语法

面向对象程序设计

第 8 讲 Java OOP- 泛型 & 枚举 & 注解

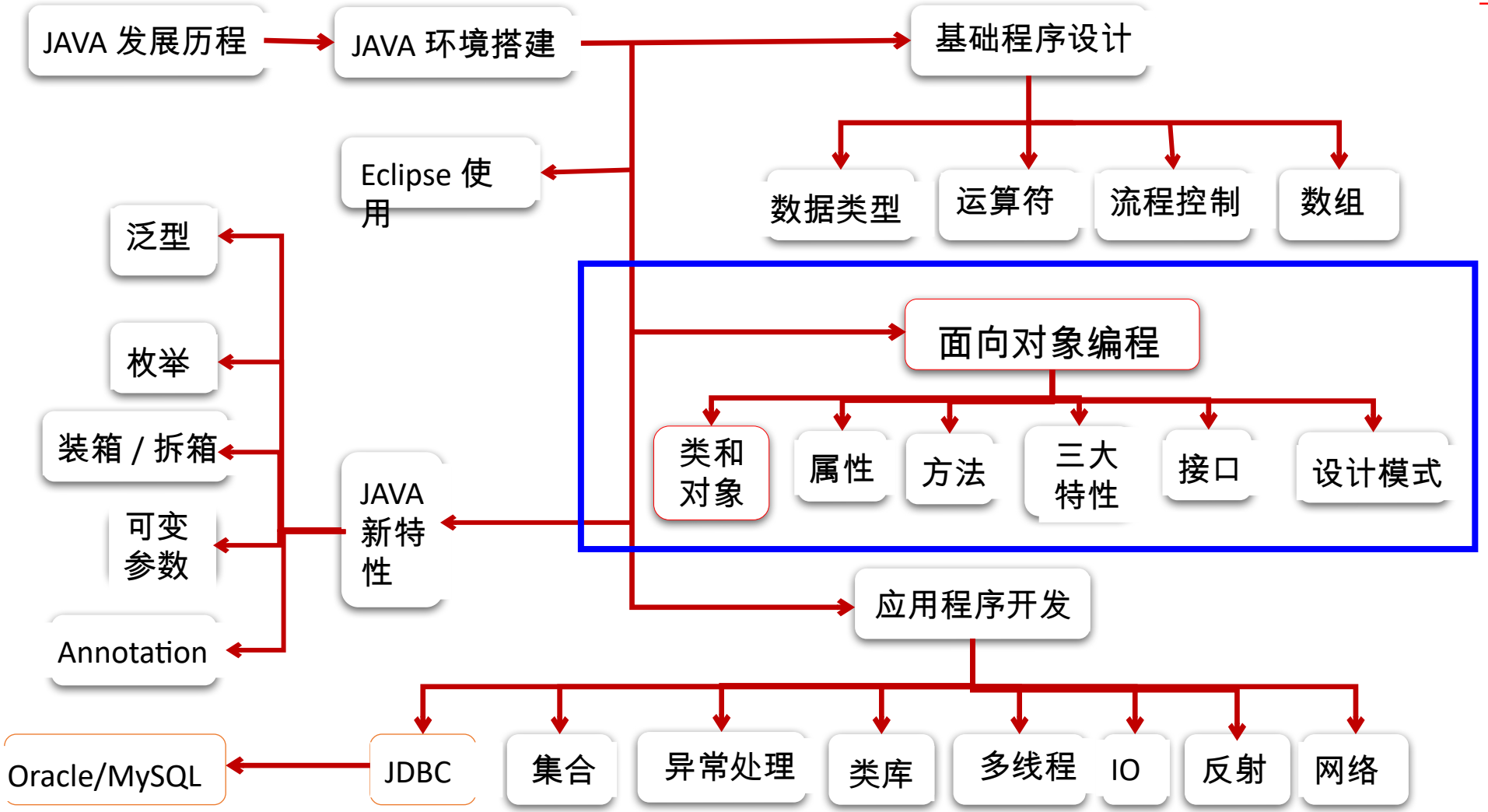
刘进

2230652597@qq.com

OOP 教辅 2022 秋季 QQ 群：

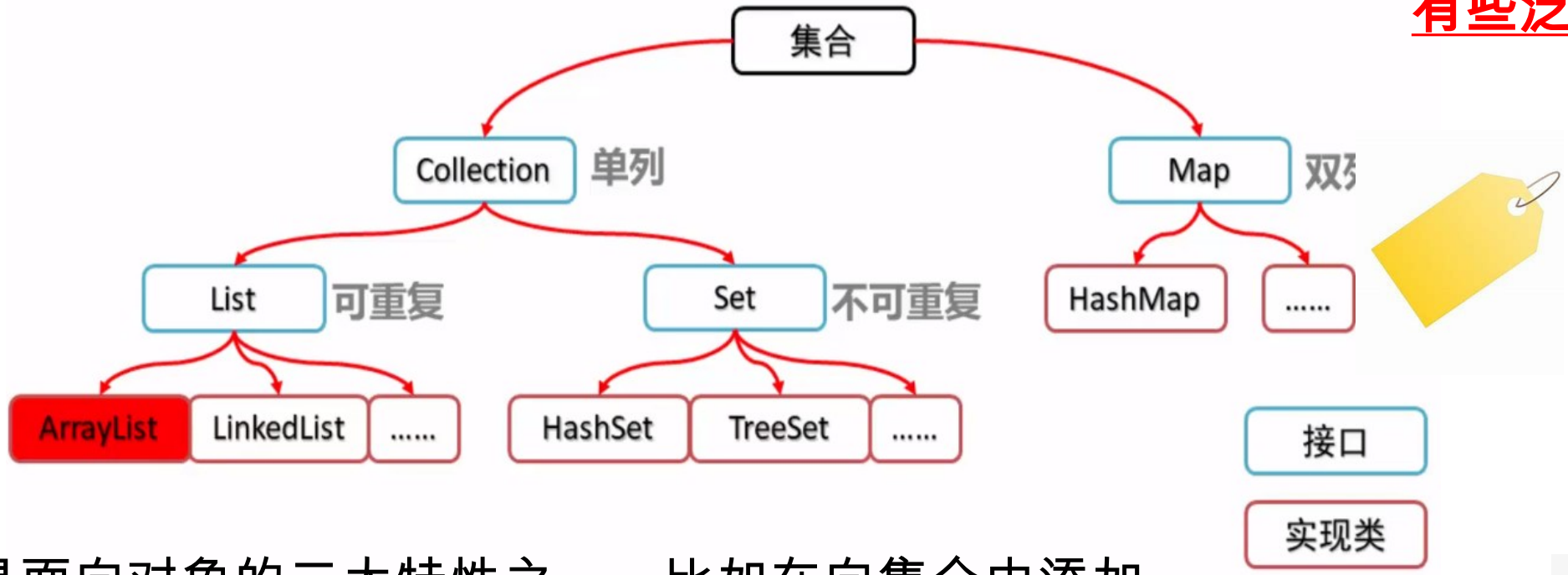
Java 基础知识图解

主要知识点

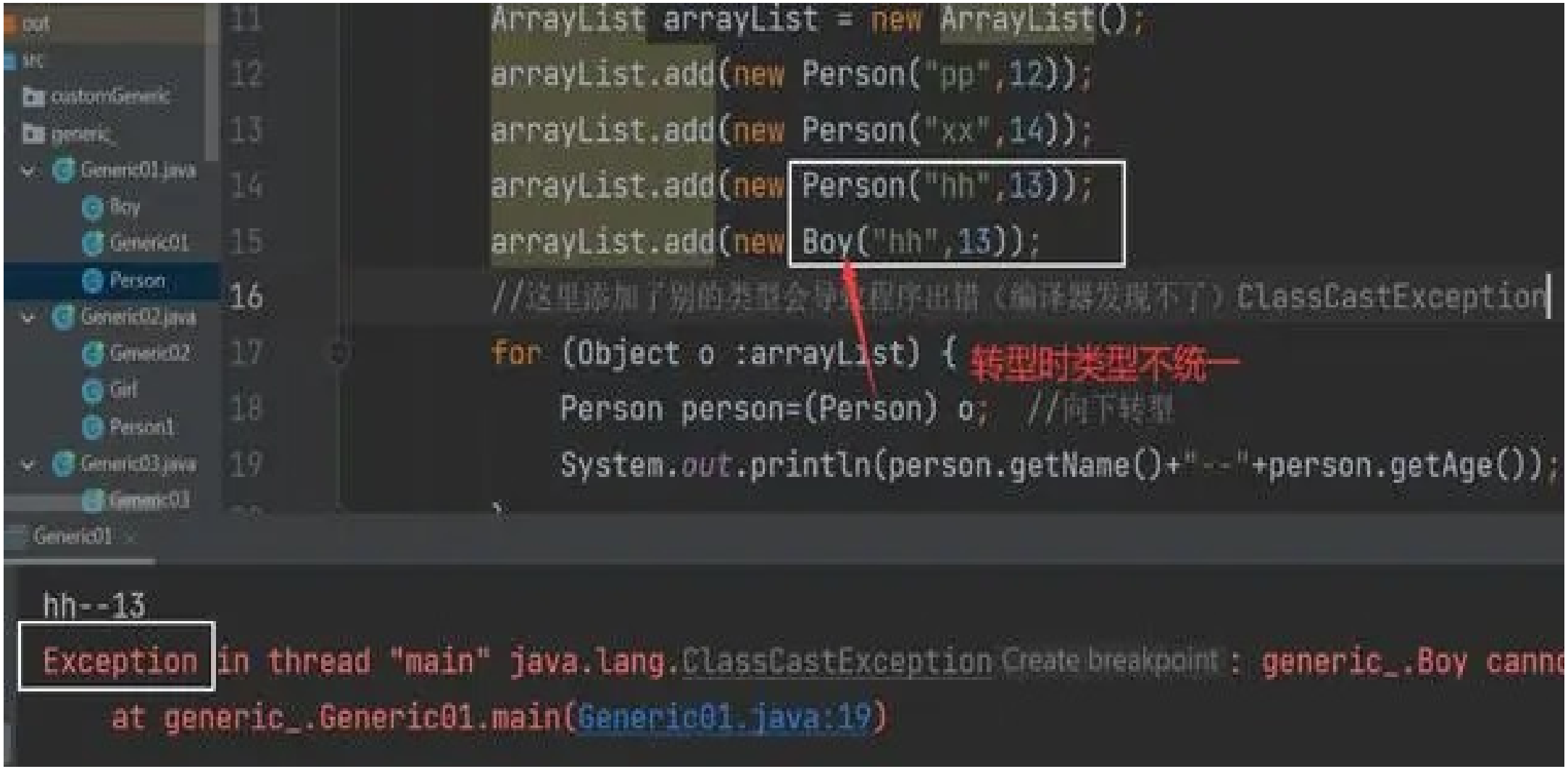


一、为什么要有泛型

有些泛型编程的意思

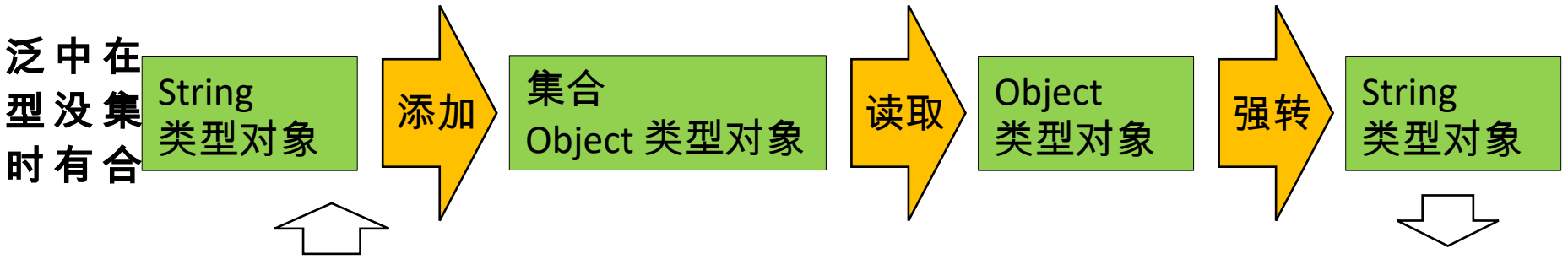


继承是面向对象的三大特性之一，比如在向集合中添加元素的过程中 add() 方法里填入的是 Object 类，而 Object 又是所有类的父类，这就产生了一个问题——添加的类型无法做到统一 由此就可能产生在遍历集合取出元素时类型不统一而报错问题。


out
src
customGeneric
generic_
Generic01.java
Boy
Generic01
Person
Generic02.java
Generic02
Girl
Person1
Generic03.java
Generic03
Generic01
ArrayList arrayList = new ArrayList();
arrayList.add(new Person("pp",12));
arrayList.add(new Person("xx",14));
