1 定义简单泛型类

泛型类就是有一个或多个类型变量的类。下面用一个简单的`Pair`类作为例子：

```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair()
    {
        first = null;
        second = null;
    }

    public Pair(T first, T second)
    {
        this.first = first;
        this.second = second;
    }

    public T getFirst() { return first; }
```

```
    public T getSecond() { return second; }

    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
}
```

`Pair`类引入了一个类型变量`T`，用尖括号括起来，放在类名后面。泛型类可以有多个类型变量，例如：

```
public class Pair<T, U>
```

类型变量在整个类定义中用于指定方法的返回类型以及字段和局部变量的类型。类型变量通常使用大写字母，Java库使用变量`E`表示集合的元素类型，`K`和`V`分别表示表的键和值的类型，`T`表示任意类型。

可以用具体的类型替换类型变量来实例化泛型类型，例如：

```
Pair<String>
```

可以把实例化得到的类看做一个普通类，例如上面的`Pair<String>`类有以下字段和方法：

```
字段：
    String first
    String second
构造器：
    Pair<String>()
    Pair<String>(String, String)
方法：
    String getFirst()
    String getSecond()
    void setFirst(String)
    void setSecond(String)
```

2 泛型方法

除了定义泛型类，还可以定义带有类型参数的方法。例如：

```
class ArrayAlg
{
    public static <T> T getMiddle(T... a)
    {
        return a[a.length / 2];
    }
}
```

定义泛型方法时，类型变量放在修饰符和返回类型之间。泛型方法可以在普通类中定义，也可以在泛型类中定义。

当调用一个泛型方法时，可以把具体类型包围在尖括号中，放在方法名前面。例如：

```
String middle = ArrayAlg.<String>getMiddle("John", "Q.", "Public");
```

编译器可以根据上下文推断泛型方法的具体类型，此时可以在方法调用中省略方法参数。例如，上面的方法调用可以简写为：

```
String middle = ArrayAlg.getMiddle("John", "Q.", "Public");
```

3 类型变量的限定

有时，类或方法需要对类型变量加以约束。例如，下面的泛型方法用于计算数组中的最小元素：

```
public static <T> T min(T[] a)
{
    if (a == null || a.length == 0) return null;
    T smallest = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (smallest.compareTo(a[i]) > 0) smallest = a[i];
    }
    return smallest;
}
```

这个方法中使用了`compareTo`方法，这就要求类型`T`必须实现`Comparable`接口。可以通过对类型变量`T`设置一个限定来实现这一点：

```
public static <T extends Comparable> T min(T[] a)
// Comparable 接口也是泛型类型，在此暂且省略其类型参数
```

现在，类型参数只能接受实现了`Comparable`接口的类。如果将一个没有实现`Comparable`接口的类传递给类型参数，编译器会报错。

限定类型变量的语法为：

```
<T extends BoundingType>
```

表示`T`是BoundingType的子类型。BoundingType可以是类，也可以是接口。

一个类型变量可以有多个限定，多个限定类型用`&`分隔。例如：

```
<T extends Comparable & Serialzable>
```

在Java的继承中，可以根据需要拥有多个接口超类型，但最多有一个超类，因此限定类型中最多有一个类。如果有一个类作为限定，必须把它放在限定列表的第一个。

4 泛型代码和虚拟机

4.1 类型擦除

无论何时定义一个泛型类型，都会自动提供一个相应的**原始类型**（raw type）。这个原始类型的名字就是去掉类型参数后的泛型类型名。类型变量会被擦除，并替换为其限定类型，对于无限定类型的变量则替换为`Object`。例如，上面提到的`Pair<T>`的原始类型如下所示：

```
public class Pair
{
    private Object first;
    private Object second;
```

```
    public Pair()
    {
        first = null;
        second = null;
    }

    public Pair(Object first, Object second)
    {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() { return first; }
    public Object getSecond() { return second; }

    public void setFirst(Object newValue) { first = newValue; }
    public void setSecond(Object newValue) { second = newValue; }
}
```

原始类型是一个普通的类，就像Java语言引入泛型之前实现的类一样。在程序中可以包含不同类型的`Pair`类，如`Pair<String>`或`Pair<LocalDate>`，不过擦除类型后，它们都会变成原始的`Pair`类型。

原始类型用限定类型列表中的第一个类型来替换类型变量，如果没有限定，就替换为`Object`。例如，`Pair<T>`中的类型变量没有显式的限定，因此用`Object`替换`T`。如果有另一个类型：

```
public class Interval<T extends Comparable & Serializable> implements
Serializable
{
    private T lower;
    private T upper;
    //...
    public Interval(T first, T second) { //... }
}
```

原始类型`Interval`如下所示：

```
public class Interval implements Serializable
{
    private Comparable lower;
    private Comparable upper;
    //...
    public Interval(Comparable first, Comparable second) { //... }
}
```

这里用限定类型中的`Comparable`替换`T`。如果将类型限定改成`<T extends Serializable & Comparable>`，就会用`Serializable`替换`T`，而编译器在必要时要向`Comparable`插入强制类型转换。为了提高效率，应该将标签接口（即没有方法的接口）放在限定列表的末尾。

