第19章 泛型

目 录

contents

19.1 引言

- 泛型（Generic）指可以把类型参数化，这个能力使得我们可以定义带类型参数的泛型类、泛型接口、泛型方法，随后编译器会用唯一的具体类型替换它；
- 主要优点是在编译时而不是运行时检测出错误。泛型类或方法允许用户指定可以和这些类或方法一起工作的对象类型。如果试图使用一个不相容的对象，编译器就会检测出这个错误。
- Java的泛型通过擦除法实现，和C++模板生成多个实例类不同。编译时会用类型实参代替类型形参进行严格的语法检查，然后擦除类型参数、生成所有实例类型共享的唯一原始类型。这样使得泛型代码能兼容老的使用原始类型的遗留代码。

19.1 引言

- 泛型类（Generic Class）是带形式化类型参数的类。形式化类型参数是一个逗号分隔的变量名列表，位于类声明中类名后面的尖括号<>中。下面的代码声明一个泛型类Wrapper，它接受一个形式化类型参数T：

```
public class Wrapper<T>  {

}
```

- T是一个类型变量，它可以是Java中的任何引用类型，例如String，Integer，Double等。当把一个具体的类型实参传递给类型形参T时，就得到了一系列的参数化类型（Parameterized Types），如Wrapper<String>，Wrapper<Integer> ，这些参数化类型是泛型类Wrapper<T>的实例类型（类似于Circle类型有实例对象c1,c2）

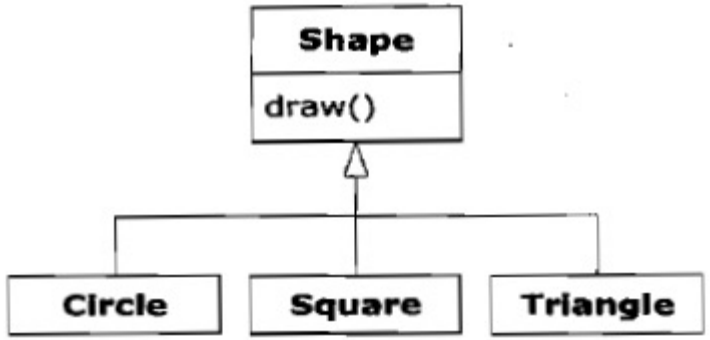
```
Wrapper<String>  stringWrapper = new Wrapper<String>();
Wrapper<Circle>  circleWrapper = new Wrapper<Circle>();
```

参数化类型（**Parameterized Types**）是在JLS里面使用的术语，为了方便描述，本章后面称为实例类型

19.1 引言

RTTI（Run-Time Type Identification）：运行时类型识别

通过运行时类型信息，程序在运行时能够检查父类引用所指的对象的实际派生类型。




```
Shape s =null;
s = new Circe();
s.draw();  //draw a circle

If(s instanceof Circle){
    System.out.println("s is a Circle")
}
```

上面的例子都是RTTI在起作用，程序在运行时，JVM知道一块内存到底是什么类型的对象

19.1 / 引言

Class类和Class对象

- 要理解RTTI在Java中的工作原理，就必须知道类型信息在运行时是如何表示的。
- 类型信息是通过Class类（类名为Class的类）的对象表示的，Java利用Class对象来执行RTTI。
- 每个类都有一个对应的Class对象，每当编写并编译了一个类，就会产生一个Class对象，这个对象当JVM加载这个类时就产生了。

19.1 引言

如何获取Class对象

- Class.forName方法，是Class类的静态方法

```
class Person{

}

class Employee extends Person{

}


class Manager extends
Employee{

}
```

```
public class ClassDemo {
    public static void main(String[] args){
        try {
            Class clz = Class.forName("ch13.Manager"); //参数是类完全限定名字符串
            System.out.println(clz.getName()); //产生完全限定名ch13.Manager
            System.out.println(clz.getSimpleName()); //产生简单名Manager

            Class superClz = clz.getSuperclass(); //获得直接父类型信息
            System.out.println(superClz.getName()); //产生完全限定名ch13.Employee
            System.out.println(superClz.getSimpleName()); //产生简单名Employee
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

产生ch13.Manager类的Class对象，赋值给Class类型的引用变量clz

注意编译器是无法检查字符串”ch13.Manager”是否为一个正确的类的完全限定名，因此在运行时可能抛出异常，比如当不小心把类名写错了时。

19.1 / 引言

如何获取Class对象

➢利用类字面常量：类名.class，得到类对应的Class对象

```
class Person{ }

class Employee extends Person{
}

class Manager extends
Employee{ }
```

```
public class ClassDemo {
   public static void main(String[] args){
      Class clz = Manager.class;  // Manager.class得到Manager的Class对象.赋给引用clz
      System.out.println(clz.getName()); //产生完全限定名ch13.Manager
      System.out.println(clz.getSimpleName()); //产生简单名Manager
   }
}
```

注意Manager是标识符（类名），因此如果Manager写错了编译器可以检查出来

某个类名.class是Class类型的字面量
正如int类型的字面量有1，2，3，
Class类型的字面量有Person.class, Employee.class, Manager.class，
它们都是Class类型的实例

➢类字面常量不仅可以用于类，也可用于数组(int[].class)，接口，基本类型，如int.class

➢相比Class.forName方法，这种方法更安全，在编译时就会被检查，因此不需要放在Try/Catch块里（见上面的标注里说明）

➢Class.forName会引起类的静态初始化块的执行，T.class不会引起类的静态初始化块的执行

19.1 / 引言

如何获取Class对象

➢通过对象。如果获得一个对象的引用o，通过o.getClass()方法获得这个对象的类型的Class对象

```
class Person{

}

class Employee extends Person{

}


class Manager extends
Employee{

}
```

```
public class ClassDemo {
    public static void main(String[] args){
        Object o = new Manager();
        Class clz = o.getClass();
        System.out.println(clz.getName()); //产生完全限定名ch13.Manager
        System.out.println(clz.getSimpleName()); //产生简单名Manager
    }
}
```

注意：getClass返回的是运行时类型

19.1 / 引言

泛化的Class引用

//非泛化的Class引用（即不带类型参数的Class引用）可指向任何类型的Class对象，但这样不安全
Class clz ;　//注意警告， Class is a raw type. References to generic type Class<T> should be parameterized
clz= Manager.class; //OK
clz = int.class; //OK

//有时我们需要限定Class引用能指向的类型：加上<类型参数>。这样可以可以强制编译器进行额外的类型检查
Class<Person> genericClz;　//泛化Class引用，Class<Person>只能指向Person的类型信息，　<Person>为类型参数
genericClz = Person.class;　//OK
//genericClz = Manager.class; //Error，不能指向非Person类型信息。注意对于类型参数，编译器检测时不看继承关系。

//能否声明一个可用指向Person及其子类的Class对象的引用？为了放松泛化的限制，用通配符?表示任何类型，并且与extends结合，创建一个范围
Class<? extends Person> clz2;　//引用clz2可以指向Person及其子类的类型信息
clz2 = Person.class;
clz2 = Employee.class;
clz2 = Manager.class;
//注意Class<?> 与Class效果一样，但本质不同，一个用了泛型，一个没有用泛型。 Class<?> 等价于Class<? extends Object >

反射（Reflection）：应用案例：实例化对象

- 完成这样的功能：输入一个类的完全限定名字符串（如“java.lang.String”），创建相应的对象。
  - Object o = new String(“Hello”); //注意String是类名标识符，不是字符串
- 因此，第一个解决方案：

```
Object o = null;
if( input.equals(“java.lang.String”) ){
    o = new String(“”);
}
else if( ( input.equals(“ch13.Student”) ){
   o = new Student();
}
…更多的else if语句
```

- 这种方法显然不行，因为事先不知道会输入一个什么类的完全限定名字符串，if语句不可能列出所有可能的类型

19.1 引言

反射（Reflection）：实例化对象（java.lang.reflect包）

- 利用Class对象我们可以在运行时动态地创建对象，调用对象的方法

```
class Student{
    private String name;
    public Student(){
        this.name = "unknown";
    }
    public Student(String name){
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String toString() {
        return "Name:" + name;
    }
}
```

