第四章

类型、值和变量

Java 编程语言是静态类型的语言，意思是每个变量和表达式的类型在编译时就是确定的。

Java 也是一种强类型的语言，因为类型限制了变量(§4.12)所能容纳的值或表达式所能产生的值，限制了对这些值支持的操作，并决定了操作的含义。强静态类型有助于在编译时检测错误。

Java语言分为两种类型：原生类型和引用类型。原生类型(§4.2)包括布尔类型和数值类型。数值类型包括整数类型 byte,short,int,long 和 char，以及浮点类型 float 和 double。引用类型(§4.3)包括类类型，接口类型和数组类型。还有一个特殊的空类型。对象(§4.3.1)是一个动态创建的类类型或动态创建的数组实例。引用类型的值是对象的引用。所有对象，包括数组，支持 Object 类(§4.3.2)的方法。字符串字面量由 String 对象(§4.3.3)表示。

4.1 类型和值的种类

Java 语言分为两种类型：原生类型(§4.2)和引用类型(§4.3)。相应地，有两种数据值可以存储在变量中，作为参数传递，由方法返回，并进行操作:原始值(§4.2)和引用值(§4.3)。

Type:
PrimitiveType
ReferenceType

还有一种特殊的空类型，即 null 表达式的类型(§3.10.8，§15.8.1)，它没有名称。

因为 null 类型没有名称，所以不可能声明 null 类型的变量或强制转换为 null 类型。

null 引用是 null 类型表达式唯一可能的值。

null 引用总是可以被赋值或强制转换为任何引用类型(§5.2、§5.3、§5.5)。

> 在实践中，程序员可以忽略 null 类型，只是把 null 当做一个特殊的字面量，它可以是任何引用类型。

4.2 原生类型和值

原生类型由 Java 编程语言预先定义，并由其保留关键字(§3.9)命名:

PrimitiveType:
  {Annotation} NumericType
  {Annotation} `boolean`

NumericType:
  IntegralType
  FloatingPointType

IntegralType:
  (one of)
  `byte short int long char`

FloatingPointType:
  (one of)
  `float double`

原生类型之间并不共享状态。

数值类型包括整数类型和浮点类型。

整型包括 byte、short、int 和 long，其值分别为 8 位、16 位、32 位和 64 位有符号的二进制补码，还有 char，其值为 16 位无符号整数，表示 UTF-16 代码单元(§3.1)。

浮点类型有 float 和 double，前者精确对应 32 位 IEEE 754 binary32 浮点数，后者精确对应 64 位 IEEE 754 binary64 浮点数。

布尔类型包含两个值：true 和 false。

4.2.1 整数类型和值

整数类型的取值范围如下:

- 对于 byte，从-128 到 127，闭区间

- 对于 short，从-32768 到 32767，闭区间
- 对于 int，从-2147483648 到 2147483647，闭区间
- 对于 long，从-9223372036854775808 到 9223372036854775807 闭区间
- 对于 char，从'\u0000'到'\uffff'闭区间，也就是从 0 到 65535

4.2.2 整数操作

Java 编程语言提供了许多作用于整数值的操作符：

- 比较操作符，其结果为 boolean 类型的值：
  - 数值比较操作符<、<=、>和>=(§15.20.1)
  - 数值相等操作符==和!=(§15.21.1)
- 数值操作符，产生值的类型为 int 或 long：
  - 一元的加减运算符+和-(§15.15.3，§15.15.4)
  - 乘法运算符*、/和%(§15.17)
  - 加性运算符+和-(§15.18)
  - 自增操作符++，包括前缀(§15.15.1)和后缀(§15.14.2)
  - 自减操作符--，包括前缀(§15.15.2)和后缀(§15.14.3)
  - 有符号和无符号移位操作符 <<, >>和>>> (§15.19)
  - 按位补操作符~(§15.15.5)
  - 整数位操作符&,^和| (§15.22.1)
- 条件运算符?:(§15.25)
- 转换运算符(§15.16)，它可以将整型值转换为任何指定的数值类型
- string 连接运算符+(§15.18.1)，当给定一个 String 操作数和一个整型操作数时，它会将整型操作数转换为 String(byte,short,int 或 long 类型操作数的十进制形式，或 char 类型操作数的字符形式)，然后生成一个由两个字符串连接而成的新 String

其他有用的构造器、方法和常量预先定义在类 Byte, Short, Integer, Long 和 Character 中。

如果除移位操作符以外的整数操作符至少有一个 long 类型的操作数，那么该操作将

使用 64 位精度执行，并且数值操作符的结果是 long 类型的。如果另一个操作数不是 long，则首先通过数字提升(§5.6)将其加宽(§5.1.5)为 long 类型。

否则，该操作将使用 32 位精度执行，并且数值操作符的结果是 int 类型的。如果两个操作数都不是 int 类型，则首先通过数字提升将它们加宽为 int 类型。

整数运算符不以任何方式指示上溢或下溢。

任何整型的任何值都可以转换为任何数值类型，也可以从任何数值类型转换而来。整型和布尔型之间不存在强制转换。

> 将整数表达式转换为布尔表达式的习惯用法见§4.2.5。

整数运算符可以因为以下原因抛出异常 (§11):

