



# Chapter 16:

# Generics

TAO Yida

[taoyd@sustech.edu.cn](mailto:taoyd@sustech.edu.cn)



# Objectives

ArrayList <String>  
Comparable <Circle>

- ▶ Motivation of generic methods (泛型方法)
- ▶ Declare and use generic methods
- ▶ Declare and use generic classes (泛型类)



# Recall Method Overloading

- ▶ A language feature that allows a class to have multiple methods with the same name, but different parameter lists.

```
public static void printArray(Integer[] array) {  
    for (Integer element : array) System.out.printf("%s ", element);  
    System.out.println();  
}  
  
public static void printArray(Double[] array) {  
    for (Double element : array) System.out.printf("%s ", element);  
    System.out.println();  
}  
  
public static void printArray(Character[] array) {  
    for (Character element : array) System.out.printf("%s ", element);  
    System.out.println();  
}
```



# Using overloaded methods

功能一致

```
public static void main(String[] args) {  
    Integer[] integerArray = { 1, 2, 3, 4, 5, 6 }; // autoboxing  
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5 }; // autoboxing  
    Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' }; // autoboxing  
    System.out.print("integerArray contains: ");  
    printArray(integerArray);  
    System.out.print("doubleArray contains: ");  
    printArray(doubleArray);  
    System.out.print("characterArray contains: ");  
    printArray(characterArray);  
}
```

转化为包装类

Compiler will find the appropriate method

```
integerArray contains: 1 2 3 4 5 6  
doubleArray contains: 1.1 2.2 3.3 4.4 5.5  
characterArray contains: H E L L O
```

# Looks good, but wait...

```
public static void printArray(Integer[] array) {  
    for (Integer element : array) System.out.printf("%s ", element);  
    System.out.println();  
}  
  
public static void printArray(Double[] array) {  
    for (Double element : array) System.out.printf("%s ", element);  
    System.out.println();  
}  
  
public static void printArray(Character[] array) {  
    for (Character element : array) System.out.printf("%s ", element);  
    System.out.println();  
}
```

These methods are identical except the data type part (in red). If the input is `Long[]` or `String[]`, shall we continue the overloading?



# A better design with generics

- If the operations performed by several overloaded methods are identical for each argument type, the overloaded methods can be more compactly coded using a generic method.

**泛型方法**

```
public static <T> void printArray(T[] array) {  
    for (T element : array) System.out.printf("%s ", element);  
    System.out.println();  
}
```

**实际类型参数**

传 `String[] array` 就调用 `String[] array`

Type-parameter section: one or more type parameters (类型参数) delimited by `<>`

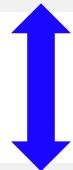
Each type parameter parameterizes the data types that can be used in the method  
(in the above example, T can be used anywhere a data type name is expected)

# Declaring generic methods

- ▶ Generic methods can be declared like any other normal methods.
- ▶ Type parameters can represent only reference types (not primitive types)

T只能用引用数据类型  
C出来

```
public static void printArray(Double[] array) {  
    for (Double element : array) System.out.printf("%s ", element);  
    System.out.println();  
}
```



No difference except the data type is parameterized

```
public static <T> void printArray(T[] array) {  
    for (T element : array) System.out.printf("%s ", element);  
    System.out.println();  
}
```



# Using generic methods

```
public static void main(String[] args) {  
    Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };  
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5 };  
    Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };  
    System.out.print("integerArray contains: ");  
    printArray(integerArray);  
    System.out.print("doubleArray contains: ");  
    printArray(doubleArray);  
    System.out.print("characterArray contains: ");  
    printArray(characterArray);  
}
```

Results are the same as before.  
But we only declare one  
printArray method!

```
integerArray contains: 1 2 3 4 5 6  
doubleArray contains: 1.1 2.2 3.3 4.4 5.5  
characterArray contains: H E L L O
```