```
public class ReflectDemo {
    public static void main(String[] args) {
        try {
            Class  clz = Class.forName("ch13.Student");
            //获取所有的Constructor对象
            Constructor[] ctors = clz.getConstructors();
                for(Constructor c : ctors){
                System.out.println(c.toString());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

显示

```
public ch13.Student()
public ch13.Student(java.lang.String)
```

clz指向一个类的Class对象时，通过 clz可以得到这个类的所有构造函数对象，一个构造函数对象类型是：

java.lang.reflect.Constructor

一个Constructor对象代表了类的一个构造函数

19.1 引言

反射（Reflection）：实例化对象（java.lang.reflect包）

- 利用Class对象我们可以在运行时动态地创建对象，调用对象的方法

```
class Student{
    private String name;
    public Student(){
        this.name = "unknown";
    }
    public Student(String name){
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String toString() {
        return "Name:" + name;
    }
}
```

```
public class ReflectDemo {
    public static void main(String[] args) {
        try {
            Class clz = Class.forName("ch13.Student");
            //获取所有的Method
            //Method[] methods = clz.getMethods(); //会显示所有方法，包括继承的
            Method[] methods = clz.getDeclaredMethods(); //本类定义的方法
            for(Method m: methods){
                System.out.println(m.toString());
            }
        } catch (Exception e) {
                e.printStackTrace();
        }
    }
}
```

显示

```
public java.lang.String ch13.Student.toString()
public java.lang.String ch13.Student.getName()
public void ch13.Student.setName(java.lang.String)
```

clz指向一个类的Class对象时，通过 clz可以得到这个类的所有方法对象，一个方法对象类型是：

java.lang.reflect.Method

一个Method对象代表了类的一个方法

19.1 引言

反射（Reflection）：实例化对象（java.lang.reflect包）

- 利用Class对象我们可以在运行时动态地创建对象，调用对象的方法

```
class Student{
    private String name;
    public Student(){
        this.name = "unknown";
    }
    public Student(String name){
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String toString() {
        return "Name:" + name;
    }
}
```

```
public class ReflectDemo {
    public static void main(String[] args) {
        try {
            Class clz = Class.forName("ch13.Student");
            //实例化对象
            //1：如有缺省构造函数，调用Class对象的newInstance方法
            Student s1 = (Student)clz.newInstance();
            //2. 调用带参数的构造函数
            Student s2 = (Student)clz.getConstructor(String.class).newInstance("John");
            //invoke method
            Method m = clz.getMethod("setName", String.class);
            m.invoke(s1, "Marry"); //调用s1对象的setName方法，实参"Marry"
            System.out.println(s1.toString());
            System.out.println(s2.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
Name:Marry
Name:John
```

首先得到参数类型为String的构造函数对象，然后调用它的newInstance方法调用构造函数，参数为“John”。等价于：Student s2 = new Student(“John”)；但是是通过反射机制调用的

clz.getMethod(“setName”, String.class);得到方法名为setName,参数为String的方法对象m，类型是Method。然后通过m.invoke去调用该方法，第一个参数为对象，第二个参数是传递给被调方法的实参。这二条语句等价于s1.setName(“Marry)，但是是通过反射去调的

19.2 动机和优点

- JDK1.5开始，Java允许定义泛型类、泛型接口和泛型方法，API中的一些接口和类使用

T表示类型形参，之后会用实际的类型实参来替换，称为“泛型实例化”（得到实例类型）。按照惯例，E或T这样的单个大写字母表示泛型类型的参数类型

```
package java.lang;

public interface Comparable {
 public int compareTo(Object o)
}
```

a）JDK 1.5 之前

```
package java.lang;

public interface Comparable<T> {
 public int compareTo(T o)
}
```

b）JDK 1.5

```
Comparable c = new Date();
System.out.println(c.compareTo("red" ));
```

a）JDK 1.5 之前

编译通过，产生运行时错误

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red" ));
```

b）JDK 1.5

产生编译时错误

19.2 动机和优点

- 泛型接

因为非泛型Comparable接口的compareTo方法的参数声明类型是Object，而传进去的实参"red"的确是Object类型。因此编译器通过。但实际运行时一个Date对象和字符串对象比较大小肯定出错。**因此在非泛型年代，保证传进去的对象与另外一个比较大小的对象实际类型的一致性是程序员的责任**

现在泛型接口Comparable<T>里的compareTo方法的参数类型必须是T，这就规定了要比较的另外一个对象类型必须是T，即和this对象的类型必须一致，否则编译器就可以检查出类型不一致。Comparable<Date> 规定了要比较大小的对象类型必须是Date，因此现在传字符串进去编译报错。这里Date是类型实参，T是类型形参。**Comparable<Date> 是Comparable<T>的类型实例，因为Comparable<T>是泛型接口，而Comparable<Date> 是具体接口类型。**

```
public interface Comparable {
 public int compareTo(Object o)
}
```

a）JDK 1.5 之前

```
package java.lang;

public interface Comparable<T> {
 public int compareTo(T o)
}
```

b）JDK 1.5

```
Comparable c = new Date();
System.out.println(c.compareTo("red" ));
```

a）JDK 1.5 之前

编译通过，产生运行时错误

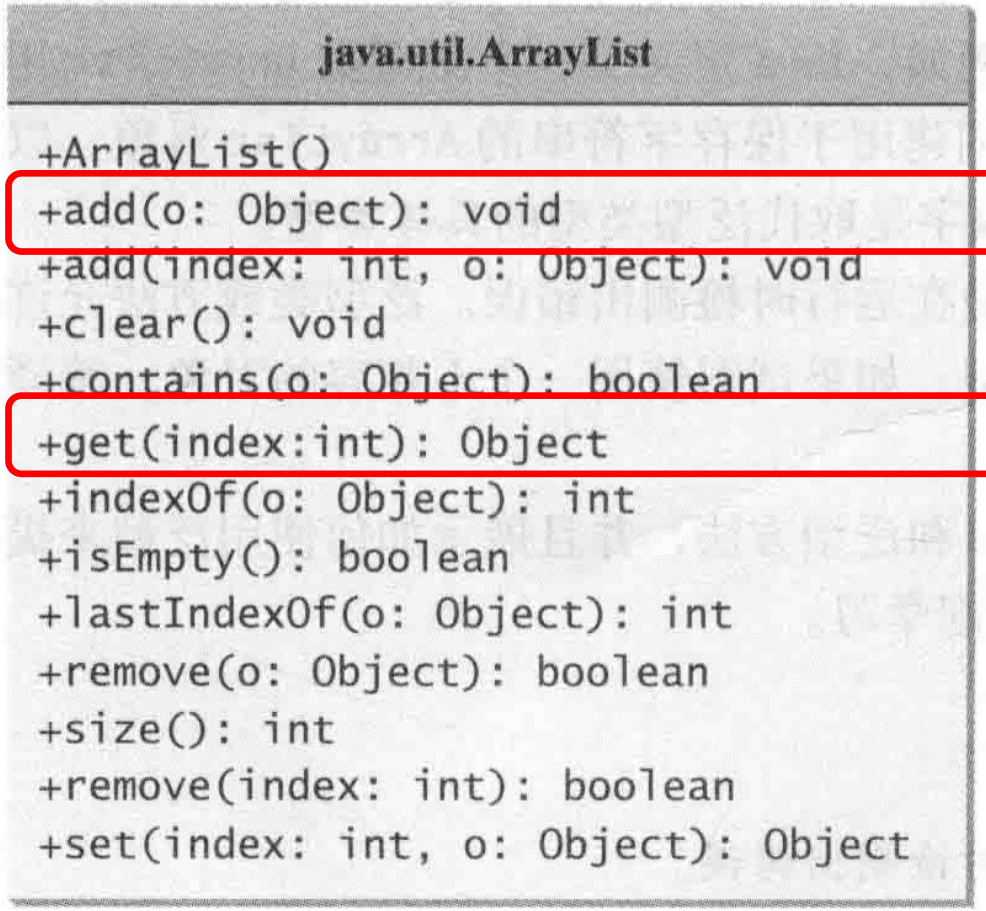
```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red" ));
```

b）JDK 1.5

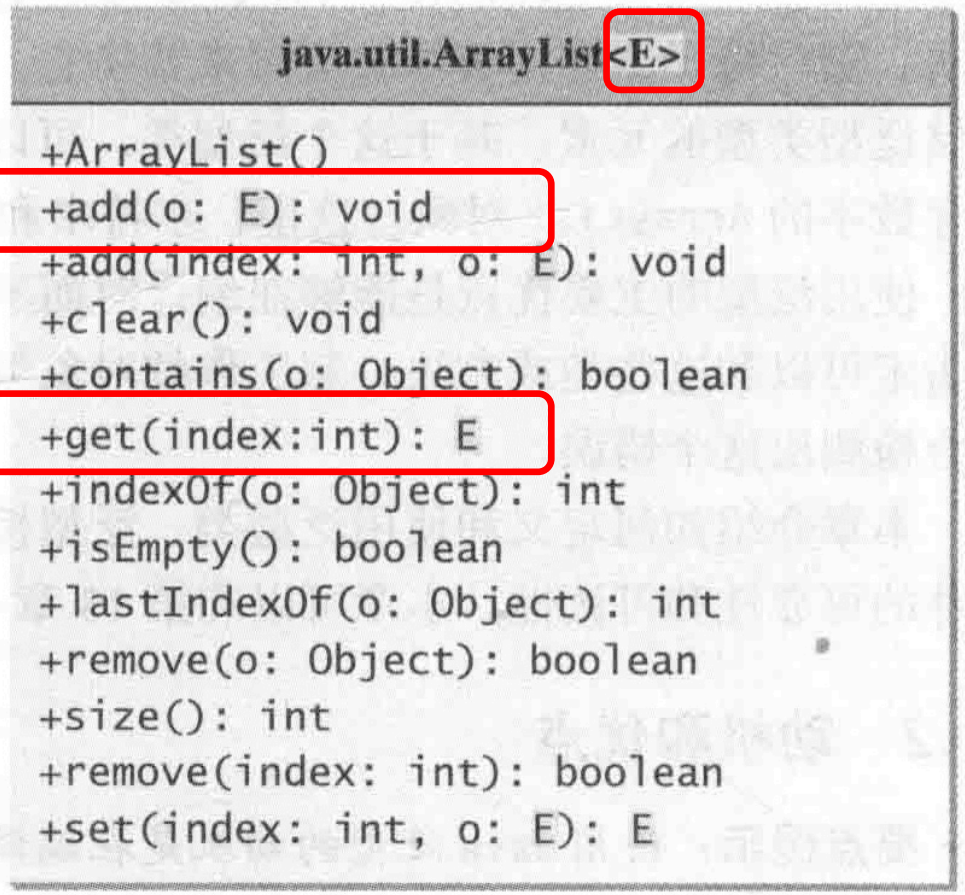
产生编译时错误

19.2 动机和优点

- 以ArrayList为例说明看看定义的不同，注意使用了泛型参数E的地方



a）JDK 1.5 之前的 ArrayList



b）从 JDK 1.5 开始的 ArrayList

19.2 动机和优点