arrayList.add(new Person("hh",13));
arrayList.add(new Boy("hh",13));
//这里添加了别的类型会导致程序出错（编译器发现不了）ClassCastException
for (Object o :arrayList) {
转型时类型不统一
Person person=(Person) o; //向下转型
System.out.println(person.getName()+"--"+person.getAge());
hh--13
Exception in thread "main" java.lang.ClassCastException Create breakpoint : generic_.Boy cannc
at generic_.Generic01.main(Generic01.java:19)

一、为什么要有泛型(Generic)?

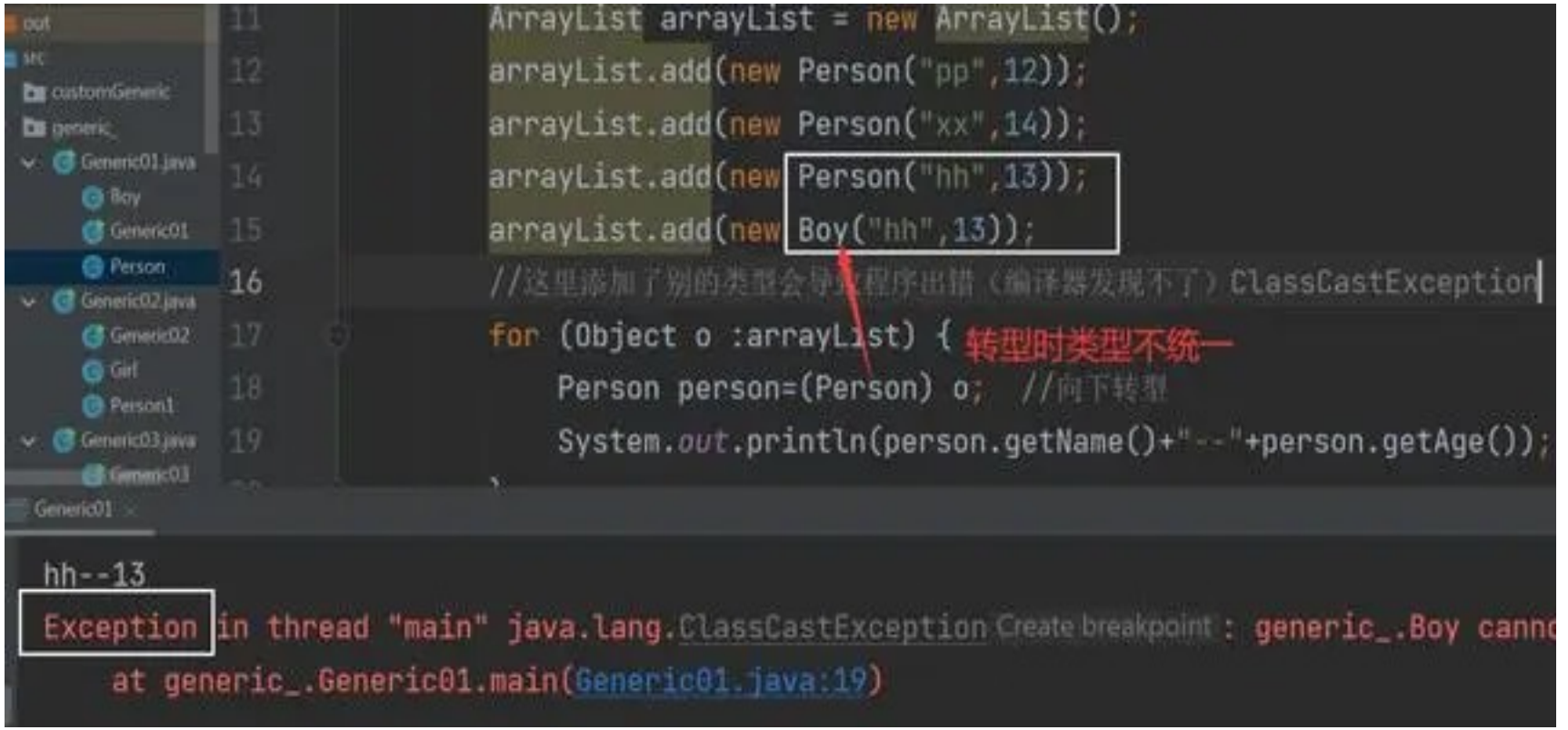
1. 解决元素存储的安全性问题
2. 解决获取数据元素时，需要类型强转的问题



任何类型都可以添加到集合中：**类型不安全**

读取出来的对象需要强转：**繁琐**
可能有 ClassCastException

数组定义的类型只能是相同数据类型，就是单一的数据类型，集合 (Object obj) 类型，各种类型都能存入到里边，不够严格，**引入泛型后表明是单一的数据类型**