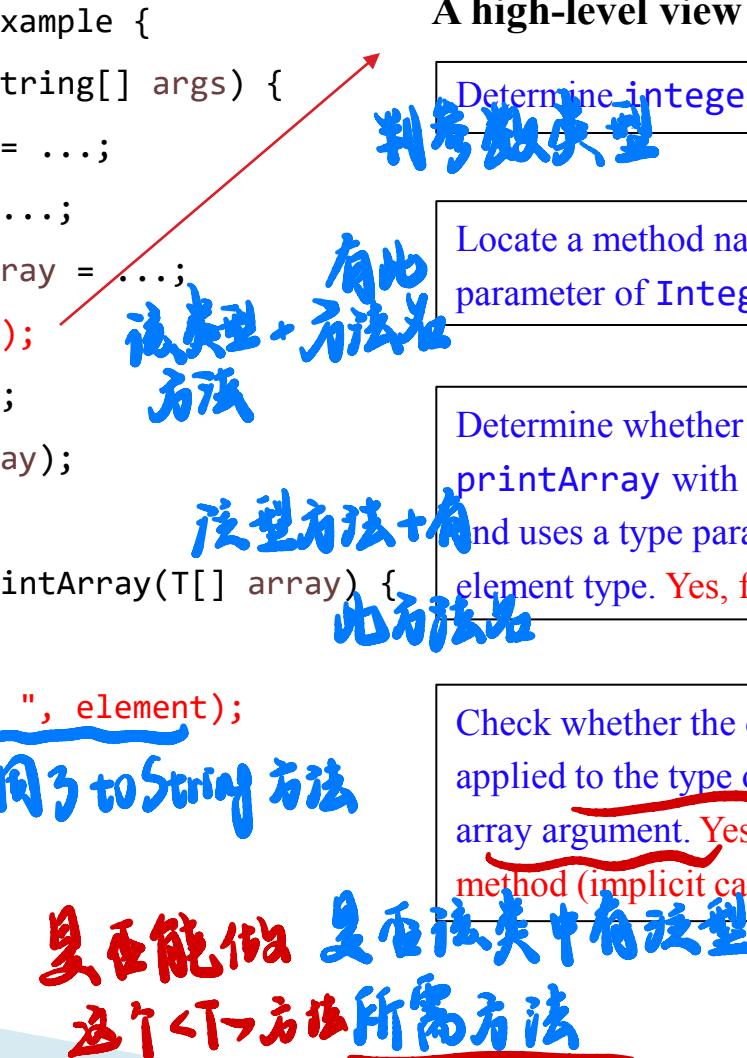
# How does compiler work here?

```

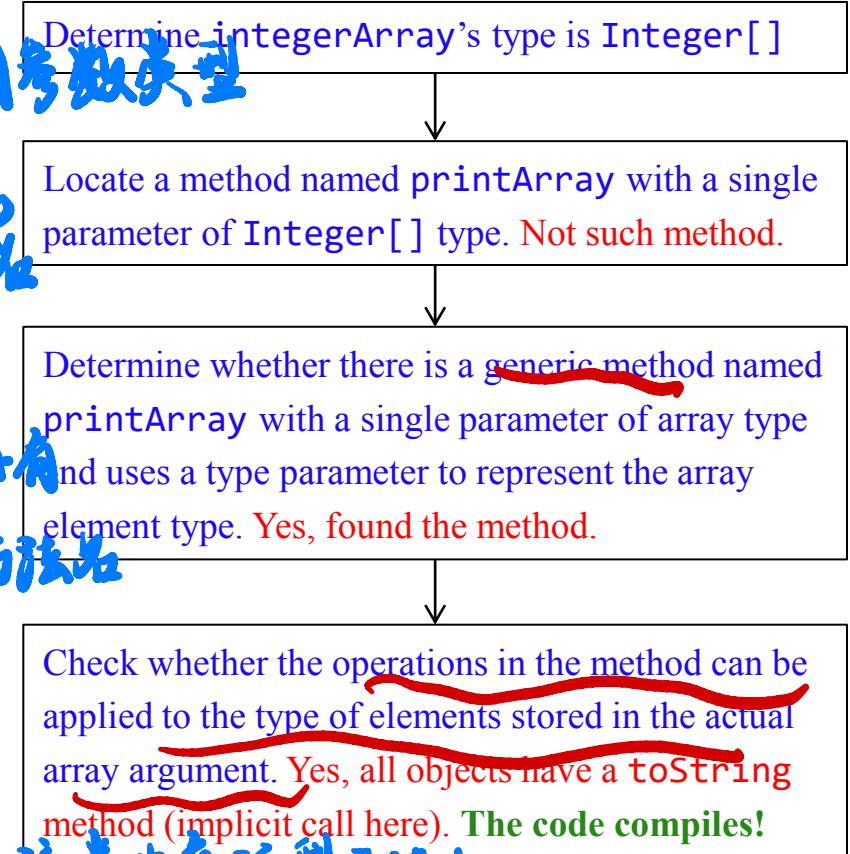
public class GenericMethodExample {
    public static void main(String[] args) {
        Integer[] integerArray = ...;
        Double[] doubleArray = ...;
        Character[] characterArray = ...;
        printArray(integerArray);
        printArray(doubleArray);
        printArray(characterArray);
    }

    public static <T> void printArray(T[] array) {
        for (T element : array)
            System.out.printf("%s ", element);
        System.out.println(); 调用了 toString 方法
    }
}