```
public class GenericDemo1 {
    public static void main(String[] args){
        ArrayList list = new ArrayList();    //非泛型的ArrayList
        list.add("Hello");                   //非泛型的ArrayList的add方法，参数类型是Object

        //非泛型的ArrayList的get方法，返回类型也是Object，因此要强制类型转换
        String s = (String) list.get(0);   //运行起来不报错

        //但是万一程序员不小心加入了Date对象
        list.add(new Date()); //Date对象也是Object类型，因此编译器不报错
        String s2 = (String)list.get(1); //编译不报错，因为在编译时uncheck
                                         //但是在运行时就抛出异常

        //因此，对非泛型ArrayList，保证放进去对象的类型一致性变成了程序员的责任
    }
}
```

19.2 动机和

尖括号里E为类型形参，以和方法参数区分开(圆括号)规定了放入ArrayList里必须是某种类型E的对象。类型形参在类体里可以用在任何其它类型可以用的地方，如成员变量类型，方法形参类型，方法返回类型。

注意抽象父类和实现的接口List也带了类型参数E。

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>,
    RandomAccess, Cloneable, java.io.Serializable{

    // 内部就是一个Object[]数组
    transient Object[] elementData;

    E elementData(int index) {
        return (E) elementData[index];
    }

    public boolean add(E e) {
        ensureCapacityInternal(size + 1);
        elementData[size++] = e;
        return true;
    }
    public E get(int index) {
        rangeCheck(index);//检查index是否越界
        return elementData(index);
    }
    //其它代码
}
```

将下标为index的元素（注意类型是Object）强制转换为E类型返回。为什么不用instanceOf检查？
因为add方法保证了数组里元素的类型一定是E

add方法参数类型是E，这就规定了加入这个集合的元素都必须是E类型，如果add类型不一致的对象，编译器一定报错

因此，有了泛型机制，程序员的责任（保证类型一致性，例如用instanceOf检查也是程序员的责任）变成了编译器的责任。这就是引入泛型机制的优点。

调用上面elementData方法，该方法直接返回E类型

19.2 动机和优点

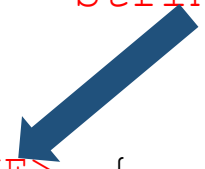
ArrayList<E>定义了一个带类型形参的泛型类，类型参数E是形参

ArrayList<String> 是一个参数化类型（**实例类型**），其中**String作为一个具体类型（实参）传递给形参E**。 这里借用了术语"实例"，不是指对象，而是一个具体的类型。

特别重要的是：类型实参String传递给类型形参E是发生在编译时（不是运行时）。因此，对于下面的语句，编译器会用String代替E，对代码进行类型检查。

```
ArrayList<String> list = new ArrayList<>();//用实例类型ArrayList<String> 声明引用变量list
list.add("China");//编译器会根据类型实参String检查传入add方法的对象类型是否匹配，否则报错
```

String作为一个具体类型（实参）传递给形参E



```
public class ArrayList<E> …{
    //…
    public boolean add(E e) {}

}
```

```
public class ArrayList<String> …{
    //…
    public boolean add(String e) {}

}
```

19.2 动机和优点

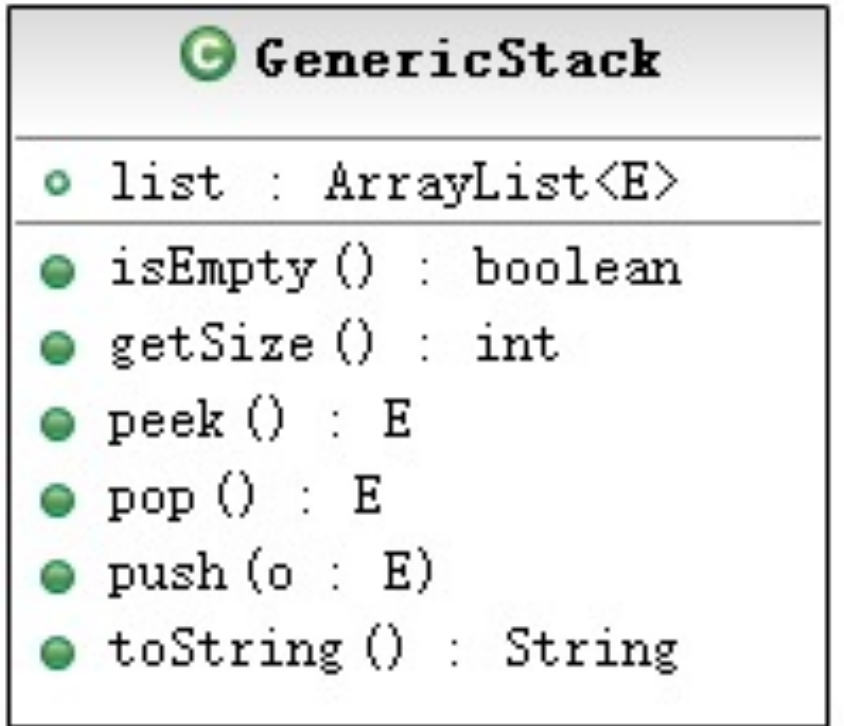
- ArrayList<String> list = new ArrayList<String>();
  - ArrayList<String> list就规定了只能往list里添加字符串
- list.add(“Hust”);　　//只能向list中添加字符串
- list.add(new Integer(1)); //错误，list中只能添加字符串
- 泛型类型的参数类型必须是引用类型
  - 如 ArrayList<int> list = new ArrayList<int>(); //错误
  - ArrayList<Integer> list = new ArrayList<Integer>(); //正确
  - list.add(5); //正确 自动打包（装箱）机制
  - int i = list.get(0); //正确，如果元素是包装类型，如Integer,Double,Character，可以直接将这个元素赋给一个基本类型的变量。这个过程称为自动拆箱机制