- 如果需要对空引用进行拆箱转换(§5.1.8)，任何整数运算符都可以抛出 NullPointerException。
- 如果右操作数为零，则整数除法运算符/(§15.17.2)和整数余数运算符%(§15.17.3)会抛出 ArithmeticException。
- 当需要进行装箱转换(§5.1.7)而又没有足够的内存来执行转换时，递增和递减运算符++(§15.14.2，§15.15.1)和--(§15.14.3，§15.15.2)会抛出 OutOfMemoryError 异常。

例子 4.2.2-1. 整数操作

```
class Test {
   public static void main(String[] args)
       {
       int i = 1000000;
       System.out.println(i * i);
       long l = i;
       System.out.println(l * l);
       System.out.println(20296 / (l - i));
   }
}
```

程序产生如下输出:

```
-727379968
1000000000000
```

在除以 1-i 时会遇到 ArithmeticException，因为 I-i 等于 0。第一次乘法以 32 位精度执行，而第二次乘法操作数是 long 类型。值-727379968 是数学结果的低 32 位的十进制值，1000000000000，这个值对 int 类型来说太大了。

4.2.3 浮点类型和值

浮点类型是 float 和 double，它们在概念上与 IEEE 754 值和操作的 32 位 binary32 和 64 位 binary64 浮点格式相关联，如 IEEE 754 标准(§1.7)所规定的。

> 在 Java SE 15 及以后版本中，Java 编程语言使用 IEEE 754 标准的 2019 版。在 Java SE 15 之前，Java 编程语言使用的是 1985 年版本的 IEEE 754 标准，其中 binary32 格式被称为单精度格式，binary64 格式被称为双精度格式。

IEEE 754 不仅包括由符号和大小组成的正负数字，还包括正零和负零、正无穷和负无穷，以及特殊的非数字值(以下简称 NaN)。NaN 值用于表示某些无效操作的结果，例如 0 除以 0。float 和 double 的 NaN 常量预定义为 Float.NaN 和 Double.NaN。

浮点类型的有限非零值都可以用以下形式表示 $s \cdot m \cdot 2^{(e-N+1)}$，其中:

- s 为 +1 或 -1,
- m 是小于 2^N 的正整数,
- e 是一个在 $E_{min} = -(2^{K-1}-2)$ 和 $E_{max} = 2^{K-1}-1$ 之间的整数，包括上下界,
- N 和 K 是依赖于类型的参数。

在这种形式中，有些值可以用多种方式表示。例如，假设一个浮点类型的值 v 可以用 s、m 和 e 的特定值表示成这种形式，那么如果 m 是偶数，e 小于 2^{K-1}，人们可以将 m 减半，并将 e 增加 1 来产生相同值 v 的第二种表示。

如果 $m \geqslant 2^{N-1}$，这种形式的表示称为规范化；否则，这种表现就被称为不正常的。如果浮点类型的值不能以 $m \geqslant 2^{N-1}$ 的方式表示，则该值被称为不正常的，因为其大小低于最小规范化值的大小。

表 4.2.3-A 总结了 float 和 double 的参数 N 和 K(以及导出参数 E_{min} 和 E_{max})的约束。

表 4.2.3-A. 浮点参数

Parameter	`float`	`double`
N	24	53
K	8	11
E_{max}	+127	+1023
E_{min}	-126	-1022

除 NaN 外，浮点值是有序的。按从小到大排列，它们是负无穷、负有限非零值、负零和正零、正有限非零值和正无穷。

IEEE 754 允许对其每一种 binary 32 和 binary 64 浮点格式使用多个不同的 NaN 值。然而，Java SE 平台通常将给定浮点类型的 NaN 值折叠成单个规范值，因此该规范通常将任意 NaN 视为规范值。

> 在 IEEE 754 下，带有非 NaN 参数的浮点操作可以生成 NaN 结果。IEEE 754 指定了一组 NaN 位模式，但没有规定使用哪一个特定的 NaN 位模式来表示 NaN 结果;这就留给硬件架构了。 程序员可以创建具有不同位模式的 NAN 来编码，例如，回溯性诊断信息。这些 NaN 值可以通过 float 的 Float.intBitsToFloat 方法和 double 的 Double.longBitsToDouble 方法创建。 相反，要检查 NaN 值的位模式，可以使用 float 的 Float.floatToRawIntBits 方法或 double 的 Double.doubleToRawLongBits 方法。

正零和负零相等，因此表达式 0.0==-0.0 的结果为真，而 0.0>-0.0 的结果为假。其他操作可以区分正零和负零；例如，1.0/0.0 的值为正无穷大，而 1.0/-0.0 的值为负无穷大。

NaN 是无序的，因此:

- 如果数值比较操作符<、<=、>和>=有一个或两个操作数都是 NaN，则返回 false(§15.20.1)。

  特别地，如果 x 或 y 为 NaN， (x<y) == !(x>=y)将为假。
- 如果任意一个操作数为 NaN，则相等操作符==返回 false。
- 如果任意一个操作数为 NaN，则不等操作符!=返回 true(§15.21.1)。