```



## A high-level view





# Under the hood: Erasure (消除)

和泛型相关的 placeholder 被擦除变成具体类

- When the compiler translates generic method `printArray` into Java bytecodes, it **removes the type-parameter section** and **replaces the type parameters with actual types**. This process is known as **erasure**.
- By default, all generic types are replaced with type **Object**
- The compiled version of `printArray` is shown below (we show source code instead of bytecodes)

upper bound 上界

```
public static void printArray(Object[] array) {  
    for (Object element : array) System.out.printf("%s ", element);  
    System.out.println();  
}
```



# Benefits of generic methods

In the earlier example, it seems that using generic methods is the same as using `Object` array as parameter of `printArray` (like the code below). Why using generics then?

```
public static void main(String[] args) {  
    Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };  
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5 };  
    Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };  
    System.out.print("integerArray contains: ");  
    printArray(integerArray);  
    System.out.print("doubleArray contains: ");  
    printArray(doubleArray);  
    System.out.print("characterArray contains: ");  
    printArray(characterArray);  
}  
public static void printArray(Object[] array) {  
    for (Object element : array) System.out.printf("%s ", element);  
    System.out.println();  
}
```

# Benefits of generic methods

```
public static Object simplyReturn(Object o) {  
    return o;  
}  
  
public static void main(String[] args) {  
    X String s = simplyReturn("hello");  
}
```



子类指向父类 *Object*

The compiler sees that the method return type is **Object**, assigning a reference of **Object** to a **String** variable is illegal, so a **compilation error** will occur.

Can we downcast to **String**?

# Benefits of generic methods

```
public static Object simplyReturn(Object o) {  
    return o;  
}  
  
public static void main(String[] args) {  
    X String s = (String)simplyReturn(100);  
}
```

Programmers need to perform explicit type cast: (String) simplyReturn("hello"),  
However, the code may generate ClassCastException if the cast fails at runtime.

强制解决

编译器不报错 (有强转但也会  
运行时类型转换错误 错)

# Benefits of generic methods

```
public static <T> T simplyReturn(T o) {  
    return o;  
}  
  
public static void main(String[] args) {  
     String s = simplyReturn("hello");  
}
```

String s = simplyReturn(100);  
compile报错

编译器推断类型后自动转换

With the generic method, the compiler will perform careful type checking and infer the return type is **String** when the actual argument's type is **String** and inserts type cast automatically (such cast will never throw ClassCastException, guaranteed to be safe).

Therefore, the code can be successfully compiled and is more type safe (类型安全).  
The benefits become obvious when the return type is also parameterized.



# Bounded Type Parameter

```
public static <T> T simplyReturn(T o) {  
    return o;  
}
```

任何引用类型

- ▶ In generic methods like the above, all reference types up to `Object` can be passed to the type parameter (we say `Object` is an **upper bound**).
- ▶ There are times when you want to **restrict the types that are allowed to be passed to a type parameter**, e.g., a method that operates on numbers might only want to accept instances of `Number` or its subclasses
- ▶ **Bounded type parameters (有界类型参数)** are useful in such cases.

# Bounded Type Parameter

- To declare a bounded type parameter, simply list the type parameter's name followed by the `extends` keyword and an upper bound (上边界)
  - Here, `extends` is used in a general sense to mean either “extends” as in classes or “implements” as in interfaces.

```
只能数类  
public static <T extends Number> T sum(T x, T y) {  
    return x.intValue() + y.intValue();  
}
```

中修改

T can be any type that extends the  
Number class, therefore we can  
invoke `intValue()` of such type T  
保证运行

# Bounded Type Parameter

- To declare a bounded type parameter, simply list the type parameter's name followed by the `extends` keyword and an upper bound (上边界)
  - Here, `extends` is used in a general sense to mean either “extends” as in classes or “implements” as in interfaces.