19.3 定义泛型类和接口

- 用泛型定义栈类

```
import java.util.ArrayList;
public class GenericStack<E> {
    private ArrayList<E> list = new ArrayList<E>();
    public boolean isEmpty() {
        return list.isEmpty();
    }
    public int getSize() {
        return list.size();
    }
    public E peek() {
        return list.get(getSize() - 1);//取值不出栈
    }
    public E pop() {
        E o = list.get(getSize() - 1) ;
        list.remove(getSize() - 1);
        return o;
    }
```

```
    public void push(E o) {
        list.add(o);
    }
    public String toString() {
        return "stack: " + list.toString();
    }
}
```



19.3 定义泛型类和接口

```
GenericStack<String> stack1 = new GenericStack< String >();
stack1.push("London");
stack1.push("Paris");
stack1.push("New York");
```

GenericStack<String>是泛型类GenericStack<E>的一个实例类型

```
GenericStack<Integer> stack2 = new GenericStack<>();
stack1.push(5);
stack1.push(10);
stack1.push(15);
```

GenericStack<Integer>是泛型类GenericStack<E>的另一个实例类型

- 注意：
- GenericStack<E>构造函数形式是擦除参数类型后的GenericStack( ),不是GenericStack<>();

```
public class GenericContainer<E> {

    //注意泛型类的构造函数不带泛型参数，连<>都不能有
    public GenericContainer(){}
}
```

19.3 定义泛型类和接口

- 注意：
- 泛型类或者泛型接口的一个实例类型，可以作为其它类的父类或者类要实现的接口
- 如Java API中，Java.lang.String类实现Comparable接口的写法是:
- public class String implements Comparable<String>

  // Comparable<String>是泛型接口Comparable<T>的实例类型 (接口），

  //Comparable<String>是一个这样接口类型：可以比较二个String对象的大小

  //而String实现了这个接口Comparable<String>
- 类似地，如果我们要定义一个实现了Comparable接口的Circle类，就得这么写
  - public class Circle implements Comparable<Circle>

19.4 泛型方法

- 除了可以定义泛型接口和泛型类，也可以定义泛型方法。下面的例子在一个非泛型类里定义了泛型方法

```
public class GenericMethodDemo {
    public static void main(String[] args) {
        Integer[] integers = {1,2,3,4,5};
        String[] strings = {"Londen","Paris","New York","Austin"};
        GenericMethodDemo.<Integer>print(integers);
        GenericMethodDemo.<String>print(strings);
    }
    public  static <E> void print(E[] list){
        for(int i = 0 ; i <list.length; i++){
            System.out.print(list[i]+" ");
        }
    }
}
```

调用泛型方法，将实际类型放于<>之中方法名之前；也可以不显式指定实际类型，而直接给实参调用，如print(integers); print(strings);由编译器自动发现实际类型

声明泛型方法，将类型参数<E>置于返回类型之前
方法的类型参数可以作为形参类型，方法返回类型，也可以用在方法体内其他类型可以用的地方

19.4 泛型方法

- 在定义泛型类、泛型接口、泛型方法时，可以将类型参数指定为另外一种类型（或泛型）的子类型（用extends），这样的类型参数称之为受限的（bounded）
- 想实现泛型方法比较二个几何对象的面积是否相等，几何对象类型很多，都从GeometricObject派生

```
public class BoundedTypeDemo{
    public  static <E  extends GeometricObject  > boolean equalArea(E object1, E object2 )
    {
        return object1.getArea() == object2.getArea( );
    }
}
```

E代表一种几何对象类型，如Circle，Triangel

E extends GeometricObject规定了E必须是GeometricObject及其子类，因此E类型对象有方法getArea

注意：类型参数放置的位置，应放在方法的返回类型之前(定义泛型方法)或者类名之后（定义泛型类时）

19.5 原始类型和向后兼容

- 没有指定具体类型实参的泛型类和泛型接口称为原始类型（raw type）。如：
  - GenericStack stack = new GenericStack( ); 等价于
  - GenericStatck<Object> stack = new GenericStack<Object>( );
- 这种不带类型参数的泛型类或泛型接口称为原始类型。使用原始类型可以向后兼容Java的早期版本。如Comparable类型.
- 尽量不要用

```
//从JDK1.5开始，Comparable就是泛型接口Comparable<T>的原始类型(raw type)
public class Max {
    public static Comparable findMax(Comparable o1, Comparable o2){
        return (o1.compareTo(o2) > 0)?o1:o2;
    }
}
```

19.5 原始类型和向后兼容

- 上例中，Comparable o1和Comparable o2都是原始类型声明，但是，原始类型是不安全的。如：Max.findMax(“Welcome”,123）；编译通过，但会引起运行时错误。
- 安全的办法是使用泛型，现在将findMax方法改成泛型方法。

```
public class Max {
    public static <E extends Comparable<E>> E findMax(E o1, E o2){
        return (o1.compareTo(o2) > 0)?o1:o2;
    }
}
```

E extends Comparable<E>>指定类型E必须实现Comparable接口，而且接口比较对象类型必须是E

注意：在指定受限的类型参数时，不管是继承父类还是实现接口，都用extends

```
public class Circle implements Comparable<Circle> {…}

Max.findMax(new Circle(),new Circle(10.0);
//编译上面这条语句时，编译器会自动发现findMax的类型实参为Circle，用Circle替换E
```

- 这个时候语句Max.findMax(“Welcome”,123）；会引起编译时错误，因为findMax方法要求两个参数类型必须一致，且E必须实现Comparable<E> 接口

19.6 通配泛型

```
public class WildCardNeedDemo {
    public static double max(GenericStack<Number> stack){
        double max = stack.pop().doubleValue();
        while (! stack.isEmpty()){
            double value = stack.pop().doubleValue();
            if(value > max)
                max = value;
        }
        return max;
    }


    public static void main(String[] args){
        GenericStack<Integer> intStack = new GenericStack<>();
        intStack.push(1);intStack.push(2);intStack.push(3);
        System.out.println("Th max value is " + max(intStack));
    }
}
```

Integer是Number的子类，但是GenericStack<Integer>并不是GenericStack<Number>的子类。

原因：泛型集合类型没有协变性

如何解决？

出错，因为intStack不是GenericStack<Number>实例

19.6 通配泛型

```
public class WildCardNeedDemo {
    public static double max(GenericStack<? extends Number> stack){
        double max = stack.pop().doubleValue();
        while (! stack.isEmpty()){
            double value = stack.pop().doubleValue();
            if(value > max)
                max = value;
        }
        return max;
    }


    public static void main(String[] args){
        GenericStack<Integer> intStack = new GenericStack<>();
        intStack.push(1);intStack.push(2);intStack.push(3);
        System.out.println("Th max value is " + max(intStack));
    }
}
```

通配泛型

double max(GenericStack<? extends Number> stack)
extends表示了类型参数的范围关系。
GenericStack<? extends Number>才是GenericStack<Integer> 的父类，
GenericStack<Number>不是GenericStack<Integer> 的父类

19.6 通配泛型

- 三种形式：
  - ？，非受限通配，等价于？extends Object，注意
    - GenericStack<?>不是原始类型， GenericStack是原始类型
  - ？ extends T, 受限通配,表示T或者T的子类，上界通配符，T定义了类型上限
  - ? super T，下限通配，表示T或者T的父类型，下界通配符，T定义了类型下限

