return myNumber.intValue() * myNumber.doubleValue();  
    }  
}
```

Output:

16.0

```
public class Main {  
    public static void main(String[] args) {  
        MyNumberClass<Integer> myNumber = new MyNumberClass<>(4);  
        System.out.println(myNumber.square());  
    }  
}
```


Bounded Type Parameters

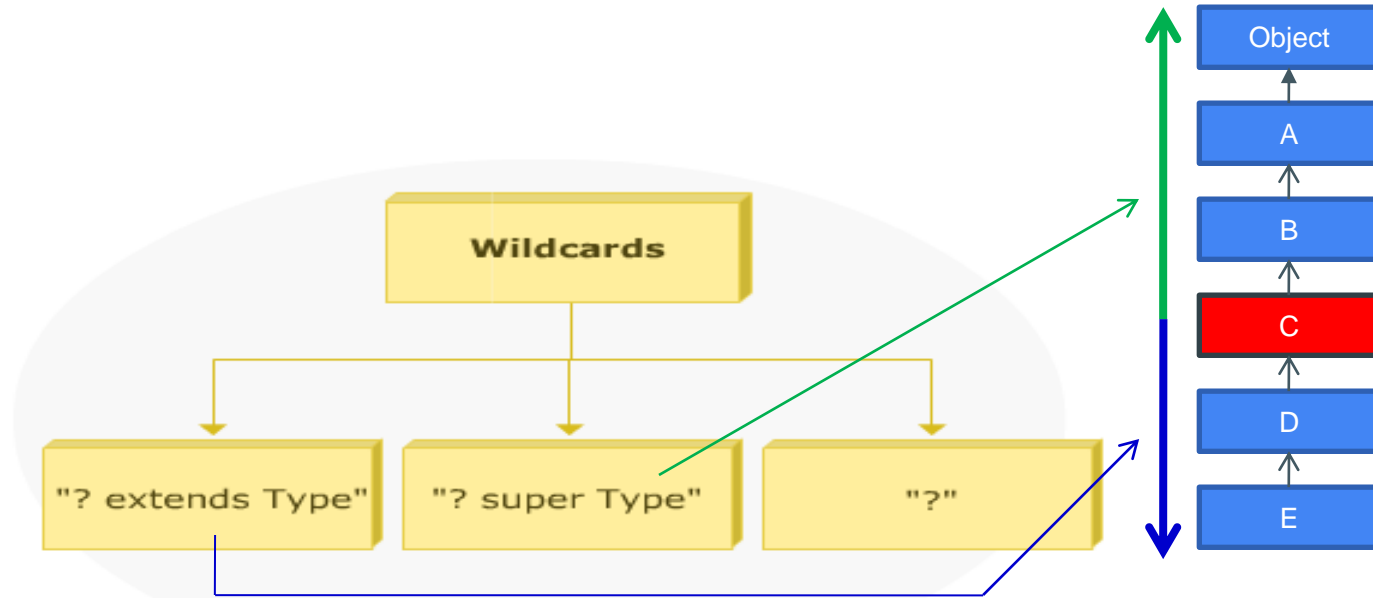


```
public class MyNumberClass <T extends Number>{  
    T myNumber;  
    public MyNumberClass(T myNumber) {  
        this.myNumber = myNumber;  
    }  
  
    public T getNumber(){  
        return myNumber;  
    }  
}
```

By using the keyword “extends”, we can force the type parameters to have the requirement to be a subclass of that super class. Like in this example, the MyNumberClass can only take subclasses types of the superclass Number!

```
public class Main {  
    public static void main(String[] args) {  
        MyNumberClass<Integer> myNumber = new MyNumberClass<>(4);  
        MyNumberClass<Double> myDouble = new MyNumberClass<>(7.5);  
  
        //MyNumberClass<String> myString = new MyNumberClass<String>("Hello");  
        //MyNumberClass<Char> myChar = new MyNumberClass<Char>('@');  
    }  
}
```

Bounded Type Parameters: Using wildcards



The ? stands for an unknown type

? **extends** Type : a *bounded wildcard*. Type is upper bound

? **super** Type : a *bounded wildcard*. Type is lower bound

Wildcards



- The question mark (?), called the *wildcard*, represents an unknown type.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable.

Wildcards Demo.

```
/* WildCard_Demo.java */
import java.util.*;
class A
{   int a=3;
    public String toString() { return "" + a; }
}
class B extends A
{   int b=5;
    public String toString() { return "" + (a+b); }
}
class C
{   int c= 10;
    public String toString() { return "" + c; }
}
```

```
package Wildcard;

import java.util.Collection;
import java.util.Vector;

public class Wildcard_Demo {

    public Wildcard_Demo(){

    }

    static void print1(Collection<?> col) {
        for(Object o: col) System.out.println(o + " , ");
    }

    static void print2(Collection<? extends A> col) {
        for(Object o: col) System.out.println(o + " , ");
    }

    static void print3(Collection<? super B> col) {
        for(Object o: col) System.out.println(o + " , ");
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Vector VA = new Vector();
        VA.add(new A()); VA.add(new A()); VA.add(new A());

        Vector VB = new Vector();
        VB.add(new B()); VB.add(new B()); VB.add(new B());

        Vector VC = new Vector();
        VC.add(new C()); VC.add(new C()); VC.add(new C());

        Wildcard_Demo.print1(VC);System.out.println();
        Wildcard_Demo.print2(VB);System.out.println();
    }
}
```



```
}
classC [c=10] ,
classC [c=10] ,
classC [c=10] ,

classB [ a+ b=8] ,
classB [ a+ b=8] ,
classB [ a+ b=8] ,
```



Advantages of Java Generics



1. Type-Safety: One can hold only a single type of objects in generics.
2. Type Casting Is Not Required: There is no need to typecast.

//Before Generics

```
List x= new ArrayList();
```

```
x.add("India");
```

```
String s1 = (String) x.get(0);    // typecasting
```

//After Generics, typecasting of the object is not required

```
List<String> y = new ArrayList<String>();
```

```
y.add("hello");
```

```
String s2 = y.get(0);
```

Advantages of Java Generics

3. Compile -Time Checking: It checks all the errors of datatype related to generics at the time of compile-time so the issue will not occur at the time of runtime.

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

Recap

- Generic
 - Generic Class
 - Bounded type parameters
 - Generic Method



Thank you for your listening!

**“Motivation is what gets you started.
Habit is what keeps you going!”**

Jim Ryun