4.2 转换泛型表达式

编写一个泛型方法调用时，如果擦除了返回类型，编译器会插入强制类型转换。例如，对于下面这段代码：

```
Pair<Employee> buddies = ...;
Employee buddy = buddies.getFirst();
```

`getFirst`方法擦除类型后的返回类型是`Object`，编译器自动插入转换到`Employee`的强制类型转换。也就是说，编译器把这个方法调用转换为两条虚拟机指令：

1. 对原始方法`Pair.getFirst`的调用。
2. 将返回的`Object`类型强制转换为`Employee`类型。

当访问一个泛型字段时也要插入强制类型转换。不过由于字段大都声明为`private`，在类外不能直接访问，因此这种情况很少出现。

4.3 转换泛型方法

类型擦除也会出现在泛型方法中。例如下面的泛型方法：

```
public static <T extends Comparable> T min(T[] a)
```

擦除类型后得到的方法为：

```
public static Comparable min(Comparable[] a)
```

方法的类型擦除带来了一些问题，通过下面的例子说明：

```
class DateInterval extends Pair<LocalDate>
{
    public void setSecond(LocalDate second) { //... }
}
```

`DateInterval`类继承了`Pair<LocalDate>`类，并覆盖了`setSecond`方法。这个方法应该具有多态性，因此可以做如下操作：

```
DateInterval interval = new DateInterval(...);
Pair<LocalDate> pair = interval;
pair.setSecond(aDate);
```

但是，类型擦除会破坏这种多态性。这个类擦除后变成：

```
class DateInterval extends Pair
{
    public void setSecond(LocalDate second) { //... }
}
```

`Pair<LocalDate>`类实际上并不存在，只存在它的原始类型`Pair`。`Pair`类的`setSecond`方法的签名为：

```
public void setSecond(Object second)
```

这两个`setSecond`方法的参数类型不同，不构成多态。可以看到，类型擦除与多态发生了冲突。为了解决这个问题，编译器在`DateInterval`方法中生成一个**桥方法**：

```
public void setSecond(Object second) { setSecond((LocalDate) second); }
```

桥方法与 `Pair` 类的 `setSecond` 方法构成多态，在桥方法中调用了 `DateInterval` 类中新定义的 `setSecond` 方法，满足了多态性的要求。

有时桥方法会很奇怪。例如：

```
class DateInterval extends Pair<LocalDate>
{
    public LocalDate getSecond() { //... }
}
```

`DateInterval` 覆盖了 `getSecond` 方法，此时在 `DateInterval` 类中有两个 `getSecond` 方法：

```
LocalDate getSecond() // 在 DateInterval 类中定义
Object getSecond() // 桥方法，与 Pair 类中的 getSecond 方法构成多态
```

正常情况下，同名方法有相同的参数类型是不合法的。但是，在虚拟机中，会由参数类型和返回类型共同指定一个方法，因此虚拟机能够正确地处理这种情况。

桥方法不仅用于泛型类型。当一个方法覆盖另一个方法时，可以指定一个更严格的返回类型，称这两个方法有可协变的返回类型。例如：

```
public class Employee implements Cloneable
{
    public Employee clone() throws CloneNotSupportedException {...}
}
```

`Cloneable` 接口中的 `clone` 方法是从 `Object` 类继承的，它的返回值是 `Object` 类型。`Employee` 类在覆盖 `clone` 方法时指定返回类型为 `Employee`，因此 `Object.clone` 和 `Employee.clone` 有可协变的返回类型。实际上，`Employee` 类有两个 `clone` 方法：

```
Employee clone() // 新定义的方法
Object clone() // 桥方法
```

5 限制与局限性

5.1 不能用基本类型实例化类型参数

不能用基本类型代替类型参数，原因在于类型擦除。类型擦除后，类型参数被替换成 `Object` 类或限制类型，这些类型不能接受基本类型的变量。所以，不能使用 `Pair<double>`，而要用 `Pair<Double>`，使用基本类型对应的包装器类型。

5.2 运行时类型查询只适用于原始类型

由于类型擦除，所有泛型类型在运行时以原始类型存在，因此所有的类型查询只产生原始类型。如果试图使用 `instanceof` 查询一个对象是否属于某个泛型类型，会产生编译错误。如果使用强制类型转换将一个对象转换成泛型类型，会产生一个警告。例如：

```
if (a instanceof Pair<String>) // 编译错误
Pair<String> p = (Pair<String>) a; // 警告
```

同理，`getClass`方法总是返回原始类型。例如：

```
Pair<String> stringPair = ...;
Pair<Employee> employeePair = ...;
if (stringPair.getClass() == employeePair.getClass()) // 一定为 true
{
    ...
}
```

5.3 不能创建参数化类型的数组

不能实例化参数化类型的数组。例如：

```
Pair<String>[] table = new Pair<String>[10]; // 错误
```

如果允许创建这样的数组，擦除之后，`table`的类型是`Pair[]`，可以把它转换为`Object[]`：

```
Object[] objarray = table;
```

考虑以下赋值：

```
objarray[0] = new Pair<Employee>();
```

尽管能够通过数组存储的类型检查，这条赋值语句能顺利运行，但在其他地方处理`table[0]`时会产生异常。出于这个原因，不允许创建参数化类型的数组。

需要说明的是，只是不允许创建这些数组，而声明类型为`Pair<String>[]`的变量仍然是合法的，不过不能用`new Pair<String>[10]`初始化这个变量。