19.6 通配泛型

- 数组的协变性（Covariant）
- 数组的协变性是指：如果类A是类B的父类，那么A[]就是B[]的父类。

```
class Fruit{}
class Apple extends Fruit{}
class Jonathan extends Apple{} //一种苹果
class Orange extends Fruit{}

//由于数组的协变性，可以把Apple[]类型的引用赋值给Friut[]类型的引用
Fruit[] fruits = new Apple[10];
fruits[0] = new Apple();
fruits[1] = new Jonathan(); // Jonathan是Apple的子类

try{
    //下面语句fruits的声明类型是Fruit因此编译通过，但运行时将Fruit转型为Apple错误
    //数组是在运行时才去判断数组元素的类型约束；
    //而泛型正好相反，在运行时，泛型的类型信息是会被擦除的，编译的时候去检查类型约束
    fruits[2] = new Fruit();//运行时抛出异常 java.lang.ArrayStoreException，这是数组协变性导致的问题
}catch(Exception e){
    System.out.println(e);
}
```

19.6 通配泛型

- 为了解决数组协变性导致的问题，Java编译器规定泛型容器（任何泛型类）没有协变性

```
ArrayList<Fruit> list = new ArrayList<Apple>(); //编译错误
//Type mismatch: cannot convert from ArrayList<Apple> to ArrayList<Fruit>
```

- 因为：我们在谈论容器的类型，而不是容器持有对象的类型
  - **A是B父类型，但泛型类(比如容器）ArrayList<A>不是ArrayList <B>的父类型**
  - 因此，上面语句报错。
- 为什么数组有协变性而泛型没有协变性：
  - 数组具有协变性是因此在运行时才去判断数组元素的类型约束（前一页PPT例子），这将导致有时发生运行时错误，抛出异常 java.lang.ArrayStoreException。这个功能在Java中是一个公认的“瑕疵”
  - 泛型没有协变性：泛型设计者认为与其在运行失败，不如在编译时就失败（禁止泛型的协变性就是为了杜绝数组协变性带来的问题，即如果泛型有协变性，面临可协变的数组一样的问题）——静态类型语言（Java,C++）的全部意义在于代码运行前找出错误。Python, JavaScript之类的语言是动态类型语言。
- 但有时希望像数组一样，一个父类型容器引用变量指向子类型容器，这时要使用通配符

19.6 通配泛型

- 采用上界通配泛型 ？ extends

```
ArrayList<? extends Fruit> list = new  ArrayList<Apple>(); //左边类型是右边类型的父类型
```

- 上面语句编译通过，但是这样的list不能加入任何东西。下面语句都会编译出错

```
list.add(new Apple()); list.add(new Fruit()); //编译都报错
//可加入null
list.add(null);

//但是从这个list取对象没有问题，编译时都解释成Fruit，运行时可以是具体的类型如Apple（有多态性）
Fruit f = list.get(0);
```

- 因为ArrayList<? extends Fruit>意味着该list集合中存放的都是Fruit的子类型（包括Fruit自身），Fruit的子类型可能有很多，但list只能存放其中的某一种类型。编译器只能知道元素类型的上限是Fruit，而无法知道list引用会指向什么具体的ArrayList，可以是ArrayList<Apple>,也可能是ArrayList<Jonathan>，为了安全，Java泛型只能将其设计成不能添加元素。
- 虽然不能添加元素，但从里面获取元素的类型都是Fruit类型（编译时）
- 因此带<? extends>类型通配符的泛型类不能往里存内容（不能set），只能读取（只能get）
- 那这样声明的容器类型有什么意义？它的意义是作为一个只读（只从里面取对象）的容器

19.6 通配泛型

- 采用上界通配泛型 ？ extends
- 假设已经实例化好了另外一个容器，对象已经放入其中，这时用`ArrayList<? extends Fruit> list`指向这个另外的容器，那么我们可以通过list取出容器的所有对象而没有任何问题

```
ArrayList<Apple> apples = new ArrayList<Apple>();
//调用apples.add方法添加很多Apple及其子类对象


ArrayList<? extends Fruit> list = apples; //现在ArrayList<? extends Fruit> 类型的引用指向apples
for(int i = 0; i < list.size(); i++){
    Fruit f = list.get(i);   //运行时从容器里取出的都是Apple及其子类对象，赋值给Fruit引用没问题
}
```

- 这个例子还是比较极端（纯粹是语法功能演示），实际更有意义的是作为方法参数：该方法接受一个放好对象的容器，然后在方法里只是逐个取出元素进行处理

19.6 通配泛型

- 采用上限通配泛型？ extends

参数类型ArrayList<? extends Fruit>

```
public static void handle(ArrayList<? extends Fruit> list){ //注意方法里只能从list get元素
        for(int i = 0; i < list.size(); i++){
                Fruit o = list.get(i); //可以确定list里面对象一定是Fruit或子类类型
                //处理对象o，注意这时调用o的实例方法时具有多态性
    }
}
```

程序其他地方创建具体类型的容器，添加对象，作为实参调handle方法

```
ArrayList<Apple> appleList = new ArrayList<>(); //等价于new ArrayList<Apple>();
appleList.add(new Apple());//ArrayList<Apple>是具体类型，编译器很清楚地知道类型参数是Apple这时可以add
//由于形参类型ArrayList<? extends Fruit>是实参类型ArrayList<Apple>的父类型，因此实参可以传给形参
handle(appleList);
```

19.6 通配泛型

- **采用下界通配泛型 ？ super**

```
//采用下界通配符 ？super T 的泛型类引用，可以指向所有以T及其父类型为类型参数的实例类型
ArrayList<? super Fruit> list = new ArrayList<Fruit>(); //这时new后边的Fruit可以省略
ArrayList<? super Fruit> list = new ArrayList<Object>(); //允许，Object是Fruit父类
ArrayList<? super Fruit> list = new ArrayList<Apple>(); //但是不能指向Fruit子类的容器
```

- **可以向list里面添加T及T的子类对象**

```
list.add(new Fruit());          //OK
list.add(new Apple());          //OK
list.add(new Jonathan());       //OK
list.add(new Orange()) ;        //OK
//list.add(new Object()); //添加Fruit父类则编译器禁止，报错
```

- **但是从list里get数据只能被编译器解释成Object**

```
Object o1 = list.get(0); //OK
Fruit o2 = list.get(0);  //报错，Object不能赋给Fruit，需要强制类型转换，
```

- **因此这种泛型类和采用？extends的泛型类正好相反：只能存数据，获取数据至少部分失效（编译器解释成Object）**

19.6 通配泛型

- ？extends 和？ super的理解

```
//现在看看通配泛型 ？ extends，注意右边的new ArrayList的类型参数必须是Fruit的子类型
//? extends Fruit指定了类型上限，因此下面的都成立：
ArrayList<? extends Fruit> list1 = new ArrayList<Fruit>(); //=号右边，如果是Fruit，可以不写，等价于new ArrayList<>();
ArrayList<? extends Fruit> list2 = new ArrayList<Apple>();  //=号右边，如果是Fruit的子类，则必须写
ArrayList<? extends Fruit> list3 = new ArrayList<Jonathan>(); //=号右边，如果是Fruit的子类，则必须写
ArrayList<? extends Fruit> list4 = new ArrayList<Orange>();  //=号右边，如果是Fruit的子类，则必须写

ArrayList<? extends Fruit> list可指向ArrayList<Fruit>|ArrayList<Apple>|ArrayList<Jonathan>| ArrayList<Orange>|…
一个ArrayList<Fruit>容器可以加入Fruit、Apple、Jonathan、Orange，
一个ArrayList<Apple>容器可以加入Apple、Jonathan，
一个ArrayList<Orange>容器可以加入Orange，
假如当ArrayList<? extends Fruit> list为方法形参时，如果方法内部调list.add，
由于编译时，编译器无法知道ArrayList<? extends Fruit>类型的引用变量会指向哪一个具体容器类型，编译器无法知道该怎么处理add。
例如当add的对象类型是Orange，如果list指向ArrayList<Apple>，加不进去。但如果list指向为ArrayList<Orange>，就可以加进去。
为了安全，编译器干脆禁止ArrayList<? extends Fruit>类型的list添加元素。
但从list里get元素，都解释成Fruit类型
```

19.6 通配泛型

- ? extends 和 ? super的理解

//? super Fruit指定了类型下限，因此下面二行都成立
ArrayList<? super Fruit> list1 = new ArrayList<Fruit>(); //=号右边，这时Fruit可以省略，等价于new ArrayList<>();
ArrayList<? super Fruit> list2 = new ArrayList<Object>(); //允许。=号右边，如果是Fruit的父类，必须写出类型
//ArrayList<? super Fruit> list3 = new ArrayList<Apple>(); //但是不能指向Fruit子类的容器