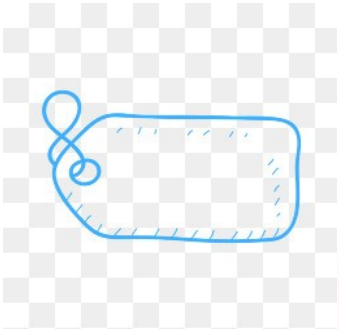
传统的方式不能对加入到集合 ArrayList 中的数据类型进行约束 (不安全)

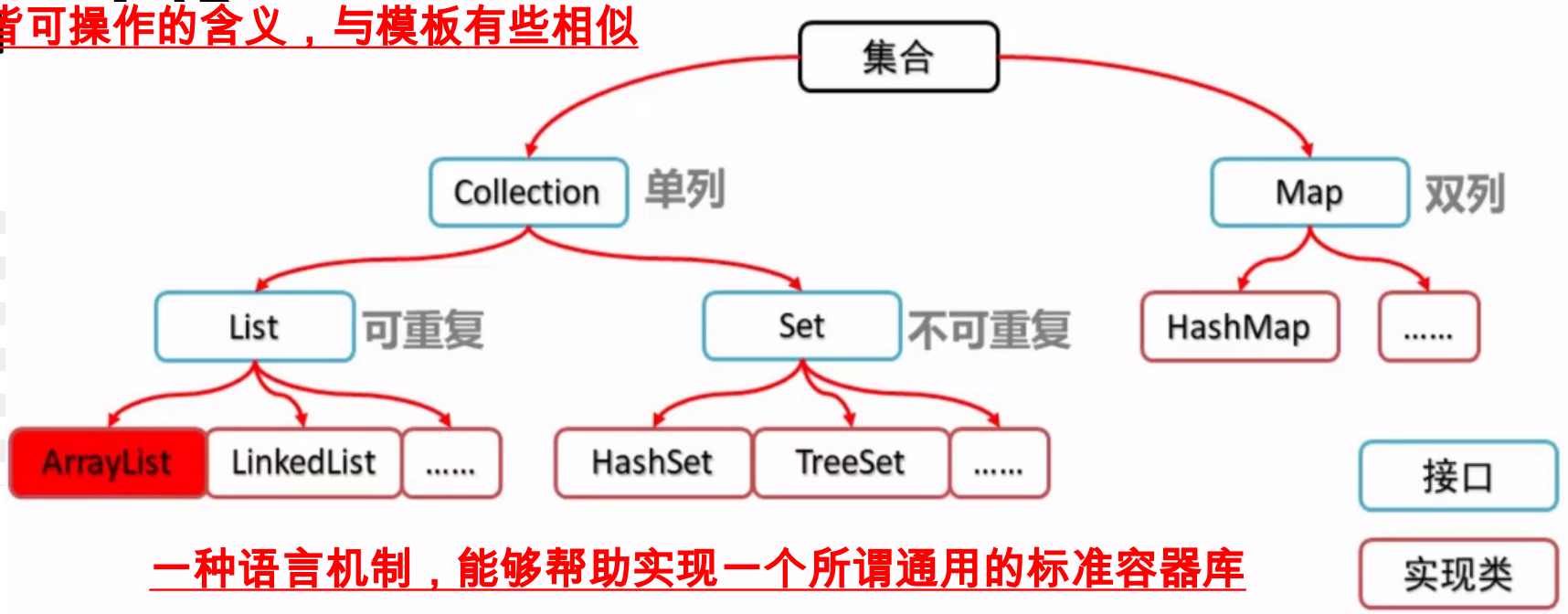
遍历的时候，需要进行类型转换，如果集合中的数据量较大，对效率有

影响 这就极大地降低了程序的健壮性，因此设计者针对此问题引入了泛

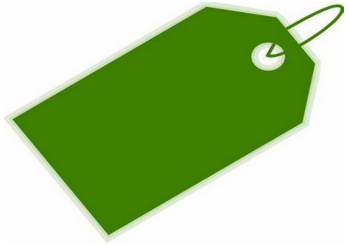
一、为什么要有泛型

泛型即是指具有在多种数据类型上皆可操作的含义，与模板有些相似





一种语言机制，能够帮助实现一个所谓通用的标准容器库

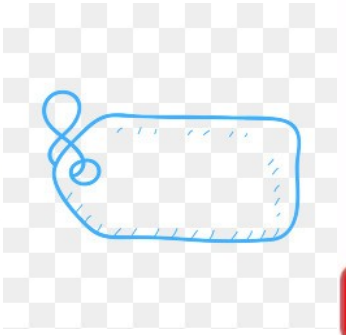


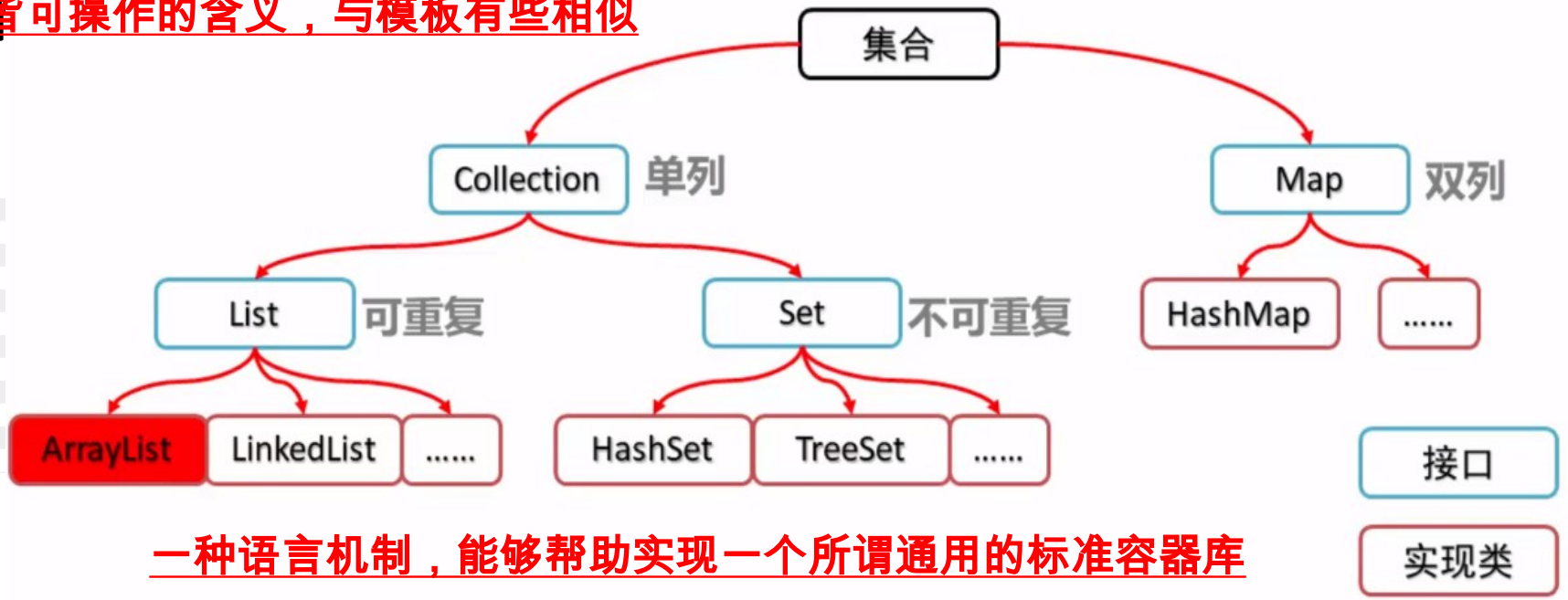
STL 以迭代器 (Iterators) 和容器 (Containers) 为基础，是一种泛型算法 (Generic Algorithms) 库，容器的存在使这些算法有东西可以操作。