特别是 x!=x 为 true 当且仅当 x 是 NaN。

4.2.4 浮点操作

Java 编程语言提供了许多作用于浮点值的操作符:

- 比较操作符，其结果为布尔型值:
  - 数值比较运算符<、<=、>和>=(§15.20.1)
  - 数值相等运算符==和!=(§15.21.1)
- 数值运算符，产生结果类型为 float 或 double:
  - 一元的加减运算符+和-(§15.15.3，§15.15.4)
  - 乘法运算符*、/和%(§15.17)
  - 加性运算符+和-(§15.18.2)
  - 自增操作符++，包括前缀(§15.15.1)和后缀(§15.14.2)

- 自减操作符--，包括前缀(§15.15.2)和后缀(§15.14.3)

- 条件运算符?:(§15.25)
- 转换运算符(§15.16)，它可以将浮点值转换为任何指定的数值类型
- 字符串连接运算符+(§15.18.1)，当给定一个 String 操作数和一个浮点操作数时，将浮点操作数转换为表示其十进制值的String(不丢失信息)，然后通过连接两个字符串生成一个新创建的 String

其他有用的构造函数、方法和常量在类 Float, Double 和 Math 中预定义。

如果二元操作符的操作数中至少有一个是浮点类型，则该操作为浮点操作，即使另一个操作数是整数。

如果数值运算符的操作数中至少有一个是 double 类型的，那么该操作将使用 64 位浮点算术来执行，并且数值运算符的结果是 double 类型的值。如果另一个操作数不是 double 类型，则首先通过数字提升(§5.6)将其加宽(§5.1.5)为 double 类型。

否则，至少有一个操作数是 float 类型;该操作使用 32 位浮点运算来执行，数值运算符的结果是 float 类型的值。如果另一个操作数不是 float，则首先通过数字提升将其扩展为 float 类型。

除了剩余运算符%(§15.17.3)之外，浮点运算按照 IEEE 754 标准的规则进行，包括溢出和下溢(§15.4)，。

浮点类型的任何值都可以转换为任何数值类型，也可以从任何数值类型转换而来。浮点类型和布尔类型之间不存在强制转换。

> 将浮点表达式转换为布尔表达式的用法见§4.2.5。

浮点操作会因为以下原因抛出异常(§11):

- 如果需要对空引用进行拆箱转换(§5.1.8)，任何浮点运算都可以抛出NullPointerException。
- 当需要进行装箱转换(§5.1.7)而又没有足够的内存来执行转换时，递增和递减运算符++(§15.14.2，§15.15.1)和--(§15.14.3，§15.15.2)会抛出 OutOfMemoryError。

例子 4.2.4-1. 浮点操作

```
class Test {
 public static void main(String[] args) {
          // An example of overflow:
          double d = 1e308;
        System.out.print("overflow produces infinity: ");
```

```
        System.out.println(d + "*10==" + d*10);
        // An example of gradual underflow: d = 1e-305 * Math.PI;
        System.out.print("gradual underflow: " + d + "\n    ");
        for (int i = 0; i < 4; i++)
        System.out.print(" " + (d /= 100000));
        System.out.println();
        // An example of NaN:
        System.out.print("0.0/0.0 is Not-a-Number: ");
        d = 0.0/0.0;
        System.out.println(d);
        // An example of inexact results and rounding:
        System.out.print("inexact results with float:");
        for (int i = 0; i < 100; i++) {
            float z = 1.0f / i;
        if (z * i != 1.0f) System.out.print(" " + i); }
        System.out.println();
        // Another example of inexact results and rounding:
        System.out.print("inexact results with double:");
        for (int i = 0; i < 100; i++) {
                  double z = 1.0 / i;
                  if (z * i != 1.0)
              System.out.print(" "+ i); }
        System.out.println();
        // An example of cast to integer rounding:
        System.out.print("cast to int rounds toward 0: ");
        d = 12345.6;
        System.out.println((int)d + " " + (int)(-d));
        }
    }
```

程序产生如下输出:

```
overflow produces infinity: 1.0E308*10==Infinity gradual underflow:
3.141592653589793E-305
    3.1415926535898E-310 3.141592653E-315 3.142E-320 0.0
                0.0/0.0 is Not-a-Number: NaN
                inexact results with float: 0 41 47 55 61 82 83 94 97
                inexact results with double: 0 49 98
                cast to int rounds toward 0: 12345 -12345
```

这个示例说明，除其他事情以外，逐渐的下溢可能会导致精度的逐渐丧失。

当 i 为 0 时，结果包括除以 0，因此 z 变成正无穷，z * 0 是 NaN，不等于 1.0。

4.2.5 布尔类型和布尔值

布尔类型表示一个逻辑量，有两个可能的值，分别用字面量 true 和 false 表示(§3.10.3)。

布尔运算符包括:

- 关系运算符==和!=(§15.21.2)

- 逻辑非操作符!(§15.15.6)
- 逻辑操作符&,^和 |(§15.22.2)
- 条件与和条件或操作符 && (§15.23) 和|| (§15.24)
- 条件操作符 ? : (§15.25)
- 字符串连接运算符+(§15.18.1)，当给定一个 String 操作数和一个布尔操作数时，将布尔操作数转换为 String (true 或 false)，然后生成一个由两个字符串连接而成的新 String