如果需要收集参数化类型对象，使用`ArrayList<Pair<String>>`更安全、有效。

5.4 Varargs警告

考虑下面这个参数个数可变的泛型方法：

```
public static <T> void addAll(Collection<T> coll, T... ts)
{
    for (T t : ts) coll.add(t);
}
```

参数ts实际上是一个数组。考虑以下调用：

```
Collection<Pair<String>> table = ...;
Pair<String> pair1 = ...;
Pair<String> pair2 = ...;
addAll(table, pair1, pair2);
```

为了调用这个方法，虚拟机必须建立一个`Pair<String>`数组，这就违反了前面的规则。这时会得到一个警告。可以采用两种方法来抑制这个警告。

方法一：为包含`addAll`调用的方法增加注解`@SuppressWarnings("unchecked")`。

方法二：在Java 7中，可以用`@SafeVarargs`注解`addAll`方法：

```
@SafeVarargs
public static <T> void addAll(Collection<T> coll, T... ts)
```

`@SafeVarargs`只能用于声明为`static`、`final`或`private`的构造器和方法。所有其他方法都可能被覆盖，使得这个注解没有意义。

5.5 不能实例化类型变量

不能在类似`new T(...)`的表达式中使用类型变量。例如，下面的构造器是非法的：

```
public Pair()
{
    first = new T();
    second = new T();
}
```

类型擦除将`T`变成`Object`，而调用`new Object()`不符合我们的需要。

在Java 8之后，最好的解决办法是让调用者提供一个构造器方法引用。例如：

```
public static <T> Pair<T> makePair(Supplier<T> constr)
{
    return new Pair<>(constr.get(), constr.get());
}

Pair<String> p = Pair.makePair(String::new);
```

比较传统的解决方法是通过反射调用`Constructor.newInstance`方法来构造泛型对象。不过不能直接使用以下调用：

```
first = T.class.getConstructor().newInstance(); // 错误
```

表达式`T.class`是不合法的，因为它会擦除为`Object.class`。必须适当地设计API以便得到一个`Class`对象，如下所示：

```
public static <T> Pair<T> makePair(Class<T> cl)
{
    try
    {
        return new Pair<>(cl.getConstructor().newInstance(),
            cl.getConstructor().newInstance());
    }
    catch (Exception e) { return null; }
}
```

5.6 不能构造泛型数组

数组本身带有类型，用来监控虚拟机中的数组存储，这个类型会被擦除。例如：

```
public static <T extends Comparable> T[] minmax(T... a)
{
    T[] mm = new T[2]; // 错误，被擦除为 Comparable[2]
    ...
}
```

如果数组仅仅作为一个类的私有实例字段，那么可以将这个数组的元素声明为擦除的类型并使用强制类型转换。例如：

```
public class ArrayList<E>
{
    private Object[] elements;
    ...
    @SuppressWarnings("unchecked") public E get(int n) { return (E) elements[n];
}
    public void set(int n, E e) { element[n] = e; }
}
```

这个技术并不适用于上面的`minmax`方法，因为`minmax`方法返回一个`T[]`数组，如果类型不对，就会得到运行时错误结果。例如：

```
public static <T extends Comparable> T[] minmax(T... a)
{
    Comparable[] result = new Comparable[2];
    ...
    return (T[]) result;
}

String[] names = ArrayAlg.minmax("Tom", "Dick", "Harry");
    // Comparable[] 转换为 String[] 时，会抛出 ClassCastException 异常
```

在这种情况下，最好让用户提供一个数组构造器表达式：

```
public static <T extends Comparable> T[] minmax(IntFunction<T[]> constr, T... a)
{
    T[] result = constr.apply(2);
    ...
}

String[] names = ArrayAlg,minmax(String[]::new, "Tom", "Dick", "Harry");
```

比较老式的方法是利用反射：

```
public static <T extends Comparable> T[] minmax(T... a)
{
    T[] result = (T[]) Array.newInstance(a.getClass().getComponentType(), 2);
    ...
}
```

5.7 泛型类的静态上下文中变量无效

不能在静态字段或方法中引用类型变量。例如：

```
public class Singleton<T>
{
    private static T singleInstance; // 错误

    public static T getSingleInstance() // 错误
    {
        if (singleInstance == null)
        {
            ...
        }
        return singleInstance;
    }
}
```

5.8 不能抛出或捕获泛型类的实例

既不能抛出也不能捕获泛型类的对象。实际上，泛型类继承`Throwable`类是不合法的。例如：

```
public class Problem<T> extends Exception // 错误
{
    ...
}
```

`catch`子句中不能使用类型变量。例如：

```
public static <T extends Throwable> void doWork(Class<T> t)
{
    try
    {
        ...
    }
    catch (T e) // 错误
    {
        Logger.global.info(...);
    }
}
```

5.9 可以取消对检查型异常的检查

可以利用泛型取消对检查型异常的检查，关键在于以下方法：

```
@SuppressWarnings("unchecked")
static <T extends Throwable> void throwAs(Throwable t) throws T
{
    throw (T) t;
}
```