STL 包含各种泛型算法 (algorithms) 、泛型迭代器 (iterators) 、泛型容器 (containers) 以及函数对象 (function objects) 。

一、为什么要有泛型(Generic)

泛型即是指具有在多种数据类型上皆可操作的含义，与模板有些相似





一种语言机制，能够帮助实现一个所谓通用的标准容器库

把一个集合中的内容限制为一个特定的数据类型，这就是 generics 背后的核心思想。

1、泛型的第一个好处是编译时的严格类型检查。

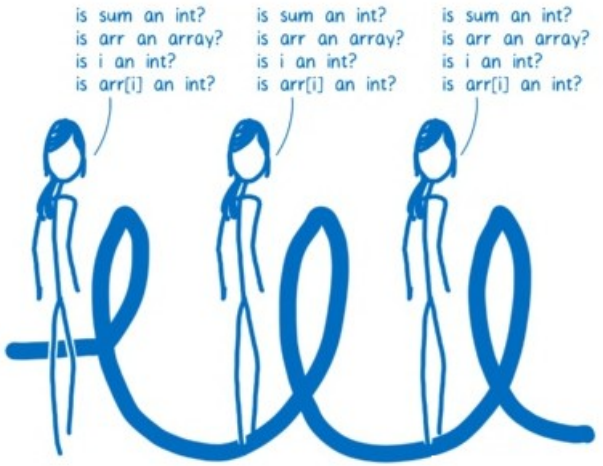
这是集合框架最重要的特点。

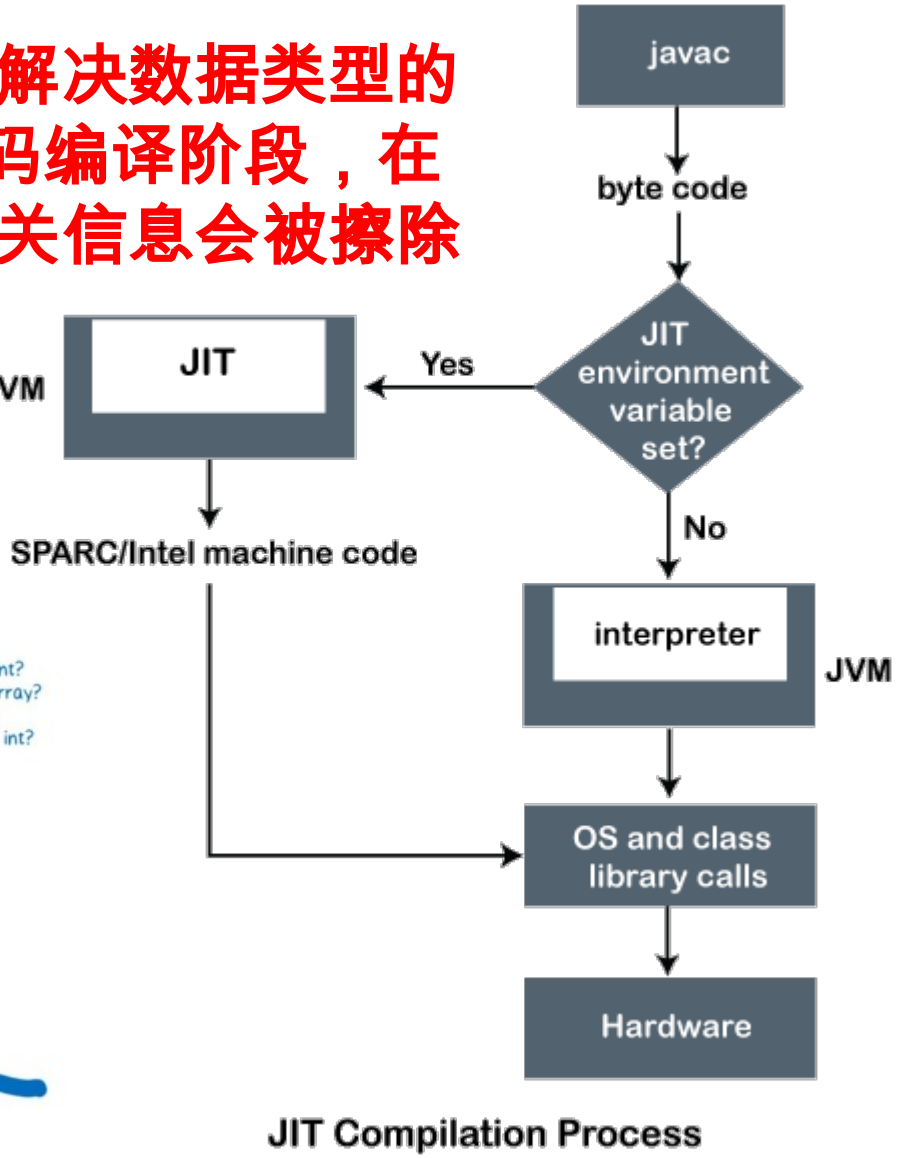
2、泛型消除了绝大多数的类型转换。

如果没有泛型，使用集合框架时，不得不进行类型转换。

一、为什么要有泛型(Generic)?

泛型，JDK1.5 新加入的，解决数据类型的安全性问题，只存在于代码编译阶段，在进入 JVM 之前，与泛型相关信息会被擦除





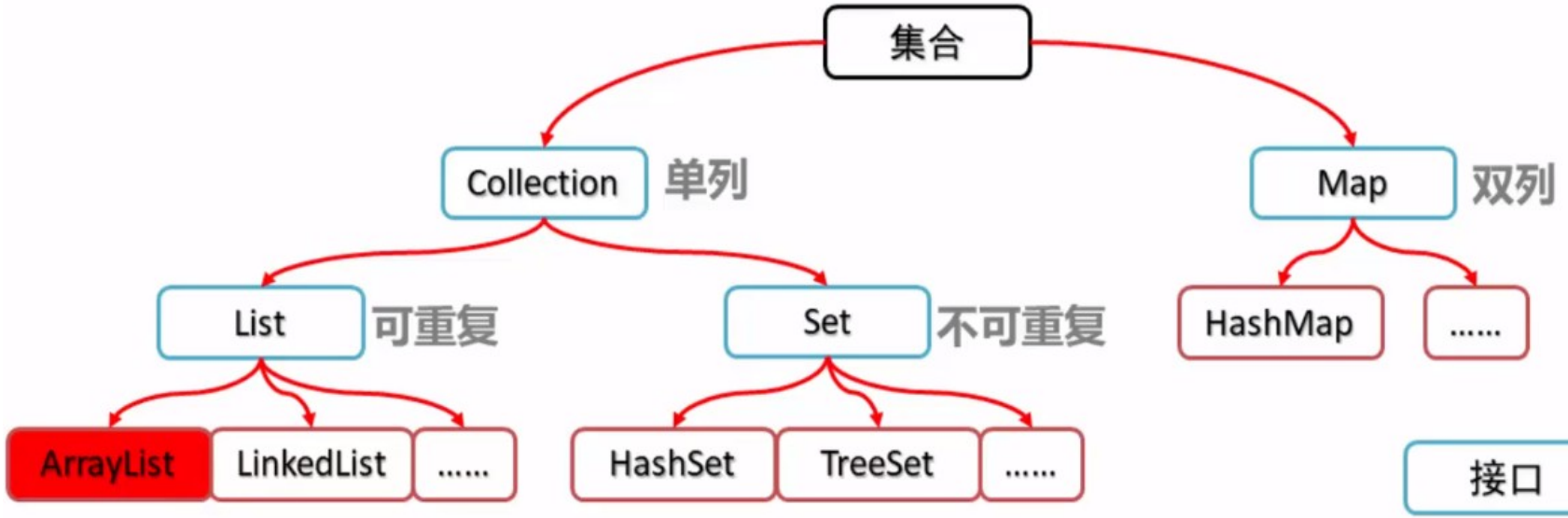
编译器检查不出这种错误，只有在运行期才能检查出来，此时就会出现

ClassCastException。所以用 Object 来实现泛型的功能就要求时刻做好类型转换，很容易出现问题。

```
1  public class A {
2      private Object b;
3      public void setB(Object b) {
4          this.b = b;
5      }
6      public Object getB() {
7          return b;
8      }
9  }
10 ------------------------------------
11 A a=new A();
12 a.setB(1);
13 int b=(int)a.getB();//需要做类型强转
14 String c=(String)a.getB();//运行时，ClassCastException
```