布尔表达式在几种语句中决定控制流:

- if 语句(§14.9)
- while 语句(§14.12)
- do 语句(§14.13)
- for 语句(§14.14)

布尔表达式还确定了哪个子表达式在条件?:操作符(§15.25)中被计算。

只有 boolean 和 Boolean 表达式可以用于控制流语句和作为条件操作符?: 的第一个操作数。

整数或浮点表达式 x 可以通过表达式 x!=0 转换为布尔值，按照 C 语言的约定，任何非零值都为 true。

对象引用 obj 可以通过表达式 obj!=null 转换为布尔值，按照 C 语言约定，除 null 以外的任何引用都为 true。

布尔值可以通过字符串转换(§5.4)转换为 String 类型。

布尔值可以转换为 boolean、Boolean 或 Object(§5.5)。对布尔类型进行其他类型转换是不允许的。

4.3 引用类型和值

有四种引用类型:类类型(§8.1)、接口类型(§9.1)、类型变量(§4.4)和数组类型(§10.1)。

ReferenceType:
ClassOrInterfaceType
TypeVariable
ArrayType

ClassOrInterfaceType:
ClassType
InterfaceType

ClassType:
{Annotation} TypeIdentifier [TypeArguments]
PackageName . {Annotation} TypeIdentifier [TypeArguments]
ClassOrInterfaceType . {Annotation} TypeIdentifier [TypeArguments]

InterfaceType:
ClassType

TypeVariable:
{Annotation} TypeIdentifier

ArrayType:
PrimitiveType Dims
ClassOrInterfaceType Dims
TypeVariable Dims
Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

样例代码:

```
class Point { int[] metrics; }
interface Move { void move(int deltax, int deltay); }
```

声明类类型 Point，接口类型 Move，并使用数组类型 int[] (int 数组)来声明类 Point 的字段度量。

类或接口类型由一个标识符或一个由标识符组成的点序列组成，其中每个标识符后面都有可选类型参数(§4.5.1)。如果类型参数出现在类或接口类型的任何地方，它就是参数化类型(§4.5)。

类或接口类型中的每个标识符都被分类为包名或类型名(§6.5.1)。分类为类型名的标识符可以被注释。如果类或接口类型具有 T.id 形式(可选地后跟类型参数)，那么 id 必须是可访问成员类型 T 的简单名称(§6.6、§8.5、§9.5)，否则将发生编译时错误。类或接口类型表示该成员类型。

4.3.1 对象

对象是类实例或数组。

引用值(通常只是引用)是指向这些对象的指针；一个特殊的空引用，它不指向任何对象。

类实例是由类实例创建表达式显式创建的(§15.9)。

数组是通过数组创建表达式显式创建的(§15.10.1)。

其他表达式可以隐式创建一个类实例(§12.5)或一个数组(§10.6)。

例子 4.3.1-1. 创建对象

```
class Point {
     int x, y;
     Point() { System.out.println("default"); }
     Point(int x, int y) { this.x = x; this.y = y; }

     /* A Point instance is explicitly created
     at class initialization time: */
     static Point origin = new Point(0,0);

     /* A String can be implicitly
     created by a + operator: */
     public String toString() { return "(" + x + "," + y + ")"; }
   }

class Test {
       public static void main(String[] args) {
          /* A Point is explicitly created
             using newInstance: */
          Point p = null;
          try {
              p = (Point)Class.forName("Point").newInstance();
          } catch (Exception e) {
              System.out.println(e);
          }

          /* An array is implicitly created
          by an array initializer: */
          Point[] a = { new Point(0,0), new Point(1,1) };

          /* Strings are implicitly created by + operators: */
          System.out.println("p: " + p);
          System.out.println("a: { " + a[0] + ", " + a[1] + " }");

          /* An array is explicitly created by an array creation
          expression: */
          String[] sa = new String[2];
          sa[0] = "he"; sa[1] = "llo";
          System.out.println(sa[0] + sa[1]);
          }
       }
```

程序产生如下输出:

```
default
p: (0,0)
```

```
a: { (0,0), (1,1) }
hello
```

对象引用的操作符为：

- 字段访问，使用限定名(§6.6)或字段访问表达式(§15.11)
- 方法调用(§15.12)
- 转换操作符(§5.5, §15.16)
- string 连接操作符+(§15.18.1)，当给定 String 操作数和引用时，将通过调用被引用对象的 toString 方法(如果引用或 toString 的结果是空引用，则使用“null”)将引用转换为 String，然后产生一个新创建的 String，这是两个字符串的连接
- Instanceof 操作符(§15.20.2)
- 引用相等操作符==和!=(§15.21.3)
- 条件操作符? : (§15.25)

同一个对象可能有许多引用。大多数对象都有状态，存储在作为类实例的对象的字段中，或者存储在作为数组对象组件的变量中。 如果两个变量包含对同一个对象的引用，则可以使用一个变量对该对象的引用来修改该对象的状态，然后通过对另一个变量的引用来观察被改变的状态。