假设这个方法包含在接口`Task`中，如果有一个检查型异常`e`，并调用`Task.<RuntimeException>throwsAs(e);`，编译器就会认为`e`是一个非检查型异常。以下代码会把所有异常都转换为非检查型异常：

```
try
{
    ...
}
catch (Throwable t)
{
    Task.<RuntimeException>throwAs(t);
}
```

下面使用这个技术解决一个问题。要在一个线程中运行代码，需要把代码放在一个实现了`Runnable`接口的类的`run`方法中，不过这个方法不允许抛出检查型异常。下面提供一个从`Task`到`Runnable`的适配器，它的`run`方法可以抛出任意异常：

```
interface Task
{
    void run() throws Exception;

    @SuppressWarnings("unchecked")
    static <T extends Throwable> void throwAs(Throwable t) throws T
    {
        throw (T) t;
    }

    static Runnable asRunnable(Task task)
    {
        return () ->
        {
            try
            {
                task.run();
            }
            catch (Exception e)
            {
                Task.<RuntimeException>throwAs(e);
            }
        };
    }
}
```

`Task`接口的用法如下所示：

```
public class Test
{
    public static void main(String[] args)
    {
        Thread thread = new Thread(Task.asRunnable(() ->
            {
                Thread.sleep(1000);
                System.out.println("Hello World");
                throw new Exception("Check this out!");
            }));
        thread.start();
    }
}
```

正常情况下，必须捕获`Runnable`接口的`run`方法中的所有检查型异常，把它们包装到非检查型异常中，因为`run`方法声明为不抛出任何检查型异常。不过在这里并没有做这种包装，只是抛出异常，并将它转换成非检查型异常。

5.10 注意擦除后的冲突

当泛型类型被擦除后，不允许创建引发冲突的条件。例如：

```
public class Pair<T>
{
    public boolean equals(T value)
    {
        return first.equals(value) && second.equals(value);
    }
    ...
}
```

考虑`Pair<String>`，它有两个`equals`方法：

```
boolean equals(String) // 在 Pair<T> 中定义
boolean equals(Object) // 从 Object 类继承
```

方法`boolean equals(T)`擦除后就是`boolean equals(Object)`，这会与`Object.equals`方法发生冲突。解决的办法是重新命名引发冲突的方法。

泛型规范说明还引用了另外一个原则：如果两个接口类型是同一接口的不同参数化，一个类或类型变量就不能同时作为这两个接口类型的子类。例如，下面的代码是非法的：

```
class Employee implements Comparable<Employee> { ... }
class Manager extends Employee implements Comparable<Manager> { ... } // 错误
```

`Manager`类会实现`Comparable<Employee>`和`Comparable<Manager>`，这是同一接口的不同参数化。

6 泛型类型的继承规则

无论S和T有什么关系，通常，`Pair<S>`与`Pair<T>`都没有任何关系。例如，`Manager`类是`Employee`类的子类，但`Pair<Employee>`和`Pair<Manager>`没有继承关系，下面的赋值操作是不合法的：

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies; // 不合法
```

总是可以将参数化类型转换为原始类型。例如，`Pair<Employee>`是原始类型`Pair`的子类，可以将`Pair<Employee>`类型转换为`Pair`类型：

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair rawBuddies = managerBuddies; // 允许转换
```

泛型类可以扩展或实现其他的泛型类。就这一点而言，它们与普通的类没有什么区别。例如，`ArrayList<T>`类实现了`List<T>`接口，这意味着`ArrayList<Manager>`实现了`List<Manager>`，`ArrayList<Employee>`实现了`List<Employee>`，但是`ArrayList<Manager>`与`ArrayList<Employee>`和`List<Employee>`没有关系。

7 通配符类型

7.1 通配符概念

在通配符类型中，允许类型参数发生变化。例如，通配符类型`Pair<? extends Employee>`表示任何泛型`Pair`类型，它的类型参数是`Employee`类的子类，这种情况称为子类型限定。

假设有如下方法：

```
public static void printBuddies(Pair<Employee> p)
{
    Employee first = p.getFirst();
    Employee second = p.getSecond();
    System.out.println(first.getName() + " and " + second.getName() + " are
buddies.");
}
```

不能将`Pair<Manager>`传递给这个方法，这一点很有限制。可以使用通配符类型解决这一问题，提高这个方法的通用性：

```
public static void printBuddies(Pair<? extends Employee> p)
```

`Pair<? extends Employee>`是原始类型`Pair`的子类，`Pair<Employee>`和`Pair<Manager>`又是`Pair<? extends Employee>`的子类。因此可以向方法传递`Pair<Employee>`或`Pair<Manager>`对象。

考虑下面的操作：

```
Employee lowlyEmployee = ...;
Pair<Manager> managerBuddies = ...;
Pair<? extends Employee> wildcardBuddies = managerBuddies; // OK
wildcardBuddies.setFirst(lowlyEmployee); // 编译错误
```

`Pair<? extends Employee>`的方法如下：

```
? extends Employee getFirst()
void setFirst(? extends Employee)
// 这种写法并不符合 Java 的语法规范，只是为了表达方便。下同
```