public static <T extends Comparable<T>> T maximum(T x, T y, T z) {  
 T max = x;  
 if (y.compareTo(max) > 0) max = y;  
 if (z.compareTo(max) > 0) max = z;  
 return max;  
}

*从这里开始理解 Comparable 接口*

T can be any type that implements the Comparable interface, therefore we can invoke compareTo of such type T



# Example

```
Maximum of 3, 4 and 5 is 5  
Maximum of 6.6, 8.8 and 7.7 is 8.8  
Maximum of pear, apple and orange is pear
```

```
public static void main(String[] args) {  
  
    System.out.printf("Maximum of %d, %d and %d is %d\n", 3, 4, 5,  
                      maximum(3, 4, 5));  
  
    System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f\n", 6.6,  
                      8.8, 7.7, maximum(6.6, 8.8, 7.7));  
  
    System.out.printf("Maximum of %s, %s and %s is %s\n", "pear",  
                      "apple", "orange", maximum("pear", "apple", "orange"));  
}
```

String 按字母序比较  
P开头

Integer, Double and String all implement the Comparable interface,  
so can be passed to the type parameter



# Compiler's view

沒有邊界擦除成Object

```
// Erasure: replacing the type parameter T with the upper bound Comparable  
public static Comparable maximum(Comparable x, Comparable y, Comparable z) {  
    Comparable max = x;  
    if (y.compareTo(max) > 0) max = y;  
    if (z.compareTo(max) > 0) max = z;  
    return max;  
}
```

有边界擦除成边界

推断 + 类型转换

When encountering method calls, infer the return type and insert explicit casts (the compiler guarantees that the cast will never throw ClassCastException):

maximum(3, 4, 5) → (Integer) maximum(3, 4, 5)

maximum(6.6, 8.8, 7.7) → (Double) maximum(6.6, 8.8, 7.7)

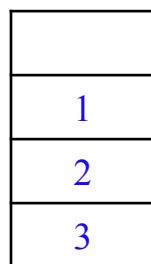
maximum("pear", "apple", "orange") → (String) maximum("pear", "apple", "orange")

# Objectives

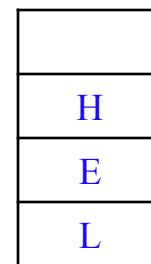
- ▶ Motivation of generic methods (泛型方法)
- ▶ Declare and use generic methods
- ▶ Declare and use generic classes (泛型类)

# Generic classes

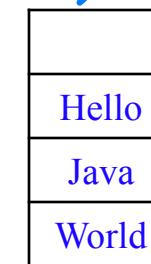
- ▶ The concept of many data structures, such as a stack, can be understood independently of the element type it manipulates.
- ▶ **Generic classes** provide a means for describing the concept of a stack (or any other classes) in a type independent manner.
- ▶ We can then instantiate type-specific objects of the generic classes. This makes software reusable (**program in general, not in specifics**).



A stack of Integer objects



A stack of Character objects



A stack of String objects

栈中类型  
一致



# We've seen generic classes before

ArrayList<E> is a **generic class**, where E is a placeholder (**占位符**) (**type parameter**) for the type of elements that you want the ArrayList to hold.

```
ArrayList<String> list;
```

Declares **list** as an ArrayList collection to store String objects

```
ArrayList<Integer> list;
```

Declares **list** as an ArrayList collection to store Integer objects

# Declaring a generic class

- ▶ A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a **type-parameter section**.
- ▶ The type-parameter section can have **one or more type parameters** separated by **commas**.
- ▶ Generic classes are also known as **parameterized classes**.
- ▶ In a generic class, type parameters can be used anywhere a type is expected (e.g., when declaring parameters, return types, defining variables ...)

可多种类型

参数化的类



# A generic Stack class

```
public class Stack<T> {  
    private ArrayList<T> elements; // use an ArrayList to implement the stack  
    public Stack() { this(10); }  
    public Stack(int capacity) {  
        int initCapacity = capacity > 0 ? capacity : 10;  
        elements = new ArrayList<T>(initCapacity);  
    }  
    public void push(T value) {  
        elements.add(value);  
    }  
    public T pop() {  
        if(elements.isEmpty())  
            throw new EmptyStackException();  
        return elements.remove(elements.size() - 1);  
    }  
}
```

泛型类 声明 可以<T,E,R>  
多个类型参数  
参数类 new 当前能用  
返回类型  
人为抛出异常  
弹出末尾元素



# Test the generic Stack class

```
public static void main(String[] args) {  
  
    Stack<Double> doubleStack = new Stack<Double>(5);  
    Stack<Integer> integerStack = new Stack<Integer>();  
  
    doubleStack.push(1.2);  
    Double value = doubleStack.pop();  
    System.out.println(value);  
  
    integerStack.push(1);  
    integerStack.push(2);  
  
    while(true) {  
        Integer i = integerStack.pop();  
        System.out.println(i);  
    }  
}
```

第三以已空栈

1.2  
2  
1  
Exception...