例子 4.3.1-2. 原生和引用一致性

```
class Value { int val; }

class Test {
      public static void main(String[] args) {
         int i1 = 3;
         int i2 = i1;
         i2 = 4;
         System.out.print("i1==" + i1);
         System.out.println(" but i2==" + i2);
         Value v1 = new Value();
         v1.val = 5;
         Value v2 = v1;
         v2.val = 6;
         System.out.print("v1.val==" + v1.val);
         System.out.println(" and v2.val==" + v2.val);
       }
      }
```

程序产生如下输出：

```
i1==3 but i2==4
v1.val==6 and v2.val==6
```

因为 i1 和 i2 是不同的变量，但 v1.val 和 v2.val 引用相同的 Value 实例变量 (§4.12.3)，这个实例变量是通过唯一的 new 表达式创建的。

每个对象都与一个监视器(§17.1)相关联，监视器由同步方法(§8.4.3)和同步语句(§14.19)来提供对多个线程状态的并发访问控制(§17)。

4.3.2 Object 类

Object 类是所有其他类的超类(§8.1.4)。

所有的类和数组类型继承了(§8.4.8)Object 类的方法，总结如下:

- clone 方法用于创建对象的副本。
- equals 方法定义了对象相等的概念，它基于值而不是引用的比较。
- finalize 方法在对象销毁之前运行(§12.6)。
- getClass 方法返回表示对象所属类的 Class 对象。每个引用类型都有一个 Class 对象。可以使用它来发现类的完全限定名、它的成员、它的直接超类以及它实现的任何接口。

  getClass 的方法调用表达式的类型是 Class<? extends |T|>，其中 T 是 getClass 寻找的类或接口，|T|表示 T 的类型擦除(§4.6)。

  一个被声明为 synchronized 的类方法在与该类的 class 对象相关联的监视器上同步。

- hashCode 方法和 equals 方法在诸如 java.util.HashMap 这样的散列表中非常有用。
- 方法 wait, notify 和 notifyAll 在线程的并发编程中使用(§17.2)。
- toString 方法返回对象的 String 表示形式。

4.3.3 String 类

String 类的实例表示 Unicode 码位序列。

String 对象有一个常量(不变的)值。

String literals (§3.10.5) and text blocks (§3.10.6) are references to instances of class `String`.

当结果不是常量表达式时，string 连接操作符+(§15.18.1)隐式创建一个新的 String 对象(§15.29)。

4.3.4 引用类型何时相等

如果两个引用类型在与相同模块相关联的编译单元中声明(§7.3)，并且它们具有相同的二进制名称(§13.1)，并且它们的类型参数(如果有的话)是相同的，则它们在编译时的类型是相同的，递归地应用这个定义。

当两个引用类型相同时，它们有时被称为相同的类或相同的接口。

在运行时，不同的类加载器可以同时装入具有相同二进制名称的几个引用类型。这些类型可能表示相同的类型声明，也可能不表示相同的类型声明。即使这两种类型确实表示相同的类型声明，它们也被认为是不同的。

两个引用类型是相同的运行时类型，当:

- 它们都是类或接口类型，由相同的类加载器定义，并具有相同的二进制名称(§13.1)，在这种情况下，它们有时被称为相同的运行时类或相同的运行时接口。
- 它们都是数组类型，而且它们的组件类型也是相同的运行时类型(§10)。

4. 4 类型变量

类型变量是在类、接口、方法和构造函数体中用作类型的非限定标识符。

类型变量通过声明泛型类、接口、方法或构造函数的类型参数来引入(§8.1.2、§9.1.2、§8.4.4、§8.8.4)。

TypeParameter:
  {TypeParameterModifier} TypeIdentifier [TypeBound]

TypeParameterModifier:
  Annotation

TypeBound:
  `extends` *TypeVariable*
  `extends` *ClassOrInterfaceType {AdditionalBound}*

AdditionalBound:
  `&` *InterfaceType*

作为类型参数声明的类型变量的作用域在§6.3 中指定。

每个声明为类型参数的类型变量都有一个边界。如果没有为类型变量声明边界，则假定为 Object。如果声明了一个边界，它包括:

- 单个类型变量 T，或者
- 类或接口类型 T 后面可能跟着接口类型 I_1 & ... & I_n。

如果任何类型 I_1 ... I_n 是类类型或类型变量，则会产生编译错误。

一个边界的所有组成类型的擦除(§4.6)必须是两两不同的，否则会发生编译时错误。

类型变量不能同时是两个接口类型的子类型，这两个接口类型是同一泛型接口的不同参数化，否则将发生编译时错误。

边界中类型的顺序只有在类型变量的擦除由其边界中的第一个类型决定，并且类类型或类型变量只能出现在第一个位置时才有意义。

类型变量 X 的成员的边界为 T & I_1 & ... & I_n，是类型变量声明位置的交集类型(§4.9)T & I_1 & ... & I_n 的成员。

例子 4.4-1. 类型变量成员

```
package TypeVarMembers;

class C {
   public   void        mCPublic()   {}
   protected             void   mCProtected(){}
             void       mCPackage()  {}
   private  void        mCPrivate()  {}
}

interface I {
    void mI();
}

class CT extends C implements I {
    public void mI() {}
}

class Test {

   <T extends C & I> void test(T t){
   t.mI();          //OK
   t.mCPublic();    //OK
   t.mCProtected();     //OK
   t.mCPackage();       //OK
   t.mCPrivate();         //Compile-time error
  }
}
```