对于`setFirst`方法，它的参数类型为`? extends Employee`，编译器只知道需要`Employee`的某个子类型，但不知道具体是什么类型，无法确定方法调用的类型兼容性。例如，当`? extends Employee`为`Manager`时，可以向方法传递`Manager`对象；但当`? extends Employee`为`Employee`时，就不允许传递`Manager`对象。因此，不允许向这个方法传递任何特定的类型，也就不能调用`setFirst`方法。`getFirst`方法就不存在这个问题，将它的返回值赋给一个`Employee`引用一定是合法的。

这个例子说明，通过子类型限定的通配符可以区分安全的访问器方法和不安全的更改器方法，不能向更改器方法传递参数，而访问器方法可以正常使用。

7.2 通配符的超类型限定

通配符限定与类型变量限定十分类似，但是它还有一个附加能力，即可以指定超类型限定。例如，`Pair<? super Manager>`表示类型参数是`Manager`类的超类。

带有超类型限定的通配符的行为与子类型限定的相反，可以为方法提供参数，但不能使用返回值。例如，`Pair<? super Manager>`有如下方法：

```
void setFirst(? super Manager)
? super Manager getFirst()
```

编译器无法知道`setFirst`方法的具体参数类型，因此不能接受`Manager`的子类作为参数，只能传递`Manager`类及其子类对象。对于`getFirst`方法，不能保证返回值的类型，只能把它赋给`Object`变量。

直观地讲，带有超类型限定的通配符允许写入一个泛型对象，而带有子类型限定的通配符允许读取一个泛型对象。

下面是一个典型的例子，把经理数组中奖金最高和最低的经理放在一个`Pair`对象中：

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
    if (a.length == 0) return;
    Manager min = a[0];
    Manager max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (min.getBonus() > a[i].getBonus()) min = a[i];
        if (max.getBonus() < a[i].getBonus()) max = a[i];
    }
    result.setFirst(min);
    result.setSecond(max);
}
```

`Pair<? super Manager>`是原始类型`Pair`的子类，`Pair<Employee>`和`Pair<Object>`又是`Pair<? super Manager>`的子类。因此，这个方法可以接收`Pair<Employee>`或`Pair<Object>`对象。

下面是超类型限定的另一种应用。`Comparable`接口本身就是泛型类型，声明如下：

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

在第3节中曾定义过这样一个方法：

```
public static <T extends Comparable> T min(T[] a)
```

由于`Comparable`是一个泛型类型，因此可以优化这个方法的类型限定：

```
public static <T extends Comparable<T>> T min(T[] a)
```

对于许多类来说，这样写工作得更好。例如，当T是`String`类型时，`String`是`Comparable<String>`的子类型。但是，处理一个`LocalDate`数组时会遇到问题。`LocalDate`实现了`ChronoLocalDate`，而`ChronoLocalDate`扩展了`Comparable<ChronoLocalDate>`，因此`LocalDate`实现的是`Comparable<ChronoLocalDate>`而不是`Comparable<LocalDate>`。在这种情况下，可以利用超类型限定来解决：

```
public static <T extends Comparable<? super T>> T min(T[] a)
```

超类型限定的另一种常见的用法是作为函数式接口的参数类型。例如，`Collection`接口有一个方法：

```
default boolean removeIf(Predicate<? super E> filter)
```

7.3 无限定通配符

还可以使用无限定的通配符，例如`Pair<?>`，它有以下方法：

```
? getFirst()
void setFirst(?)
```

`getFirst`的返回值只能赋给`Object`变量，`setFirst`方法不能被调用。`Pair<?>`和`Pair`本质的不同在于：可以用任意`Object`对象调用原始`Pair`类的`setFirst`方法。

无限定的通配符对于很多简单操作非常有用。例如，下面这个方法可用来测试一个`Pair`对象是否包含`null`引用：

```
public static boolean hasNulls(Pair<?> p)
{
    return p.getFirst() == null || p.getSecond() == null;
}
```

这个方法也可以不使用通配符类型：

```
public static <T> boolean hasNulls(Pair<T> p)
```

但是带有通配符的版本可读性更好。

7.4 通配符捕获

通配符不是类型变量，因此不能在编写代码中使用`?`作为一种类型。例如，下面的代码是非法的：

```
public static void swap(Pair<?> p)
{
    ? t = p.getFirst(); // 错误
    p.setFirst(p.getSecond()); // 无法调用
    p.setSecond(t); // 无法调用
}
```

可以写一个辅助方法`swapHelper`来解决这个问题：

```
public static <T> void swapHelper(Pair<T> p)
{
    T t = p.getFirst();
    p.setFirst(p.getSecond());
    p.setSecond(t);
}

public static void swap(Pair<?> p)
{
    swapHelper(p);
}
```

`swapHelper`是泛型方法，而`swap`不是，它有一个固定的`Pair<?>`类型的参数。在这种情况下，`swapHelper`方法的类型参数`T`捕获通配符，它不知道通配符指示哪种类型，但是这是一个明确的类型。

通配符捕获只有在非常限定的情况下才是合法的，编译器必须能够保证通配符表示单个确定的类型。例如，`ArrayList<Pair<T>>`中的T不能捕获`ArrayList<Pair<?>>`中的通配符，因为数组列表可以保存多个`Pair<?>`，其中的`?`可能有不同的类型。

下面的测试程序将通配符的各种用法综合在一起：

