Java EE基础与应用

翁秀木

Java 5 绰号（code name）: Tiger

第五回 Java 新特性（5.0之后)

- Generics 泛型
- Enhanced for Loop 增强版的for循环
- Auto boxing/unboxing 自动装箱和拆箱
- import static 导入静态成员
- Annotation 标注
- Variable arguments 可变的参数

学习目标

- **了解**Java新特性（Java 5.0之后），**能基本读懂**新特性代码，包括泛型、可变参数、增强版的for循环、自动装箱和拆箱、导入静态类和标注。

Enhanced for Loop 增强版的for循环

- 看代码示例

Variable Argument 可变参数 (varargs) *

- 在5.0之前，若要方法能接收任意数量的参数，常用数组作参数。

- 在5.0之前，若要方法能支持1个或3个同类型数量的参数，常用方法重载。

Auto boxing and unboxing自动装箱和拆箱*

- 看代码示例

import static *

- 导入类的static member。
- 看代码示例

Annotations 标注

- **给编译器提供信息（指令）**：帮助编译器检查错误(error)或屏蔽警告（warnings)
- **编译时或部署时的处理**：软件工具可处理注释信息，以产生代码、XML文件等
- **运行时的处理**：某些注释可在运行时（runtime）被检查

@SuppressWarnings

@SuppressWarnings("unchecked")

"unchecked" 意思是编译器不能在编译时执行类型检查，以保证类型安全。这个标注会屏蔽unchecked警告。这个标注可以设置在类、属性（field）、方法（method）、构造器、局部变量、参数上。

@SuppressWarnings

记住：若需要屏蔽警告，则应尽可能细粒度，例如，若要屏蔽某个方法的警告，就在这个方法上，而不要在该方法所属类上@SuppressWarnings。

往往警告说明代码有问题，如代码质量不高，有没用到的变量，存在着运行时类型危险等。不要轻易屏蔽警告，除非你真的十分确定这个警告是可以忽略的。

若你十分确定这个警告是没危险的，就在最细粒度上用@SuppressWarning屏蔽它，最好不要无视它，因为这容易导致在后续编码中，不小心漏掉有危险的警告。

Generics 泛型

Generics =泛型，Generic type = 泛型类型（包括泛型类和泛型接口）， Generic method = 泛型方法。

关于翻译，Generics =泛，Generic type = 泛型（包括泛类和泛接口）， Generic method = 泛方法，或许更贴近英文原意。

Generics 泛型 *

Java 5.0之前

```
List aList = new ArrayList();
aList.add("test");
Integer i = (Integer)aList.get(0); //运行时（Run time）错误
```

Java 5.0之后

```
List<String> aList = new ArrayList<String>();
aList.add("test");
Integer i = (Integer)aList.get(0); // 编译时（complile time)错误
```

记住：泛型作用1是提供了编译时的更安全的类型检查。

Generics 泛型 *

Java5.0之前

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Java 5.0之后

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast
```

记住：泛型作用2是消除了不必要的类型强制转换。

Generics 泛型示例 - 定义接口或类

Java 5.0之前

```
public interface List extends Collection {
        boolean add(Object o);
        Object get(int index);
        boolean removeAll(Collection c);
    }
```

Java 5.0之后

```
public interface List<E> extends Collection<E> {
        boolean add(E e);
        E get(int index);
        boolean removeAll(Collection<?> c);
}
```

Generics 泛型示例 - 使用泛型的类或接口

Java 5.0之前

```
List myList = new LinkedList(); // 1
myList.add(new Integer(0)); // 2
Integer x = (Integer) myList.iterator().next(); // 3
```

Java 5.0之后

```
List<Integer> myList = new LinkedList<Integer>(); // 1'
myList.add(new Integer(0)); // 2'
Integer x = myList.iterator().next(); // 3'
```

Generics泛型示例

```
public class Box {    // Java 5.0之前
    private Object object;
    public void set(Object object) { this.object = object;}
    public Object get() { return object; }
}

public class Box<T> {          // Java 5.0之后
     // T stands for "Type"
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Generics的类型参数（type parameter），也叫类型变量（type variable）

- 类型参数用尖括号包着，可以是任何非原始类型，可以是类、接口、数组等。
- 类型参数的命名规范：通常是大写单字母。

E - Element (常被用于Collections 框架)

K - Key ； N - Number； T - Type ； V - Value；

S,U,V 等. - 可用于第2, 第3, 第4个类型参数

Generics 泛型示例

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();