类型变量 T 具有与交集类型 C&I 相同的成员，交集类型 C&I 又具有与空类 CT 相同的成员，它们定义在相同的作用域中，具有等效超类型。接口的成员总是公共的，因此总是被继承(除非被重写)。因此 mI 是

CT 的成员，也是 T 的成员。C 中的所有成员，除了 mCPrivate 以外，都被 CT 继承，因此也是 CT 和 T 类型的成员。

如果 C 与 T 声明在不同的包中，那么对 mCPackage 的调用将导致编译时错误，因为在声明 T 的地方无法访问该成员。

4.5 参数化类型

泛型的类或接口(§8.1.2，§9.1.2)定义了一组参数化类型。

参数化类型是一个类或接口类型，形式为 $C<T_1,...,T_n>$，其中 C 是泛型类或接口的名称，$<T_1,...,T_n>$是表示泛型类或接口的特定参数化的类型参数列表。

泛型类或接口具有类型参数 $F_1,...,F_n$，相应的边界为 $B_1,...,B_n$。参数化类型的每个类型参数 T_i 的作用域覆盖相应边界中列出的所有类型的子类型。也就是说，对于 B_i 中的所有边界类型 S，T_i 是 $S[F_1:=T_1,...,F_n:=T_n]$的子类型(§4.10)。

如果下面描述为真，参数化类型 $C<T_1,...,T_n>$是符合语法规范的：

- C 是泛型类或接口的名字。
- 类型参数的数量与 C 的泛型声明中的类型参数的数量相同。
- 当受到捕获转换时(§5.1.10)产生类型 $C<X_1,...,X_n>$，对于 B_i 里的每一边界类型 S 来说，每一个类型参数 X_i 是 $S[F_1:=X_1,...,F_n:=X_n]$的子类型。

如果参数化类型不符合语法规范，则产生编译错误。

在这个规范中，除非显式地排除，每当我们提到类或接口类型时，也包括参数化类型。

如果下列任意一个为真，则可证明两个参数化类型是不同的：

- 它们是不同泛型类型声明的参数化。
- 它们的任何类型参数都可以被证明是不同的。

鉴于§8.1.2 示例中的泛型类，这里有一些格式良好的参数化类型：

- `Seq<String>`
- `Seq<Seq<String>>`
- `Seq<String>.Zipper<Integer>`
- `Pair<String,Integer>`

还有一些不正确的泛型类参数化：

- `Seq<int>` is illegal, as primitive types cannot be type arguments.
- `Pair<String>` is illegal, as there are not enough type arguments.
- `Pair<String,String,String>` is illegal, as there are too many type arguments.

参数化类型可以是嵌套的泛型类或接口的参数化。 例如，如果非泛型类 C 具有具有一个类型参数的泛型成员类 D，那么 C.D<Object>是一个参数化类型。同时，如果具有一个类型参数的泛型类 C 具有非泛型成员类 D，则成员类类型 C<String>.D 是一个参数化类型，尽管类 D 不是泛型。

4.5.1 参数化类型的类型参数

类型参数可以是引用类型或通配符。通配符在只需要部分了解类型参数的情况下很有用。

TypeArguments:
  `<` *TypeArgumentList* `>`

TypeArgumentList:
  TypeArgument *{*`,` *TypeArgument}*

TypeArgument:
  ReferenceType
  Wildcard

Wildcard:
  {Annotation} `?` *[WildcardBounds]*

WildcardBounds:
  `extends` *ReferenceType*
  `super` *ReferenceType*

通配符可以被赋予显式边界，就像常规类型变量声明一样。一个上界由以下语法表示，其中 B 是上界:

```
? extends B
```

与在方法签名中声明的普通类型变量不同，使用通配符时不需要类型推断。 因此，允许使用以下语法声明通配符的下界，其中 B 是下界:

```
? super B
```

通配符? extends Object 和无边界的通配符?相同。

如果下列情况之一为真，则可证明两种类型参数是不同的:

- 两个参数都不是类型变量或通配符，而且两个参数的类型也不相同。

- 其中一个类型参数是类型变量或通配符，有一个 S 的上界（如有必要，从捕获转换(§5.1.10)获得）；另一个类型参数 T 不是类型变量或通配符；|S| <: |T|不成立，|T| <: |S|也不成立(§4.8, §4.10)。
- 每一个类型参数是类型变量或通配符，有 S 和 T 的上界（如有必要，从捕获转换获得）；|S| <: |T|不成立，|T| <: |S|也不成立

一个类型参数 T_1 包含另一个类型参数 T_2，写为 T_2 <= T_1，如果可以证明 T_2 所表示的类型集合是 T_1 所表示的类型集合的子集，在以下自反和传递闭包的规则下(其中<:表示子类型(§4.10))：