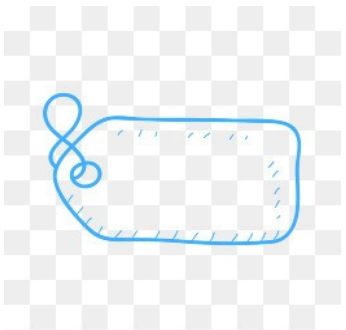
有没有办法将这些检查放在编译期做呢？泛型在编译期进行类型检查，问题就容易多了。

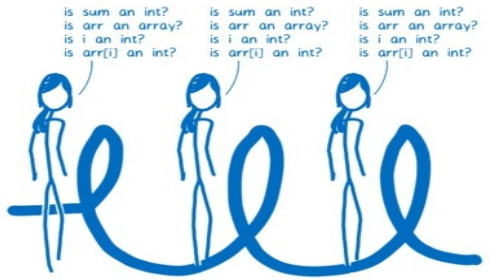
一、为什么要有泛型

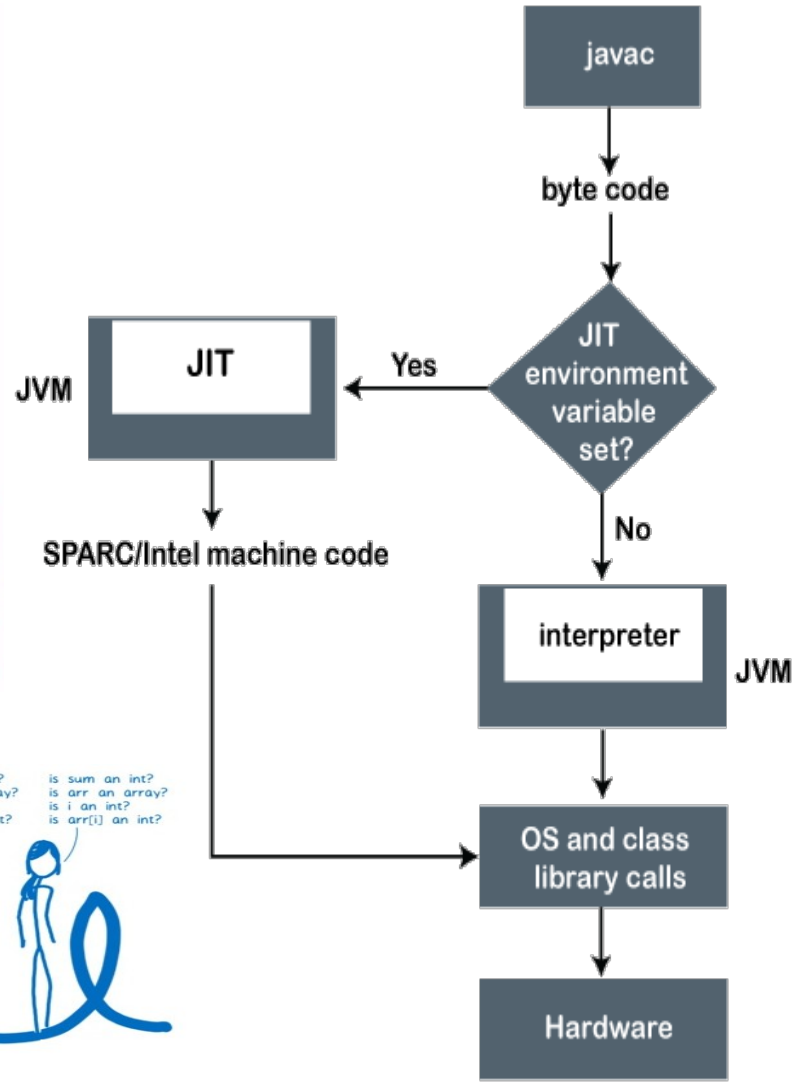


接口

实现类

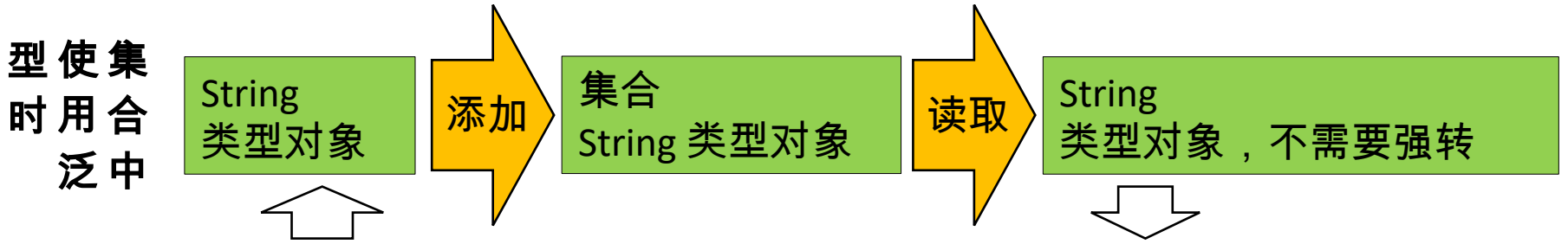






JIT Compilation Process

一、为什么要有泛型 Generic?



只有指定类型才可以添加到集合中：**类型安全**　　读取出来的对象不需要强转：**便捷**

泛型，JDK1.5 新加入的，解决数据类型的安全性问题，其**主要原理是在类声明时通过一个标识表示类中某个属性的类型或者是某个方法的返回值及参数类型。这样在类声明或实例化时只要指定好需要的具体的类型即可**。

Java 泛型可以保证如果程序在编译时没有发出警告，运行时就不会产生 ClassCastException 异常。同时，代码更加简洁、健壮。

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

```
1  public class GenericTest {
2
3      public static void main(String[] args) {
4          /*
5          List list = new ArrayList();
6          list.add("qqyumidi");
7          list.add("corn");
8          list.add(100);
9          */
10
11         List<String> list = new ArrayList<String>();
12         list.add("qqyumidi");
13         list.add("corn");
14         //list.add(100);    // 1  提示编译错误
15
16         for (int i = 0; i < list.size(); i++) {
17             String name = list.get(i); // 2
18             System.out.println("name:" + name);
19         }
20     }
21 }
```

操作的数据类型被指定为一个参数（type parameter）这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法

“参数化类型”

采用泛型写法后，在 //1 处想加入一个 Integer 类型的对象时会出现编译错误，通过 List<String> ，直接限定了 list 集合中只能含有 String 类型的元素，从而在 //2 处无须进行强制类型转换，因为此时，集合能够记住元素的类型信息，编译器已经能够确认它是 String 类型了。

泛型的出现减少了很多强转的操作，同时避免了很多运行时的错误，在编译期完成检查

```java
public class A<T> {
    private T b;
    public void setB(T b) {
        this.b = b;
    }
    public T getB() {
        return b;
    }
}
// Test1.java
A<Integer> a=new A<Integer>();
a.setB(1);
int b=a.getB();//不需要做类型强转，自动完成
String c=(String)a.getB();//编译期报错,直接编译不通过
```

“参数化集合类的类型”

把类型当作参数

“实例化集合类的类型”