System.out.println(l1.getClass() == l2.getClass());
```

记住：java程序运行时（runtime），所有泛型类的实例都属于同一个类型（类或接口），不管实际的类型参数（type parameter)是否相同。因为编译器编译时会将泛型信息（即尖括号及其内容消除掉），如上例ArrayList<String>和ArrayList<Integer>都会变成ArrayList。

Generics泛型与子类型 *

```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

```
lo.add(new Object()); // 3
String s = ls.get(0); // 4
```

Generics泛型与子类型

记住：Collection<String>不是Collection<Object>的子类型，而是Collection<?>的子类型

List<String>不是List<Object>的子类型，而是List<?>的子类型

Generics泛型与子类型

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { // compile-time err
    // ...
}
ArrayList<String> cs = new ArrayList<String>();
if (cs instanceof ArrayList<String>) { // compile-time err
    // ...
}
```

记住：不能把对泛型的具体调用用于instanceof

Generics泛型与子类型

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<?>) { // no err

}
```

```
ArrayList<String> cs = new ArrayList<String>();
if (cs instanceof ArrayList<?>) { // no err
    // ...
}
```

Generics泛型 - 通配符 wildcards

java 5之前的用法，参数c能接受所有对象的集合

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
}}
```

若改成如下，则参数c不能接受Object子类对象的集合，如Collection<String>对象。

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Generics泛型 - 通配符 wildcards

若改成如下，则参数c能接受Object及其子类对象的集合

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

记住：wildcard表示未知的、不确定的对象类型

Generics泛型 - 通配符 wildcards

```
Collection<?> c = new ArrayList<String>();
c.add(new Object());
c.add(new String());
```

上面红字语句编译时错误，因为不知道c引用哪一种Collection。在运行时，ArrayList对象才被创建，并把该对象的地址赋给c。

记住：不能把一个**非**空的对象加入带通配符的集合，因为当集合的类型参数是通配符时，表示**不知道集合中存放的是什么类型**。

被限定的通配符 Bounded wildcards *

List <? extends Number>，?表示一个未知的类型，该类型是Number本身或者是其子类。

例1：

```
Number read(List<? extends Number> aList) {
  return aList.get(0);
}
```

例2：