- ? extends T <= ? extends S if T <: S
- ? extends T <= ?
- ? super T <= ? super S if S <: T
- ? super T <= ?
- ? super T <= ? extends Object
- T <= T
- T <= ? extends T
- T <= ? super T

通配符与已建立的类型理论的关系是一个有趣的关系，我们在这里简要地提到了它。通配符是存在类型的一种受限形式。给定泛型类型声明 *G*< *T* extends *B*>，*G*<?>大致类似于 *Some X* <: *B*. *G*<*X*>

从历史上看，通配符是 Atsushi Igarashi 和 Mirko Viroli 作品的直接后代。对更全面的讨论感兴趣的读者可以参考 Atsushi Igarashi 和 Mirko virroli 在第 16 届欧洲面向对象编程会议(ECOOP 2002)上发表的《基于方差的参数类型的子类型》。这项工作本身建立在 Kresten Thorup 和 Mads Torgersen(统一通用性，ECOOP 99)的早期工作的基础上，以及基于声明的方差的长期传统工作，这可以追溯到 Pierre America 关于 POOL 的工作(OOPSLA 89)。

通配符在某些细节上与上述论文中描述的构造不同，特别是在捕获转换(§5.1.10)的使用上，而不是由 Igarashi 和 virroli 描述的闭合操作上。关于通配符的正式描述，请参阅 Mads Torgersen, Erik Ernst 和 Christian Plesner Hansen 在第 12 届面向对象编程基础研讨会(FOOL 2005)上的 Wild FJ。

例子 4.5.1-1. 无边界通配符

```
import java.util.Collection;
import java.util.ArrayList;

class Test {
   static void printCollection(Collection<?> c) {
                                    // a wildcard collection
```

```
        for (Object o : c) {
            System.out.println(o);
        }
    }

    public static void main(String[] args) {
        Collection<String> cs = new ArrayList<String>();
        cs.add("hello");
        cs.add("world"); printCollection(cs);
    }
}
```

请注意，使用 Collection<Object>作为传入参数，c 可能没有那么有用；该方法只能与类型为 Collection<object>的参数表达式一起使用，这将非常罕见。相反，使用无界通配符允许将任何类型的集合作为参数传递。

下面是一个使用通配符参数化数组元素类型的示例:

```
public Method getMethod(Class<?>[] parameterTypes) { ... }
```

Example 4.5.1-2. 边界通配符

```
boolean addAll(Collection<? extends E> c)
```

这里，该方法在接口 Collection<E>中声明，并被设计为将其传入参数的所有元素添加到集合中。一种自然的趋势是使用 Collection<E>作为 c 的类型，但这是不必要的限制。另一种方法是将方法本身声明为泛型:

```
<T> boolean addAll(Collection<T> c)
```

这个版本足够灵活，但是请注意类型参数在签名中只使用一次。这反映了一个事实，即类型形参没有用于表示实参的类型、返回类型和/或抛出类型之间的任何类型的相互依赖关系。在没有这种相互依赖的情况下，泛型方法被认为是糟糕的方式，通配符是首选。

```
Reference(T referent, ReferenceQueue<? super T> queue)
```

在这里，referent 可以插入到任何队列中，其元素类型是 referent 类型 T 的超类型；T 是通配符的下界。

4.5.2 参数类型的成员和构造函数

假设 C 是有类型参数 $A_1,...,A_n$ 的泛型类或接口，$C< T_1, \cdots,T_n>$是 C 的参数化，其中 $1 \leqslant i \leqslant n$，$T_i$是类型而不是通配符。那么:

- 假设 m 是 C 声明的成员或构造函数，类型是 T(§8.2, §8.8.6)。

$c<T_1,...,T_n>$里 m 的类型是 $T[A_1:=T_1,...,A_n:=T_n]$。

- 假设 m 是 D 中的成员或构造函数声明，其中 D 是由 C 扩展的类或由 C 实现的接口。 假设 D<U_1,...,U_k>是 C<T_1,...,T_n>的超类型(§4.10.2)。

C<T_1,...,T_n>里 m 的类型和 D<U_1,...,U_k>里 m 的类型相同。

如果 C 的参数化中的任何类型参数都是通配符，则：

- C<T_1,...,T_n>中的字段、方法和构造函数的类型和 C<T_1,...,T_n> 的捕获转换(§5.1.10)中的字段、方法和构造函数相同。
- 假设 D 是 C 中的一个(可能是泛型的)类或接口声明。那么 C<T_1,...,T_n>里 D 的类型是 D，如果 D 是泛型，所有类型参数是无边界通配符。