```
public class PairTest
{
    public static void main(String[] args)
    {
        Manager ceo = new Manager("Gus", 800000, 2003, 12, 15);
        Manager cfo = new Manager("Sid", 600000, 2003, 12, 15);
        Pair<Manager> buddies = new Pair<Manager>(ceo, cfo);
        printBuddies(buddies);

        ceo.setBonus(1000000);
        cfo.setBonus(500000);
        Manager[] managers = { ceo, cfo };

        Pair<Employee> result = new Pair<Employee>();
        minmaxBonus(managers, result);
        System.out.println("first: " + result.getFirst().getName()
            + ", second: " + result.getSecond().getName());
        maxminBonus(managers, result);
        System.out.println("first: " + result.getFirst().getName()
            + ", second: " + result.getSecond().getName());
    }

    public static void printBuddies(Pair<? extends Employee> p)
    {
        Employee first = p.getFirst();
        Employee second = p.getSecond();
        System.out.println(first.getName() + " and " + second.getName() + " are
buddies.");
    }

    public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
    {
        if (a.length == 0) return;
        Manager min = a[0];
```

```
        Manager max = a[0];
        for (int i = 1; i < length; i++)
        {
            if (min.getBonus() > a[i].getBonus()) min = a[i];
            if (max.getBonus() < a[i].getBonus()) max = a[i];
        }
        result.setFirst(min);
        result.setSecond(max);
    }

    public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
    {
        minmaxBonus(a, result);
        PairAlg.swapHelper(result); // 通配符捕获
    }
}

class PairAlg
{
    public static boolean hasNulls(Pair<?> p)
    {
        return p.getFirst() == null || p.getSecond() == null;
    }

    public static <T> void swapHelper(Pair<T> p)
    {
        T t p.getFirst();
        p.setFirst(p.getSecond());
        p.setSecond(t);
    }

    public static void swap(Pair<?> p)
    {
        swapHelper(p);
    }
}
```

8 反射和泛型

8.1 泛型Class类

`Class`类是泛型类。例如，`String.class`实际上是一个`Class<String>`类的对象。

类型参数十分有用，它允许`Class<T>`方法的返回类型更加具有特定性。`Class<T>`的以下方法就使用了类型参数：

```
/* java.lang.Class<T> */
T newInstance()
    // 返回无参数构造器构造的一个新实例
T cast(Object obj)
    // 如果 obj 为 null 或有可能转换成类型 T，则返回 obj；否则抛出一个 BadCastException
异常
T[] getEnumConstants()
    // 如果 T 是枚举类型，则返回所有值组成的数组，否则返回 null
Class<? super T> getSuperclass()
    // 返回这个类的超类。如果 T 表示 Object 类、接口、基本类型或 void，则返回 null
Constructor<T> getConstructor(Class<?>... parameterTypes)
    // 获得有给定参数类型的公共构造器
Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
    // 获得有给定参数类型的构造器
```

8.2 使用Class参数进行类型匹配

匹配泛型方法中`Class<T>`参数的类型变量有时会很有用。下面是一个标准的示例：

```
public static <T> Pair<T> makePair(Class<T> c)
        throws InstantiationException, IllegalAccessException
{
    return new Pair<>(c.newInstance(), c.newInstance());
}
```

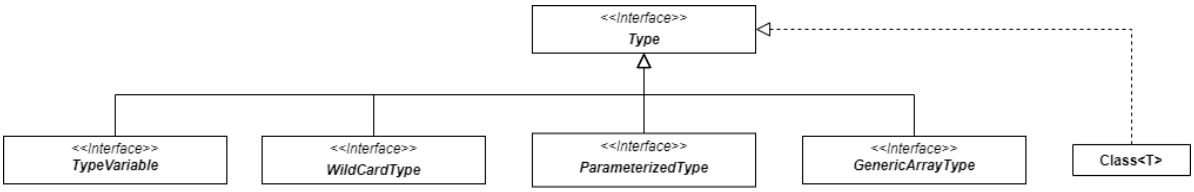
如果调用`makePair(Employee.class)`，`Employee.class`将是一个`Class<Employee>`类型的对象，`makePair`方法的类型参数`T`与`Employee`匹配，编译器可以推断出这个方法将返回一个`Pair<Employee>`。

8.3 虚拟机中的泛型类型信息

为了表述泛型类型声明，可以使用`java.lang.reflect`包中的接口`Type`，这个接口包含以下子类型：

- `Class<T>`类，描述具体类型。
- `TypeVariable`接口，描述类型变量。
- `WildcardType`接口，描述通配符。
- `ParameterizedType`接口，描述参数化类型。
- `GenericArrayType`接口，描述泛型数组。

下图给出了这些类和接口的继承层次：



下面的程序使用泛型反射API打印出给定类的有关内容：

```
import java.lang.reflect.*;
import java.util.*;

public class GenericReflectionTest
{
    public static void main(String[] args)
    {
```