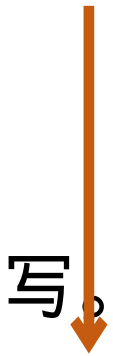
对参数化的类型“赋值”

getB 方法时不需要手动做类型强转，其实并不是不需要，而是编译器给我们进行了处理，具体来讲，泛型方法的返回类型是被擦除了，并不会进行强转，而是在调用方法的地方插入了强制类型转换

二、使用泛型

通配符

1. 泛型的声明



interface List<T> 和 class TestGen<K,V>

其中，T,K,V 不代表值，而是表示类型。这里使用任意字母都可以。常用 T 表示，是 Type 的缩写。

2. 泛型的实例化：

一定要在类名后面指定类型参数的值（类型）。如：

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

Iterator<Customer> iterator = customers.iterator();
```

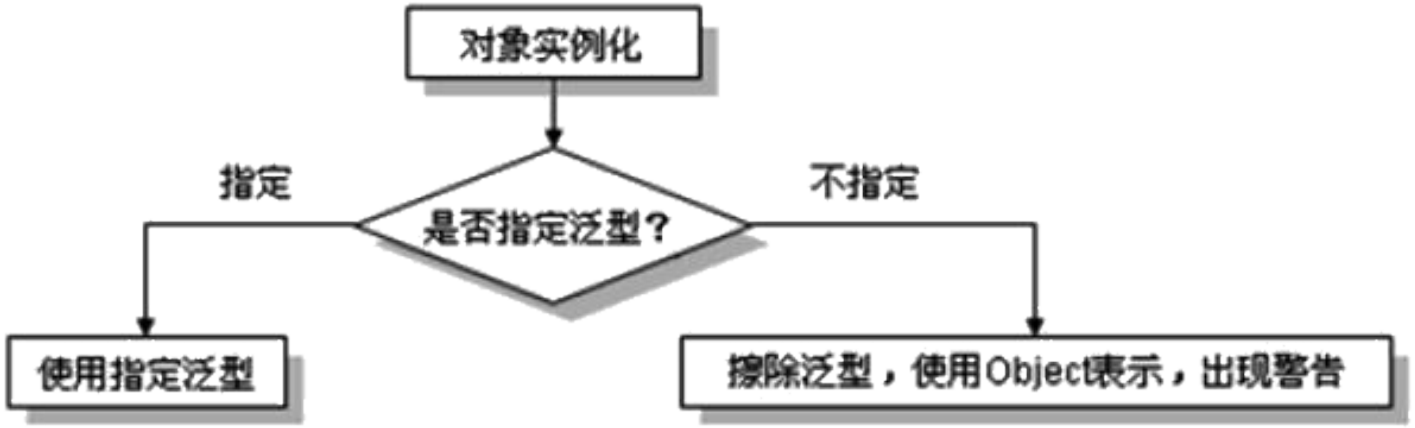
- T 只能是类，不能用基本数据类型填充。

泛型的几个重要使用

1. 在集合中使用泛型
2. 自定义泛型类
3. 泛型方法
4. 泛型接口

对于泛型类（含集合类）

1. 对象实例化时不指定泛型，默认为： Object 。
2. 泛型不同的引用不能相互赋值。
3. 加入集合中的对象类型必须与指定的泛型类型一致。



对于泛型类（含集合类）

4. 静态方法中不能使用类的泛型

5. 如果泛型类是一个接口或抽象类，则不可创建泛型

类的对象。

6. 不能在 catch 中使用泛型

7. 从泛型类派生子类，泛型类型需具体化

把一个集合中的内容限制为一个特定的数据类型，这就是 generics 背后的核心思想。

3.2 自定义泛型类

```
class Person<T>{
        // 使用 T 类型定义变量
        private T info;
        // 使用 T 类型定义一般方法
        public T getInfo(){
                return info;
        }
        public void setInfo(T info){
                this.info = info;
        }
        // 使用 T 类型定义构造器
        public Person(){}
        public Person(T info){
                this.info = info;
        }
        //static 的方法中不能声明泛型
        //public static void show(T t){
        //}
        // 不能在 try-catch 中使用泛型定义
        //try{}
        //catch(T t){}
        }
```

一个最简单的泛型类和方法定义：

```
11 class Box<T> {
12
13     private T data;
14
15     public Box() {
16
17     }
18
19     public Box(T data) {
20         this.data = data;
21     }
22
23     public T getData() {
24         return data;
25     }
26
27 }
```

```
1 public class GenericTest {
2
3     public static void main(String[] args) {
4
5         Box<String> name = new Box<String>("corn");
6         System.out.println("name:" + name.getData());
7     }
8
9 }
```

通配符

常见的如 T 、 E 、 K 、 V 等形式的参数常用于表示泛型形参，由于接收来自外部使用时候传入的类型实参。

```java
public class A<T> {
    private T b;
    public void setB(T b) {
        this.b = b;
    }
    public T getB() {
        return b;
    }
}
// Test1.java
A<Integer> a=new A<Integer>();
a.setB(1);
int b=a.getB();//不需要做类型强转，自动完成
String c=(String)a.getB();//编译期报错,直接编译不通过
```

getB 方法时不需要手动做类型强转，其实并不是不需要，而是编译器给我们进行了处理，具体来讲，泛型方法的返回类型是被擦除了（泛型擦除）

一个最简单的泛型类和方法

```
11 class Box<T> {
12
13     private T data;
14
15     public Box() {
16
17     }
18
19     public Box(T data) {
20         this.data = data;
21     }
22
23     public T getData() {
24         return data;
25     }
26
27 }
```

```
1 public class GenericTest {
2
3     public static void main(String[] args) {
4
5         Box<String> name = new Box<String>("corn");
6         System.out.println("name:" + name.getData());
7     }
8
9 }
```

泛型类型参数不能是基本类型。例如直接使用 new ArrayList<int>() 是不合法的，因为类型擦除后会替换成 Object(如果通过 extends 设置了上限，则替换成上限类型)， int 显然无法替换成 Object ，**泛型参数必须是引用类型**

泛型的目的只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦出。

也就是说，成功编译过后的 class 文件中是不包含任何泛型信息的。泛型信息不会进入到运行时阶段。

```java
public class A<T> {
    private void test1(Object arg) {
        if (arg instanceof T) { // 编译不通过
        }
    }
    private void test2() {// 编译不通过
        T obj = new T();
    }
    private void test3() {// 编译不通过
        T[] vars = new T[10];
    }
}
```

Java 泛型在很大程度上是 Java 语言中的东西而不是虚拟机中的，Java 程序编译期间会将泛型信息擦除而转变为非泛型类，例如 List<String> 和 List<Integer> 在编译后擦除了泛型类型只留下了原始类，因此 Java 虚拟机看到的都只是 List

泛型擦除会导致任何在运行时需要知道确切类型信息的操作都无法编译通过。例如 test1 ，test2 ，test3 都无法编译通过，这里说明下，**instanceof 语句是不可以直接用于泛型比较的，上文代码中，a instanceof A<integer> 不可以**，**但是 a instanceof A 或者 a instanceof A<?> 都是没有问题的**，只是具体的泛型类型不可以使用 instanceof

```java
public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");
        Box<Integer> age = new Box<Integer>(712);

        System.out.println("name class:" + name.getClass());
        System.out.println("age class:" + age.getClass());
        System.out.println(name.getClass() == age.getClass());