```
void write(List<? extends Number> aList) {  // 1
  aList.add(new Integer(1));  // 2
}
```

// 例2第2句编译时错误，因为不确定aList接受到的是哪种Number的子类型的对象。在运行时可能aList里能放的元素不是Integer。

被限定的通配符 Bounded wildcards *

List <? super String>，?表示一个未知的类型，该类型是String本身或者是其超类。

例1：

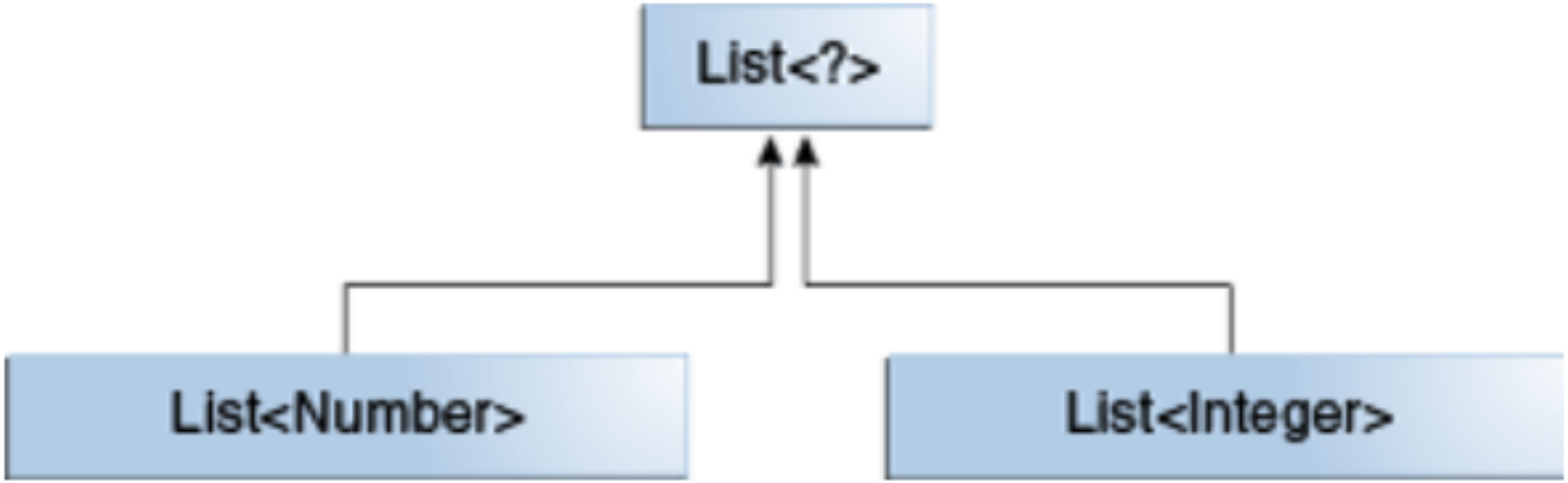
```
Object read(List<? super String> aList) {
return aList.get(0);
   }
```

例2：

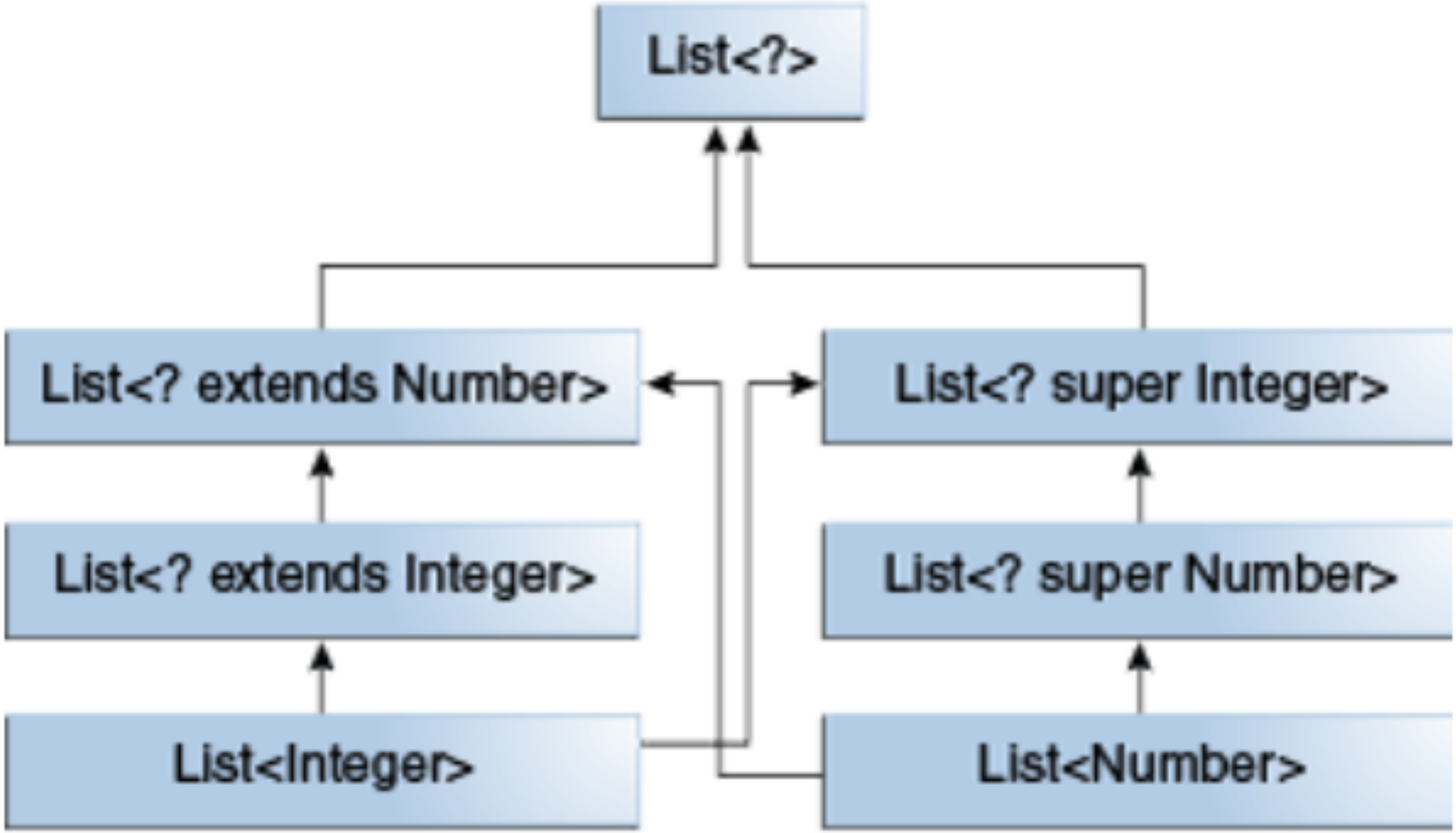
```
CharSequence read(List<? super String> aList) { // 1
return aList.get(0); // 2
   }
```

// 例2第1句编译时错误，因为不知道aList里头放的是哪一种String的超类型的对象。在运行时才知道。

Generics泛型 - 通配符 wildcards



Generics泛型 - 通配符 wildcards



Generic Methods 泛型方法（泛方法）：

