Chapter 13: Generics Part I

Yepang LIU

liuyp1@sustech.edu.cn

(C) 2010 Pearson Education, Inc. All rights reserved.

Objectives

- Motivation of generic methods
- Declare and use generic methods
- Declare and use generic classes

(C) 2010 Pearson Education, Inc. All rights reserved.

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

```
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);
}
```

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?

(C) 2010 Pearson Education, Inc. All rights reserved.

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.

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

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)

```
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

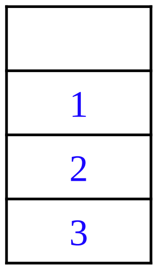
```
public static void main(String[] args) {
    Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
    System.out.print("integerArray contains: ");
    printArray(integerArray);
    System.out.print("doubleArray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

Same as before!!!

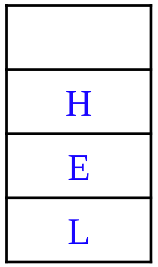
```
integerArray contains: 1 2 3 4 5 6
doubleArray contains: 1.1 2.2 3.3 4.4 5.5
characterArray contains: H E L L O
```

Generic classes

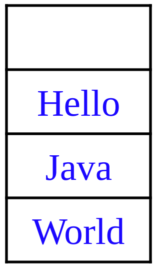
- The concept of many data structures, such as a stack, can be understood independently of the element type it manipulates.
- **Generic classes** provide a means for describing the concept of a stack (or any other classes) in a type independent manner.
- We can then instantiate type-specific objects 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

We’ve seen generic classes before

`ArrayList<E>` is a **generic class**, where `E` is a placeholder (**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 …)

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("Stack is empty, cannot pop");
        return elements.remove(elements.size() - 1);
    }
}
```

Note: `EmptyStackException` is a self-defined exception type

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);
    }
}
```

```
1.2

2

1

Exception...
```