```
        // 从命令行或用户输入读取类名
        String name;
        if (args.length > 0)
        {
            name = args[0];
        }
        else
        {
            try (Scanner in = new Scanner(System.in))
            {
                System.out.println("Enter class name (e.g.,
java.util.Collections): ");
                name = in.next();
            }
        }

        try
        {
            // 输出类及其方法的泛型信息
            Class<?> cl = Class.forName(name);
            printClass(cl);
            for (Method m : cl.getDeclaredMethods())
            {
                printMethod(m);
            }
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }
    }

    public static void printType(Type type, boolean isDefinition)
    {
        if (type instanceof Class)
        {
            Class<?> t = (Class<?>) type;
            System.out.print(t.getName());
        }
        else if (type instanceof TypeVariable)
        {
            TypeVariable<?> t = (TypeVariable<?>) type;
            System.out.print(t.getName());
            if (isDefinition)
            {
                printTypes(t.getBounds(), " extends ", " & ", "", false);
            }
        }
        else if (type instanceof WildcardType)
        {
            WildcardType t = (WildcardType) type;
            System.out.print("?");
            printTypes(t.getUpperBounds(), " extends ", " & ", "", false);
            printTypes(t.getLowerBounds(), " super ", " & ", "", false);
        }
        else if (type instanceof ParameterizedType)
        {
            ParameterizedType t = (ParameterizedType) type;
```

```
            Type owner = t.getOwnerType();
            if (owner != null)
            {
                printType(owner, false);
                System.out.print(".");
            }
            printType(t.getRawType(), false);
            printTypes(t.getActualTypeArguments(), "<", ", ", ">", false);
        }
        else if (type instanceof GenericArrayType)
        {
            GenericArrayType t = (GenericArrayType) type;
            System.out.print("");
            printType(t.getGenericComponentType(), isDefinition);
            System.out.print("[]");
        }
    }

    public static void printTypes(Type[] types, String pre, String sep, String
suf,
            boolean isDefinition)
    {
        if (pre.equals(" extends ") && Arrays.equals(types, new Type[] {
Object.class }))
        {
            return;
        }
        if (types.length > 0) System.out.print(pre);
        for (int i = 0; i < types.length; i++)
        {
            if (i > 0) System.out.print(sep);
            printType(types[i], isDefinition);
        }
        if (types.length > 0) System.out.print(suf);
    }

    public static void printClass(Class<?> cl)
    {
        System.out.print(cl);
        printTypes(cl.getTypeParameters(), "<", ", ", ">", true);
        Type sc = cl.getGenericSuperclass();
        if (sc != null)
        {
            System.out.print(" extends ");
            printType(sc, false);
        }
        printTypes(cl.getGenericInterfaces(), " implements ", ", ", "", false);
        System.out.println();
    }

    public static void printMethod(Method m)
    {
        String name = m.getName();
        System.out.print(Modifier.toString(m.getModifiers()));
        System.out.print(" ");
        printTypes(m.getTypeParameters(), "<", ", ", ">", true);

        printType(m.getGenericReturnType(), false);
```

```
            System.out.print(" ");
            System.out.print(name);
            System.out.print("(");
            printTypes(m.getGenericParameterTypes(), "", ", ", "", false);
            System.out.println(")");
        }
    }
```

其中涉及的方法如下：

```
/* java.lang.Class<T> */
TypeVariable[] getTypeParameters()
    // 如果这个类型被声明为泛型类型，则获得泛型类型变量，否则返回长度为 0 的数组
Type getGenericSuperclass()
    // 获得这个类型的直接超类。如果超类是参数化类型，则返回的 Type 对象必须准确反映源代码中使用的实际类型参数
    // 如果这个对象表示 Object 类、接口、基本类型或 void，则返回 null。如果这个对象表示数组类，则返回表示 Object 类的 Class 对象
Type[] getGenericInterfaces()
    // 获得这个类型所实现的接口类型的数组，数组中对象的顺序与类定义时声明的顺序一致
    // 如果这个对象表示一个类，则返回这个类所实现的接口类型的数组
    // 如果这个对象表示一个接口，则返回这个接口所扩展的接口类型的数组
    // 如果这个对象所表示的类或接口没有扩展任何接口，或者这个对象表示基本类型或 void，则返回长度为 0 的数组

/* java.lang.reflect.Method */
TypeVariable[] getTypeParameters()
    // 如果这个方法被声明为泛型方法，则获得泛型类型变量，否则返回长度为 0 的数组
Type getGenericReturnType()
    // 获得这个方法的泛型返回类型
Type[] getGenericParameterTypes()
    // 获得这个方法的泛型参数类型。如果这个方法没有参数，则返回长度为 0 的数组

/* java.lang.reflect.TypeVariable */
String getName()
    // 获得这个类型变量的名字
Type[] getBounds()
    // 获得这个类型变量的子类限定。如果该变量无限定，则返回长度为 0 的数组

/* java.lang.reflect.WildcardType */
Type[] getUpperBounds()
    // 获得这个通配符的子类限定。如果没有子类限定，则返回长度为 0 的数组
Type[] getLowerBounds()
    // 获得这个通配符的超类限定。如果没有超类限定，则返回长度为 0 的数组

/* java.lang.reflect.ParameterizedType */
Type getRawType()
    // 获得这个参数化类型的原始类型
Type[] getActualTypeArguments()
    // 获得这个参数化类型声明的类型参数
Type getOwnerType()
    // 如果是内部类型，则返回其外部类类型；如果是一个顶级类型，则返回 null