```
void arrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o); // compile time error
    }
}
```

```
<T> void arrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o); // correct
    }
}
```

Generic Methods 泛型方法：

- 泛型方法，可理解为给方法声明了类型参数，类型参数用尖括号包着，设置在方法的返回类型之前，并必须紧挨着返回类型。

- 可用于静态static或非静态方法，或者构造器。

- 泛型方法中的类型参数只能用在该方法内。

Generic Methods 泛型方法：

- 使用泛型方法目的是提供方法的返回类型、多于1个的参数类型之间的依赖性。

定义： public <T> T playGenericMethod(T param1) {
    return param1;
}

调用： <String>playGenericMethod("FORZAMILAN");

定义： public <T> void playGenericMethod(T param1, T param2) {}

调用： <String>playGenericMethod("FORZA", "MILAN");

Generic Methods 泛型方法与通配符：

```
interface Collection<E> {
    public boolean containsAll(Collection<?> c);
    public boolean addAll(Collection<? extends E> c);
}
```

```
interface Collection<E> {
    public <T> boolean containsAll(Collection<T> c);
    public <T extends E> boolean addAll(Collection<T> c);
    // hey, type variables can have bounds too!
}
```

Generic Methods 泛型方法与通配符：

记住：若只是为了支持多态性，而不是为了提供方法返回类型、参数类型之间的依赖性，则最好用通配符，而且，通配符的另一好处是其作用范围不仅限于该方法。

```
public <T> void play (List<T> param1) {
}

// 应该用wildcard，而非泛型方法generic method
public  void play (List<?> param1) {
}
```

Type Inference类型推断：

对于要调用泛型、包括泛型方法的代码，在某些情境下，编译器会自动推断其类型参数，而无需明确指定类型参数，如此，可使代码更简化，可读性更好。

Type Inference类型推断：

```
Map<String, List<String>> myMap =
                    new HashMap<String, List<String>>();
```

```
Map<String, List<String>> myMap =  new HashMap<>();
```

编译器会在编译时推断出尖括号中的类型参数，注意这个特性（称为diamond）在java7及其之后才出现。

Type Inference类型推断：

```
class Test {
static <T> T pick(T a1, T a2) { return a2; }  // 方法

public static void main(String[] args) {
// 方法调用有两种方式
Serializable s1 = Test.<Serializable>pick("d", new Integer(1} ))
Serializable s2 = Test.pick ("d", new Integer(1} ))
}
```

第2种方式无需指定泛型方法的类型参数，编译器会推断T为Serializable

Type Erasure**类型消除：**

对于使用了泛型，包括泛型方法的代码，编译器会先把这些代码翻译（translate）成没有泛型的样子（即消除泛型信息），然后再编译成.class文件（即字节码）。

Type Erasure类型消除：

public class Node<T> { // 源码

```
private T data;
private Node<T> next;
public Node(T data, Node<T>
next) {
 this.data = data;
 this.next = next;
}
public T getData() {
return data;
 }
// ...
}
```

```
public class Node { // 类型消除后
private Object data;
 private Node next;
public Node(Object data, Node
next) { this.data = data;
this.next = next; }
public Object getData() {
return data;
}
// ...
 }
```

Type Erasure类型消除：

```
public class Node<T extends Comparable<T>> {  // 源码
private T data;
private Node<T> next;
public Node(T data, Node<T> next) {
this.data = data;
this.next = next;
 }
public T getData() {
 return data;
}
// ...
 }
```

```
public class Node {  // 类型消除后
private Comparable data;
 private Node next;
public Node(Comparable data, Node next) {
this.data = data;
this.next = next;
}
public Comparable getData() {
 return data;
}
// ... }
```

泛型方法的Type Erasure类型消除

```
public static <T> int count(T[] anArray, T elem) { //
源码
int cnt = 0;
for (T e : anArray)
    if (e.equals(elem))
        ++cnt;
return cnt;
}
```

```
// 类型消除后
public static int count(Object[] anArray, Object elem)
{
 int cnt = 0;
for (Object e : anArray)
     if (e.equals(elem))
         ++cnt;
return cnt;
}
```

泛型方法的Type Erasure类型消除

假设存在以下类：

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }
```

为画出不同形状，创建泛型方法如下：

```
public static <T extends Shape> void draw(T shape1, T sharep2) { /* ... */ }
```

```
// 类型消除后
public static void draw(Shape shape, Shape shape) { /* ... */ }
```

泛型代码与原始类型 (raw type)代码的兼容

```
// 泛型 to 原始类型
List list1 = new ArrayList<String>();

// 原始类型 to 泛型
List<String> list2 = new ArrayList();
```

未完待续，谢谢！