# Compiler's view

**Erasure (similar to generic methods):** Replacing all type parameters with Object or their bounds if the type parameters are bounded

The produced bytecodes contain only ordinary classes, interfaces, and methods, i.e.,  
**no generics at the bytecode level**

```
public class Stack {  
    private ArrayList<Object> elements;  
    public Stack() { this(10); }  
    public Stack(int capacity) {...  
        elements = new ArrayList<Object>(initCapacity);  
    }  
    public void push(Object value) { ... }  
    public Object pop() { ... }  
}
```



# Compiler's view

The compiler will insert type casts if necessary to preserve type safety

```
Stack<Double> doubleStack = new Stack<Double>(5);  
doubleStack.push(1.2);  
Double value = doubleStack.pop();
```



```
Stack doubleStack = new Stack(5);  
doubleStack.push(1.2);  
Double value = (Double) doubleStack.pop();
```

检查 强制转成  
(Double)  
包装类

# Let's test our understanding

- ▶ Q1: Will the compiler successfully compile the following code?

```
String s = "hello world";
```

```
Object obj = s;
```

父引用指向子对象



class文件中擦除完没有泛型概念了

- ▶ It is **safe** to assign `s` (of type `String`) to `obj` (of type `Object`) because an instance of a subclass (subtype) is also an instance of a superclass (supertype).
- ▶ “**Safe**” means any operations that can be done via the reference `obj` are also allowed to be done via the reference `s`

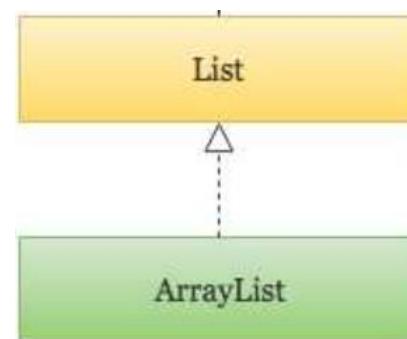
# A more difficult question

- ▶ **Q2:** Will the compiler successfully compile the following code?

```
ArrayList<String> ls = new ArrayList<String>();  
List<String> ls2 = ls;
```



- ▶ It is safe to assign `ls` to `ls2` because an `ArrayList` of `String` is also a `List` of `String`.
- ▶ Any operations that can be done via the reference `ls2` can also be done via the reference `ls`



# The “hardest” question about generics



- ▶ Q3: Will the compiler successfully compile the following code?

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls;
```

泛型间不交叉有继承关系



- ▶ This boils down to the question: *is a List of String a List of Object?* (Most people will instinctively answer “yes”...)
- ▶ What if we ask the safety question: *is it true that any operations that can be done via the reference Lo can also be done via Ls?*

父类能做，子类应该也能做

# Let's do some analysis

- As a reference of type `List<Object>`, `lo` can be used for the following operation:

```
lo.add(new Double());
```

- However, we cannot perform the same operation via the reference `ls` because it is of type `List<String>`

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls; // type mismatch
```



# Further analysis from compiler's view

- If the compiler allows assigning `ls` to `lo`, then the code

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls; 指向同一地址  
lo.add(new Double(0.0));  
String s = ls.get(0);
```

- will be compiled into the following form:

```
List ls = new ArrayList();  
List lo = ls;  
lo.add(new Double(0.0));  
String s = (String) ls.get(0);
```



**ClassCastException**

Generic classes are designed to provide type safety, such exceptions are awkward



# General Rule

- ▶ If Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is not the case that  $G<\text{Foo}>$  is a subtype of  $G<\text{Bar}>$ .
- ▶  $\underline{G<?>}$  is a superclass of  $\underline{G<T>}$  for any T
- ▶ This is probably the hardest thing one needs to learn about generics, because it goes against our intuitions...