    }

}
```

泛型类型在逻辑上看以看成是多个不同的类型，擦除后实际上都是相同的基本类型。

运行结果为 true

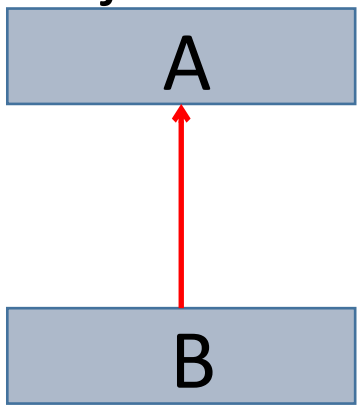
```java
class Box<T> {

    private T data;

    public Box() {

    }

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

    public T getData() {
        return data;
    }

}
```

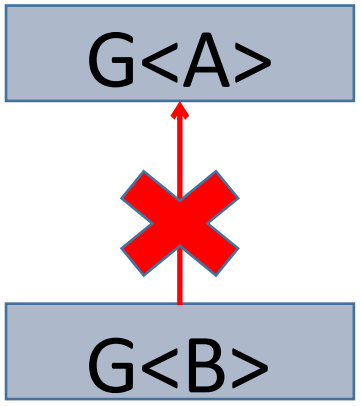
在使用泛型类时，虽然传入了不同的泛型实参，但并没有真正意义上生成不同的类型，传入不同泛型实参的泛型类在内存上只有一个，即还是**原来的最基本的类型（本实例中为 Box）**，当然，在逻辑上我们可以理解成多个不同的泛型类型

泛型和继承的关系

如果 **B 是 A 的一个子类型**（子类或者子接口），而 G 是具有泛型声明的类或接口， G<B> 并不是 G<A> 的子类型！，即 **G<B> 和 G<A> 是并列关系，不是继承关系**

比如： String 是 Object 的子类，但是 List<String > 并不是 List<Object> 的子类。





泛型的通配符

限定型通配符

泛型中的通配符一般分为非限定通配符和限定通配符两种，限定通配符有两种：<? extends T> 和 <? super T> 。

<? extends T> 保证泛型类型必须是 T 的子类来设定泛型类型的上边界，**<? super T> 来保证泛型类型必须是 T 的父类**来设定类型的下边界，泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。

非限定通配符指的是 <?> 这种形式，可以用任意泛型类型来代替，因为泛型是不支持继承关系的，所以 <?> 很大程度上弥补了这一不足。

无限定型通配符

有两个 List ，一个是 List<Integer> ，一个是 List<String> ，想用一个方法打印下 list 里面的值，因为泛型是无法继承的， List<Integer> 和 List<Object> 是没有关系的，我们此时可以借助于通配符解决

```java
public class Test1 {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(12);
        handle(list);
        List<Float> list1 = new ArrayList<Float>();
        list1.add(123.0f);
        handle(list1);
    }
    private static void handle(List<?> list) {
        System.out.println(list.get(0));
    }
}
```

类型通配符一般是使用 ？ 代替具体的类型实参。此处是类型实参，而不是类型形参！且 Box<?> 在逻辑上是 Box<Integer> 、Box<Number>... 等所有 Box< 具体类型实参 > 的父类。

```java
public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");
        Box<Integer> age = new Box<Integer>(712);
        Box<Number> number = new Box<Number>(314);

        getData(name);
        getData(age);
        getData(number);
    }

    public static void getData(Box<?> data) {
        System.out.println("data :" + data.getData());
    }

}
```

类型通配符

类型通配符

```java
class Box<T> {

    private T data;

    public Box() {

    }

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

    public T getData() {
        return data;
    }

}
```

有限制的通配符—类型通配符上限和类型通配符下限

<?>
允许所有泛型的引用调用
举例：
<? extends Number>　（无穷小 , Number]　　它和它的子类型
只允许泛型为 **Number 及 Number 子类**的引用调用

<? super Number>　　[Number , 无穷大)
只允许泛型为 **Number 及 Number 父类**的引用调用它和它的父类型

<? extends Comparable>
只允许泛型为实现 Comparable 接口的实现类的引用调用

使用泛型方法打印不同字符串的元素

```java
public class GenericMethodTest
{
   // 泛型方法 printArray
   public static < E > void printArray( E[] inputArray )
   {
      // 输出数组元素
         for ( E element : inputArray ){
            System.out.printf( "%s ", element );
         }
         System.out.println();
    }

    public static void main( String args[] )
    {
        // 创建不同类型数组： Integer, Double 和 Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "整型数组元素为:" );
        printArray( intArray  ); // 传递一个整型数组

        System.out.println( "\n双精度型数组元素为:" );
        printArray( doubleArray ); // 传递一个双精度型数组

        System.out.println( "\n字符型数组元素为:" );
        printArray( charArray ); // 传递一个字符型数组
    }
}
```

泛型示例：效果类似于重载

```
整型数组元素为:
1 2 3 4 5

双精度型数组元素为:
1.1 2.2 3.3 4.4

字符型数组元素为:
H E L L O
```

限制那些被允许传递到一个类型参数的类型种类范围。声明一个有界的类型参数，首先列出类型参数的名称，后跟 extends 关键字，最后紧跟它的上界。

右边的例子演示了 "extends" 如何使用在一般意义上的意思 "extends"（类）或者 "implements"（接口）。**该例子中的泛型方法返回三个可**

泛型示例：效果类似于重载

```
public class MaximumTest
{
   // 比较三个值并返回最大值
   public static <T extends Comparable<T>> T maximum(T x, T y, T z)
   {
      T max = x; // 假设x是初始最大值
      if ( y.compareTo( max ) > 0 ){
         max = y; //y 更大
      }
      if ( z.compareTo( max ) > 0 ){
         max = z; // 现在 z 更大
      }
      return max; // 返回最大对象
   }
   public static void main( String args[] )
   {
      System.out.printf( "%d, %d 和 %d 中最大的数为 %d\n\n",
                  3, 4, 5, maximum( 3, 4, 5 ) );

      System.out.printf( "%.1f, %.1f 和 %.1f 中最大的数为 %.1f\n\n",
                  6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

      System.out.printf( "%s, %s 和 %s 中最大的数为 %s\n","pear",
         "apple", "orange", maximum( "pear", "apple", "orange" ) );
   }
}
```

```
3, 4 和 5 中最大的数为 5

6.6, 8.8 和 7.7 中最大的数为 8.8

pear, apple 和 orange 中最大的数为 pear
```

泛型示例

```
public class Box<T> {

  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    Box<String> stringBox = new Box<String>();

    integerBox.add(new Integer(10));
    stringBox.add(new String(" 山清水秀 "));

    System.out.printf(" 整型值为   :%d\n\n", integerBox.get());
    System.out.printf(" 字符串为   :%s\n", stringBox.get());
  }
}
```

整型值为   :10

字符串为   : 山清水秀

泛型示例
限定只存在于编译时的语法层面上

```
import java.util.*;

public class GenericTest {

    public static void main(String[] args) {
        List<String> name = new ArrayList<String>();
        List<Integer> age = new ArrayList<Integer>();
        List<Number> number = new ArrayList<Number>();

        name.add("icon");
        age.add(18);
        number.add(314);

        getData(name);
        getData(age);
        getData(number);

    }

    public static void getData(List<?> data) {
      System.out.println("data :" + data.get(0));
    }
}
```

```
data :icon
data :18
data :314
```

因为 getData() 方法的参数是 List 类型的，所以 name ，age ， number 都可以作为这个方法的实参，这就是通配符的作用

类型通配符一般是使用？代替具体的类型参数。

例如 List<?> 在逻辑上是 List<String>,List<Integer> 等所有 List< 具体类型实参 > 的父类

泛型示例

```
import java.util.*;

public class GenericTest {

    public static void main(String[] args) {
        List<String> name = new ArrayList<String>();
        List<Integer> age = new ArrayList<Integer>();
        List<Number> number = new ArrayList<Number>();

        name.add("icon");
        age.add(18);
        number.add(314);

        //getUperNumber(name);//1
        getUperNumber(age);//2
        getUperNumber(number);//

    }

    public static void getData(List<?> data) {
        System.out.println("data :" + data.get(0));
    }

    public static void getUperNumber(List<? extends Number> data) {
        System.out.println("data :" + data.get(0));
    }
}
```