因此ArrayList<? super Fruit> list引用可以指向ArrayList<Fruit>|Fruit父类型的容器如ArrayList<Object>。
当ArrayList<? super Fruit> list为方法形参时,编译器知道list指向的具体容器的类型参数至少是Fruit。当向list里add对象o时，分析几种可能的情况：
1 o是Fruit及其子类类型，这里面又分二种情况
  1.1 ArrayList<? super Fruit> list实际指向ArrayList<Fruit>，可以加入
  1.2 ArrayList<? super Fruit> list实际指向ArrayList<Object>,可以加入
2 o是Fruit的父类型如Object，这里面又分二种情况
  2.1 ArrayList<? super Fruit> list实际指向ArrayList<Fruit>，这时编译器不允许加入，Object不能转型为Fruit
  2.2 ArrayList<? super Fruit> list实际指向ArrayList<Object>，可以加入
综合以上四种情况，可以看到，**只要对象o的类型是Fruit及其子类型，这时将对象o加入list一定是安全的（1.1, 1.2）；**
如果对象是Fruit父类型，则不允许加入最安全（因为可能出现2.1的情况)。由于? super Fruit规定了list元素类型的下限，因此取元素时编译器只能全部解释成Object

list1.add(new Fruit()); list1.add(new Apple()); list1.add(new Jonathan());//只要加入Fruit及其子类对象都OK
//list1.add(new Object()); //添加Fruit父类则编译器禁止，报错

取对象时都必须解释成Object类型。因此我们说**带<? super>通配符的泛型类的get方法至少是部分失效**
Object o1 = list.get(0);
//Fruit o2 = list.get(0); //报错，Object不能赋给Fruit，需要强制类型转换，但是引入泛型就是想去掉强制类型转换

19.6 通配泛型

- ? extends 和 ? super的使用原则

Producer Extends，Consumer Super原则（PECS）

Producer Extends：如果需要一个只读泛型类，用来Produce T，那么用 ? extends T

Consumer Super： 如果需要一个只写泛型类，用来Consume T，那么用 ? super T

如果需要同时读取和写入，那么就不能用通配符。

例如我们读JDK的Collections类的copy方法，可以看到一个经典使用PECS原则的例子：

```
public class Collections {
    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        //其中重要的代码片段为
         for (int i=0; i<srcSize; i++)
                dest.set(i, src.get(i)); //dest:写，src：读
         }
    }
    //...
}
```

19.6 通配泛型

```
public class SuperWildCarDemo {
    public static void main(String[] args) {
        GenericStack<String> strStack= new GenericStack<>();
        GenericStack<Object> objStack = new GenericStack<>();
        objStack.push(“Java");
        objStack.push(2); //装箱
        strStack.push(“Sun");
        add(strStack , objStack);
    }
    public static <T> void add(GenericStack<T> stack1,
                               GenericStack<? super T> stack2){
        while(!stack1.isEmpty())
                stack2.push(stack1.pop());
    }
}
```

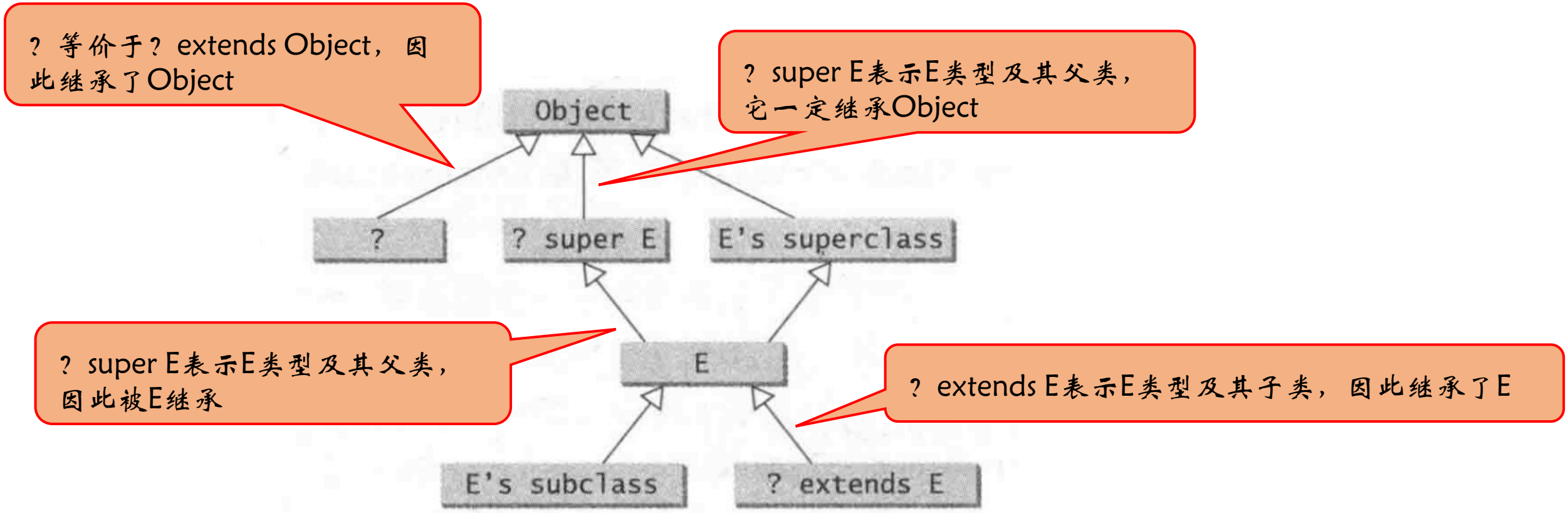
编译时，编译器根据实参strStack的类型参数String确定类型形参T为String。GenericStack<? super T> stack2表示元素类型是String父类的堆栈，因此实参objStack的类型GenericStack<Object>与形参stack2的类型匹配，参数传递没有问题。注意stack2只写

19.6 通配泛型

```
public class SuperWildCarDemo {
    public static void main(String[] args) {
        GenericStack<String> strStack= new GenericStack<>();
        GenericStack<Object> objStack = new GenericStack<>();
        objStack.push(“Java");
        objStack.push(2); //装箱
        strStack.push(“Sun");
        add(strStack, objStack);
    }
    public static <T> void add(GenericStack<? extends T> stack1,
                               GenericStack<T> stack2){
        while(!stack1.isEmpty())
                stack2.push(stack1.pop());
    }
}
```

add方法参数改成(GenericStack<？Extends T> stack1,GenericStack<T> stack2)也没有问题。这时根据第二个实参类型推断出T为Object，而第一个实参类型GenericStack<String>是GenericStack<？Extends Object>的子类型，因此参数传递也没问题。注意stack1只读。