/* java.lang.reflect.GenericArrayType */
Type getGenericComponentType()
    // 获得这个数组类型声明的泛型元素类型
```

8.4 类型字面量

有时，你会希望由值的类型决定程序的行为，通常的实现方法是将`Class`对象与一个动作关联。不过如果有泛型类，擦除会带来问题。例如，`ArrayList<Integer>`和`ArrayList<String>`都擦除为`ArrayList`，无法区分它们。

下面的技巧可以在某些情况下解决这个问题。定义一个`TypeLiteral<T>`类：

```
class TypeLiteral<T>
{
    private Type type;

    public TypeLiteral()
    {
        Type parentType = getClass().getGenericSuperclass();
        if (parentType instanceof ParameterizedType)
        {
            type = ((ParameterizedType) parentType).getActualTypeArguments()[0];
        }
        else
        {
            throw new UnsupportedOperationException(
                "Construct as new TypeLiteral<...>(){}");
        }
    }
}
```

这个类的构造器会捕获泛型超类型，如果运行时有一个泛型类型，就可以将它与`TypeLiteral`匹配。我们无法从一个对象得到泛型类型（已经被擦除），不过字段和方法的泛型类型还留存在虚拟机中。捕获泛型类型之后，构造一个匿名子类，例如：

```
var type = new TypeLiteral<ArrayList<Integer>>(){};
```

下面的程序展示了这个技巧的用法。给定一个对象，我们可以罗列它的字段，找出泛型类型的字段并查找相关联的格式化动作。

```
import java.lang.reflect.*;
import java.util.*;
import java.util.function.*;

/**
 * 类型字面量描述一个可能是泛型的类型，例如 ArrayList<String>
 */
class TypeLiteral<T>
{
    private Type type;

    public TypeLiteral()
    {
        Type parentType = getClass().getGenericSuperclass();
        if (parentType instanceof ParameterizedType)
        {
            type = ((ParameterizedType) parentType).getActualTypeArguments()[0];
        }
        else
```

```
        {
            throw new UnsupportedOperationException(
                "Construct as new TypeLiteral<...>(){}");
        }
    }

    private TypeLiteral(Type type)
    {
        this.type = type;
    }

    /**
     * 生成描述给定类型的类型字面量
     */
    public static TypeLiteral<?> of(Type type)
    {
        return new TypeLiteral<Object>(type);
    }

    public String toString()
    {
        if (type instanceof Class)
            return ((Class<?>) type).getName();
        else
            return type.toString();
    }

    public boolean equals(Object otherObject)
    {
        return otherObject instanceof TypeLiteral
            && type.equals(((TypeLiteral<?>) otherObject).type);
    }

    public int hashCode()
    {
        return type.hashCode();
    }
}

/**
 * 格式化对象，使用将类型与格式化函数关联的规则
 */
class Formatter
{
    private Map<TypeLiteral<?>, Function<?, String>> rules = new HashMap<>();

    /**
     * 向格式化器添加格式化规则
     * @param type 应用此规则的类型
     * @param formatterForType 对这一类型的对象进行格式化的函数
     */
    public <T> void forType(TypeLiteral<T> type, Function<T, String>
formaterForType)
    {
        rules.put(type, formatterForType);
    }

    /**
```

```
     * 用这个格式化器的规则对一个对象中的所有字段进行格式化
     * @param obj 一个对象
     * @return 一个包含所有字段的名字及其格式化值的字符串
     */
    public String formatFields(Object obj)
            throws IllegalArgumentException, IllegalAccessException
    {
        StringBuilder result = new StringBuilder();
        for (Field f : obj.getClass().getDeclaredFields())
        {
            result.append(f.getName());
            result.append("=");
            f.setAccessible(true);
            Function<?, String> formatterForType =
rules.get(TypeLiteral.of(f.getGenericType()));
            if (formatterForType != null)
            {
                // formatterForType 有类型参数 ?，不能调用 apply 方法
                // 将类型参数转换为 Object，就可以正常调用方法了
                @SuppressWarnings("unchecked")
                Function<Object, String> objectFormatter
                    = (Function<Oject, String>) formatterForType;
                result.append(objectFormatter.apply(f.get(obj)));
            }
            else
            {
                result.append(f.get(obj).toString());
            }
            result.append("\n");
        }
        return result.toString();
    }
}

public class TypeLiterals
{
    public static class Sample
    {
        ArrayList<Integer> nums;
        ArrayList<Character> chars;
        ArrayList<String> strings;

        public Sample()
        {
            nums = new ArrayList<>();
            nums.add(42);
            nums.add(1729);
            chars = new ArrayList<>();
            char.add('H');
            char.add('i');
            strings = new ArrayList<>();
            strings.add("Hello");
            strings.add("World");
        }
    }

    private static <T> String join(String separator, ArrayList<T> elements)
    {
```

```
        StringBuilder result = new StringBuilder();
        for (T t : elements)
        {
            if (result.length() > 0) result.append(separator);
            result.append(e.toString());
        }
        return result.toString();
    }

    public static void main(String[] args)
    {
        Formatter formatter = new Formatter();
        formatter.forType(new TypeLiteral<ArrayList<Integer>>(){},
            lst -> join(" ", lst));
        formatter.forType(new TypeLiteral<ArrayList<Character>>(){},
            lst -> "\"" + join("", lst) + "\"");
        System.out.println(formatter.formatFields(new Sample()));
    }
}
```