```
data :18
data :314
```

在 (//1) 处会出现错误，因为 getUperNumber() 方法中的参数已经限定了参数

泛型上限为 Number ，所以泛型为 String 是不在这个范围之内，所以会报错

类型通配符上限通过形如 List 来定义，如此定义就是通配符泛型值接受 Number 及其下层子类类型

范例：泛型应用

用户在设计类的时候往往会使用类的关联关系，例如，一个人中可以定义一个信息的属性，但是一个人可能有各种各样的信息（如联系方式、基本信息等），所以此信息属性的类型就可以通过泛型进行声明，然后只要设计相应的信息类即可。

泛型应用：效果类似于重载（严格讲也属于多态）

GenericPerson.java

第 8 讲 Java OOP- 泛型 & 枚举 & 注解

8.1 泛型

8.2 枚举

8.3 注解

一、枚举类

主要内容：

- 如何自定义枚举类
- 如何使用 enum 定义枚举类
  - 枚举类的主要方法
- 实现接口的枚举类

枚举类入门

- JDK1.5 之前需要自定义枚举类

- JDK 1.5 新增的 **enum 关键字**用于定义**枚举类**
- 若枚举只有一个成员，则可以作为一种单例模式的实现方式

枚举类的属性

- **枚举类对象的属性不应允许被改动，所以应该使用 private final 修饰**
  - 枚举类的使用 private final 修饰的属性应该在构造器中为其赋值
  - 若枚举类显式的定义了带参数的构造器，则在列出枚举值时也必须对应的传入参数

Enum 枚举类

- **必须在枚举类的第一行声明枚举类对象。**
- 枚举类和普通类的区别：
  - 使用 enum 定义的枚举类**默认继承**了 java.lang.Enum 类
  - 枚举类的构造器只能使用 private 访问控制符
  - 枚举类的所有实例必须在枚举类中显式列出 (, 分隔 ; 结尾). **列出的实例系统会自动添加 public static final 修饰**
- **JDK 1.5 中可以在 switch 表达式中使用 Enum 定义的枚举类的对象作为表达式**, case 子句可以直接使用枚举值的名字，无需添加枚举类作为限定

使用 Enum 定义的 Season

```
enum SeasonEnum{

    SPRING("春天", "春风又绿江南岸"),
    SUMMER("夏天", "映日荷花别样红"),
    AUTUMN("秋天", "秋水共长天一色"),
    WINTER("冬天", "窗含西岭千秋雪");

    private final String seasonName;
    private final String seasonDesc;

    private SeasonEnum(String seasonName, String seasonDesc){
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }

    public String getSeasonName() {
        return seasonName;
    }

    public String getSeasonDesc() {
        return seasonDesc;
    }
}
```

Enum 枚举类

- **枚举类的主要方法：**
  - **values() 方法**：返回枚举类型的对象数组。该方法可以很方便地遍历所有的枚举值。
  - **valueOf(String str)**：可以把一个字符串转为对应的枚举类对象。要求字符串必须是枚举类对象的“名字”。如不是，会有运行时异常。

枚举的方法

方法名	详细描述
valueOf	传递枚举类型的 Class 对象和枚举常量名称给静态方法 valueOf，会得到与参数匹配的枚举常量。
toString	得到当前枚举常量的名称。你可以通过重写这个方法来使得到的结果更易读。
equals	在枚举类型中可以直接使用“==”来比较两个枚举常量是否相等。Enum 提供的这个 equals()方法，也是直接使用“==”实现的。它的存在是为了在 Set、List 和 Map 中使用。注意，equals()是不可变的。
hashCode	Enum 实现了 hashCode()来和 equals()保持一致。它也是不可变的。
getDeclaringClass	得到枚举常量所属枚举类型的 Class 对象。可以用它来判断两个枚举常量是否属于同一个枚举类型。
name	得到当前枚举常量的名称。建议优先使用 toString()。
ordinal	得到当前枚举常量的次序。
compareTo	枚举类型实现了 Comparable 接口，这样可以比较两个枚举常量的大小（按照声明的顺序排列）
clone	枚举类型不能被 Clone。为了防止子类实现克隆方法，Enum 实现了一个仅抛出 CloneNotSupportedException 异常的不变 Clone()。

实现接口的枚举类

- 和普通 Java 类一样，枚举类可以实现一个或多个接口

- **若需要每个枚举值在调用实现的接口方法呈现出不同的行为方式，则可以让每个枚举值分别来实现该方法**

第 8 讲 Java OOP- 泛型 & 枚举 & 注解

8.1 泛型

8.2 枚举

8.3 注解

二、注解 Annotation

主要内容

- JDK 内置的基本注解类型（ 3 个 ）
- 自定义注解类型
- 对注解进行注解（ 4 个 ）
- 利用反射获取注解信息（在反射部分涉及 ）

注解（Annotation）概述

- 从 JDK 5.0 开始，Java 增加了对元数据 (MetaData) 的支持，也就是 Annotation(注解)
- Annotation 其实就是代码里的**特殊标记**，这些标记可以在编译，类加载，运行时被读取，并执行相应的处理．通过使用 Annotation，程序员可以在不改变原有逻辑的情况下，在源文件中嵌入一些补充信息．
- Annotation 可以像修饰符一样被使用，可用于**修饰包，类，构造器，方法，成员变量，参数，局部变量的声明**，这些信息被保存在 Annotation 的“name=value”对中．
- Annotation 能被用来为程序元素 (类，方法，成员变量等) 设置元数据

基本的 Annotation

- 使用 Annotation 时要在其前面增加 @ 符号，并**把该 Annotation 当成一个修饰符使用。**用于修饰它支持的程序元素
  - **@Override:** 限定重写父类方法，该注释只能用于方法
  - **@SuppressWarnings:** 抑制编译器警告

自定义 Annotation

- 定义新的 Annotation 类型使用 **@interface** 关键字
- **Annotation 的成员变量在 Annotation 定义中以无参数方法的形式来声明．其方法名和返回值定义了该成员的名字和类型．**
- 可以在定义 Annotation 的成员变量时为其指定初始值，**指定成员变量的初始值可使用 default 关键字**

```
public @interface MyAnnotation{
        String name() default “atguigu";
}
```

- 没有成员定义的 Annotation 称为标记；包含成员变量的 Annotation 称为元数据 Annotation

提取 Annotation 信息

- JDK 5.0 在 java.lang.reflect 包下新增了 **AnnotatedElement 接口**，该接口**代表程序中可以接受注解的程序元素**
- **当一个 Annotation 类型被定义为运行时 Annotation 后**，该注释才是运行时可见，当 class 文件被载入时保存在 class 文件中的 Annotation 才会被虚拟机读取
- 程序可以调用 AnnotationElement 对象的如下方法来访问 Annotation 信息

<T extends Annotation> T	**getAnnotation**(Class<T> annotationClass) Returns this element's annotation for the specified type if such an annotation is present, else null.
Annotation[]	**getAnnotations**() Returns all annotations present on this element.
Annotation[]	**getDeclaredAnnotations**() Returns all annotations that are directly present on this element.
boolean	**isAnnotationPresent**(Class<? extends Annotation> annotationClass) Returns true if an annotation for the specified type is present on this element, else false.

JDK 的元 Annotation

- JDK 的元 Annotation 用于修饰其他 Annotation 定义

- JDK5.0 提供了专门在注解上的注解类型，分别是：
  - **Retention**
  - **Target**
  - **Documented**
  - **Inherited**

元数据
String name = “atguigu”;

JDK 的元 Annotation

- @Retention: 只能用于修饰一个 Annotation 定义，用于指定该 Annotation 可以保留多长时间，@Rentention 包含一个 **RetentionPolicy** 类型的成员变量，使用 @Rentention 时必须为该 value 成员变量指定值：

  - **RetentionPolicy.SOURCE:** 编译器直接丢弃这种策略的注释
  - **RetentionPolicy.CLASS**: 编译器将把注释记录在 class 文件中。当运行 Java 程序时，JVM 不会保留注解。这是默认值
  - **RetentionPolicy.RUNTIME: 编译器将把注释记录在 class 文件中。当运行 Java 程序时，JVM 会保留注释。程序可以通过反射获取该注释**

```
public enum RetentionPolicy{
        SOURCE,
        CLASS,
        RUNTIME
}

@Retention(RetentionPolicy.SOURCE)
@interface MyAnnotation1{  }
@interface MyAnnotation2{  }

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation3{  }
```

JDK 的元 Annotation

- **@Target**: 用于修饰 Annotation 定义，用于指定被修饰的 Annotation 能用于修饰哪些程序元素 .
  @Target 也包含一个名为 value 的成员变量 .
- @Documented: 用于指定被该元 Annotation 修饰的 Annotation 类将被 javadoc 工具提取成文档 .
  - 定义为 Documented 的注解必须设置 Retention 值为 RUNTIME 。
- @Inherited: 被它修饰的 Annotation 将具有**继承性** .
  如果某个类使用了被 @Inherited 修饰的 Annotation, 则其子类将自动具有该注解
  - 实际应用中，使用较少