19.6 通配泛型

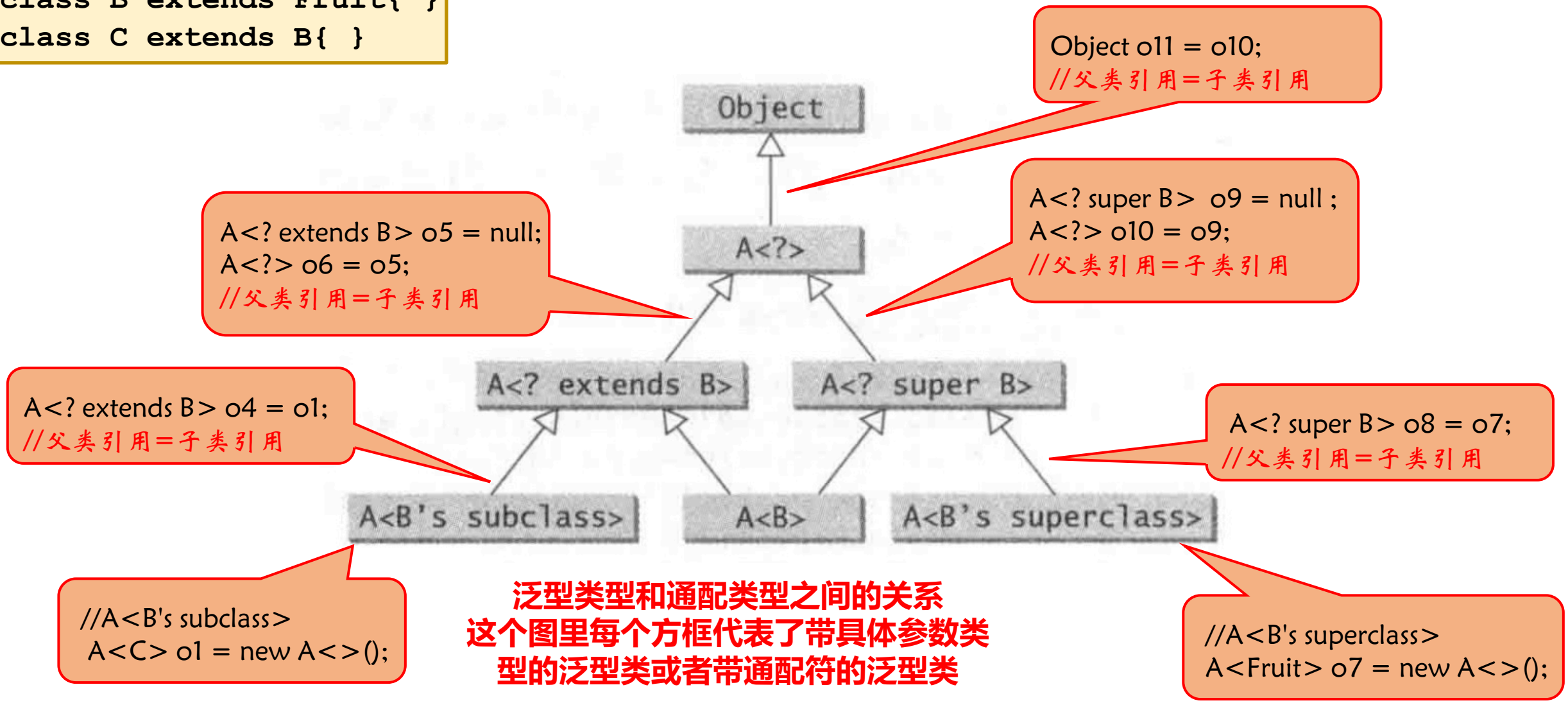


通配类型

19.6 通配泛型

注意前面关于通配泛型的讨论都适用于任何泛型类（不仅是泛型容器类）

```
class A<T> { } //泛型类
class B extends Fruit{ }
class C extends B{ }
```



19.7 泛型擦除和对泛型的限制

- 泛型是用类型擦除（type erasure）方法实现的。泛型的作用就是使得编译器在编译时通过类型参数来检测代码的类型匹配性。当编译通过，意味着代码里的类型都是匹配的。因此，所有的类型参数使命完成而全部被擦除。因此，泛型信息(类型参数)在运行时是不可用的，这种方法使得泛型代码向后兼容使用原始代码的遗留代码。

```
ArrayList<String> list = new ArrayList<>();
list.add("Oklahoma");
String state = list.get(0);
```

a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

b)

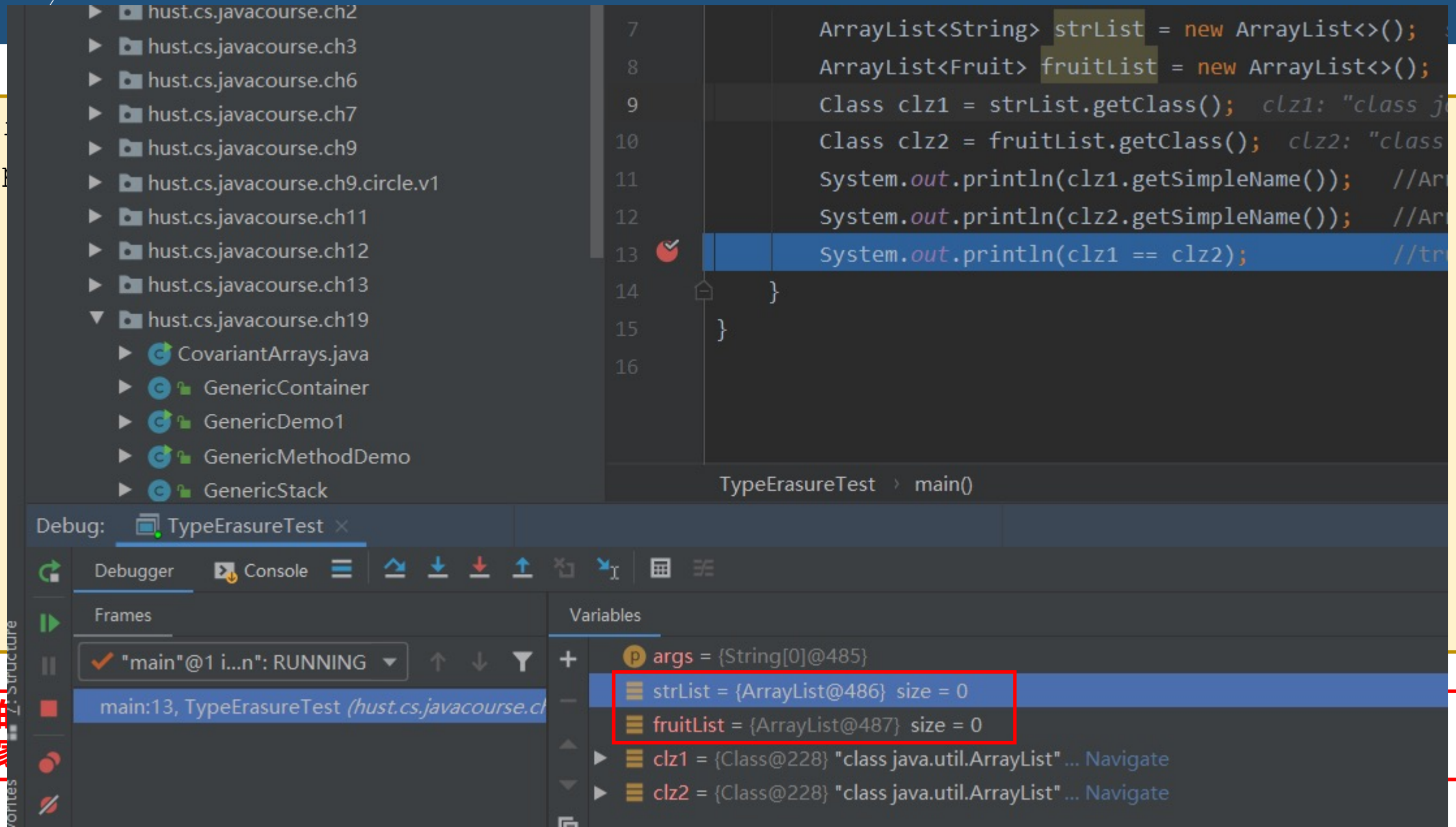
- 泛型存在于编译时，当编译器认为泛型类型是安全的，就会将其转化为原始类型。这时(a)所示的源代码编译后变成(b)所示的代码。注意在(b)里，由于list.get(0)返回的对象运行时类型一定是String，因此强制类型转换一定是安全的。

19.7 泛型擦除和对泛型的限制

```
public class TypeErasureTest {
    public static void main(String[] args){
        ArrayList<String> strList = new ArrayList<>();
        ArrayList<Fruit> fruitList = new ArrayList<>();
        Class clz1 = strList.getClass(); //getClass返回运行时信息
        Class clz2 = fruitList.getClass();
        System.out.println(clz1.getSimpleName());    //ArrayList
        System.out.println(clz2.getSimpleName());    //ArrayList
        System.out.println(clz1 == clz2);            //true
    }
}
```

所有参数化类型（实例类型）ArrayList<String> 、ArrayList<Fruit> 在运行时共享同一个类型：ArrayList。
请大家再回到PPT第2页去理解最后一段话

19.7 泛型擦除和对泛型的限制



所有的
请大家

19.7 泛型擦除和对泛型的限制

- 当编译泛型类、接口和方法时，会用Object代替非受限类型参数E。<E extends Object>

```
public static <E> void print(E[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

a)

```
public static void print(Object[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

b)

- 如果一个泛型的参数类型是受限的，编译器会用该受限类型来替换它。

```
public static <E extends GeometricObject>
    boolean equalArea(
      E object1,
      E object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

a)

```
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

b)

19.7 泛型擦除和对泛型的限制

- 泛型类会擦除类型参数，所有泛型的实例类型共享擦除后形成的原始类型如ArrayList
  - 泛型类所有实例类型在运行时共享原始类型，如：

    ArrayList<String> list1 = new ArrayList<>( );

    ArrayList<Integer> list2= new ArrayList<>( );

    在运行时只有一个擦除参数类型后的原始ArrayList类被加载到JVM中
  - 所以， list1 instanceOf ArrayList<String>是错误的，可用：

    list1 instanceOf ArrayList

    list2 instanceOf ArrayList

    instanceOf是根据运行时类型进行检查

19.7 泛型擦除和对泛型的限制

- 使用泛型类型的限制
  - 不能使用new E( ); //只能想办法得到E的类型实参的Class信息，再newInstance(...)

    不能用泛型的类型参数创建实例，如： E object = new E( ); //错误

    错误，泛型类型参数在运行时不可用
  - 不能使用new E[ ]

    不能用泛型的类型参数创建数组，如： E[ ] element = new E[cpacity]; //错误
    - new是运行是发生的，因此new 后面一定不能出现类型形参E，运行时类型参数早没了
- 强制类型转换可以用类型形参E，通过类型转换实现无法确保运行时类型转换是否成功
  - E[ ] element = (E[ ])new Object[cpacity]; //编译可通过(所谓编译通过就是指编译时uncheck，至于运行时是否出错，那是程序员自己的责任

17.7 泛型擦除和对泛型的限制

```
public class GenericOneDimensionArrayUncheck<T> {//实现一维数组的泛型包装类。不可能实现泛型数组
    private T[] elements;  //T[]类型数组存放元素
    public GenericOneDimensionArrayUncheck(int size){
        //new Object[]强制类型转换。强制类型转换就是uncheck，就是强烈要求编译器把=右边的类型解释成T[]
        elements = (T[])new Object[size]; //注意：在运行时，elements引用变量指向的是Object[]
    }
    //这里value的类型是T，这点非常重要，保证了放进去的元素类型必须是T及子类型。否则编译报错
    public void put(T value,int index){ elements[index] = value; }
    public T get(int index){ return elements[index]; }//elements声明类型就是T[]，因此类型一致
    public T[] getElements() {return elements;} //这个方法非常危险，编译没问题
    public static void main(String[] args){
        GenericOneDimensionArrayUncheck<String> strArray = new
                                        GenericOneDimensionArrayUncheck<>(10);
        strArray.put("Hello",0);
//          strArray.put(new Fruit(),0);  //不是String对象放不进去
        String s = strArray.get(0);  //strArray.get(0)返回对象的运行时类型一定是String，由put保证的
        //但是下面的语句抛出运行时异常：java.lang.ClassCastException
        //因为运行时，elements引用变量指向的是Object[]，无法转成String[]
        String[] a = strArray.getElements(); //返回内部数组，但为String[]类型
    }
}
```

这个泛型数组实现的版本去掉getElements方法后，还是可用的，通过公有的put、get方法存取元素即可。

19.7 泛型擦除和对泛型的限制

这里第一个参数是Class<? extends T> clz，表示一个T类型及其子类的Class对象。通过Class对象，可以通过反射创建运行时类型为T[]的数组。
但是Array.newInstance方法返回的是Object，因此需要强制类型转换。但这里的强制类型转换是安全的，因为创建的数组的运行时类型就是T[]

```
public class GenericOneDimensionArray<T> {
    private T[] elements = null;  //T[]类型

    public GenericOneDimensionArray(Class<? extends T> clz,int size){
        elements = (T[])Array.newInstance(clz,size);
    }

    //get, put等其他方法省略

    public T[] getElements(){ return elements; }

    public static void main(String[] args){
        GenericOneDimensionArray<String> stringArray =
                new GenericOneDimensionArray(String.class,10);
        String[] a = stringArray.getElements();  //这里不会抛出运行时异常了
//          a[0] = new Fruit(); //不是String类型的对象，编译报错
        a[1] = "Hello";
    }
}
```

Array.newInstance(数组元素类型的Class对象, size)
通过反射机制创建运行时类型为T[]的数组

这个泛型数组实现的版本比前一个要好多了，但构造函数要多传一个Class对象，指明数组元素类型信息。举这个例子还想说明反射机制的重要性。

19.7 泛型擦除和对泛型的限制

- 使用泛型类型的限制：不能new泛型数组（数组元素是泛型），但可以声明
  - 不能使用new A<E>[ ]的数组形式，因为E已经被擦除

    ArrayList<String>[ ] list = new ArrayList<String>[10]；//错误
  - E已经被擦除，只能用泛型的原始类型初始化数组, 必须改为new ArrayList[10]

    ArrayList<String> [ ] list = new ArrayList[10]；
    - 为什么这里不需要强制类型转换：参数化类型与原始类型的兼容性
    - 参数化类型对象可以被赋值为原始类型的对象，原始类型对象也可以被赋值为参数化类型对象

      ArrayList a1 = new ArrayList(); //原始类型

      ArrayList<String> a2 = a1; //参数化类型

19.7 泛型擦除和对泛型的限制

- 使用泛型类型的限制(续)
  - 静态上下文中不允许使用泛型的类型参数。由于泛型类的所有实例类型都共享相同的运行时类，所以泛型类的静态变量和方法都被它的所有实例类型所共享。因此，在静态方法、数据域或者初始化语句中，使用泛型的参数类型是非法的。

```
public class Test<E> {
  public static void m(E o1) { // Illegal
  }

  public static E o1; // Illegal

  static {
    E o2; // Illegal
  }
}
```

Test<String>和Test<Integer>这二个实例类型共享同一个运行时类型，如果静态上下文可以使用类型参数E, 会导致矛盾：
m方法的形参类型到底是String还是Integer？

19.7 泛型擦除和对泛型的限制

- 使用泛型类型的限制(续)
  - 异常类不能是泛型的。泛型类不能继承java.lang.Throwable。

```
public class MyException<T> extends Exception {
}
```

  - 非法，因为如果允许这么做，则应为MyException添加一个catch语句

```
try {
  ...
}
catch (MyException<T> ex) {
  ...
}
```

  - JVM必须检查这个从try语句中抛出的异常以确定与catch语句中的异常类型匹配，但这不可能，因为运行时的类型信息是不可获得的。

19.8 如何实现带泛型参数的对象工厂（一种设计模式）

- 使用泛型类型的限制
  - 不能使用new E( ); //只能用newInstance(...)
- 如何利用反射机制，通过newInstance(...)来创建对象

```
public class ObjectFactory<T> {
        private Class<T> type;
        public ObjectFactory(Class<T> type) {
                this.type = type;
        }
        public T create() {
                T o = null;
                try {
                        o= type.newInstance();
                } catch (InstantiationException | IllegalAccessException e) {
                        e.printStackTrace();
                }
                return o;
        }
}
```

定义私有数据成员，保存要创建的对象的类型信息

构造函数传入要创建的对象的类型信息

对象工厂的create方法负责产生一个T类型的对象，利用newInstance

这里顺便说明和强调一下，一个类定义缺省构造函数（不带参数）多么重要

19.8 如何实现带泛型参数的对象工厂

- 使用泛型类型的限制
  - 不能使用new E( ); //只能用newInstance(...)
- 如何利用反射机制，通过newInstance(...)来创建对象

```
public class Test {
        public static void main(String[] args) {
                //首先创建一个负责生产Car的对象工厂，传进去需要创建对象的类的Class信息
                ObjectFactory<Car> carFactory = new ObjectFactory<Car>(Car.class);
                Car o = carFactory.create(); //由对象工厂负责产生car对象
                System.out.println(carFactory.create().toString());
        }
}
public class Car {
        private String s = null;
        public Car() {
                s = "Car";
        }
        public String toString() {
                return s;
        }
}
```

以Car.class为参数去构造一个ObjectFactory<Car>类型的对象工厂，再调用对象工厂的create方法，一定会返回Car对象。