  这无关紧要，因为不执行捕获转换就不可能访问参数化类型的成员，并且在类实例创建表达式的关键字 new 后面使用通配符是不可能的(§15.9)。

  上一段唯一的例外是在 instanceof 操作符(§15.20.2)中使用嵌套参数化类型作为表达式时，此时没有应用捕获转换。

在泛型类或接口中声明的静态成员必须使用泛型类或接口的名称来引用(§6.1，§6.5.5.2，§6.5.6.2)，否则将发生编译时错误。

换句话说，通过使用参数化类型引用泛型类型声明中的静态成员是非法的。

4.6 类型擦除

类型擦除是从一个类型 (可能包含参数化类型和类型变量) 到另一个类型 (不是参数化类型或类型变量)的映射。T 的类型擦除写做|T|。擦除映射定义如下：

- 参数化类型 (§4.5) G<T_1,...,T_n>的擦除是|G|。
- 嵌套类型 T.C 的擦除是|T|.C。
- 数组类型 T[]的擦除是|T|[]。
- 对类型变量的擦除(§4.4)就是对其最左边界的擦除。
- 所有其他类型的擦除就是类型本身。

类型擦除还将构造函数或方法的签名(§8.4.2)映射到没有参数化类型或类型变量的签名。构造函数或方法签名 s 的擦除是由与 s 相同的名称和 s 中给出的所有形式参数类型的擦除组成的签名。

如果方法或构造函数的签名被擦除，那么方法的返回类型(§8.4.5)和泛型方法或构造函数的类型参数(§8.4.4，§8.8.4)也会被擦除。

泛型方法签名的擦除没有类型参数。

4.7 可具体化的类型

因为某些类型信息会在编译期间被擦除，所以并非所有类型在运行时都可用。在运行时完全可用的类型称为可具体化类型。

当且仅当下列条件之一成立时，类型是可具体化的:

- 它引用非泛型类或接口类型声明。
- 它是一个参数化类型，其中所有类型参数都是无界通配符(§4.5.1)。
- 它是一种原始类型(§4.8)。
- 它是一种原生类型(§4.2)。
- 它是一种数组类型(§10.1)，其元素类型是可具体化的。
- 它是一个嵌套类型，其中每个类型 T 由一个“.”分隔，T 本身是可具体化的。

例如，如果泛型类 X<T>有一个泛型成员类 Y<U>，那么类型 X<?>.Y<?>是可具体化的，因为 X<?>是可具体化的，Y<?>也是可具体化的。类型 X<?>.Y<Object>不是可具体化的，因为 Y<Object>不是可具体化的。

交集类型不是可具体化的。

不使所有泛型类型具体化是涉及 Java 编程语言的类型系统的最关键的、也是最具争议的设计决策之一。

最终，这个决定的最重要动机是与现有代码的兼容性。 从简单的意义上说，添加新构造(如泛型)对现有代码没有任何影响。Java 编程语言本身与早期版本是兼容的，只要在早期版本中编写的每个程序在新版本中保留其意义。然而，这个概念，可以被称为语言兼容性，是纯粹的理论兴趣。真正的程序(甚至是琐碎的程序，如“Hello World”)由几个编译单元组成，其中一些由 Java SE 平台提供(如 java.lang 或 java.util 的元素)。实际上，最低的要求是平台兼容性—任何为以前版本的 Java SE 平台编写的程序在新版本中继续保持不变的功能。

提供平台兼容性的一种方法是保持现有平台功能不变，只添加新功能。例如，不要修改 java.util 中现有的 Collections 层次结构，可以引入一个利用泛型的新库。

这种方案的缺点是，Collection 库的之前的客户端很难迁移到新库。集合用于在独立开发的模块之间交换数据; 如果某个供应商决定切换到新的通用库，那么该供应商还必须发布其代码的两个版本，以便与客户端兼容。依赖于其他供应商代码的库在供应商的库更新之前不能修改为使用泛型。 如果两个模块相互依赖，则必须同时进行更改。

显然，如上所述，平台兼容性并没有为采用普遍存在的新特性(如泛型)提供现实的途径。因此，泛型类型系统的设计寻求支持迁移兼容性。迁移兼容性允许现有代码的演化利用泛型，而不需

要在独立开发的软件模块之间施加依赖关系。

迁移兼容性的代价是不可能实现泛型类型系统的完整而健全的具体化，至少在进行迁移时是这样。

4.8 原始类型

为了方便与非泛型遗留代码的交互，可以使用对参数化类型(§4.5)的擦除(§4.6)或对元素类型为参数化类型的数组类型(§10.1)的擦除(§4.6)作为类型。这样的类型称为原始类型。

更精确地说，原始类型定义为以下类型之一：

- 采用泛型类或接口声明的名称而不附带类型参数列表而形成的引用类型。
- 元素类型为原始类型的数组类型。
- 不从 R 的超类或超接口继承的原始类型 R 的内部成员类的名称。

非泛型类或接口的类型不是原始类型。

要了解为什么原始类型的内部成员类的名称被认为是原始的，考虑以下示例：

```
class Outer<T>{
   T t;
   class Inner {
   T setOuterT(T t1) { t = t1; return t; }
   }
}
```

Inner 的成员类型取决于 Outer 的类型参数。如果 Outer 是原始的，那么 Inner 也必须被视为原始的，因为 T 没有有效的绑定。

该规则仅适用于不继承的内部成员类。依赖于类型变量的继承内部成员类将作为原始类型继承，这是原始类型的超类型被擦除规则的结果，本节稍后将对此进行描述。

上述规则的另一个含义是原始类型的泛型内部类本身只能用作原始类型：

```
class Outer<T>{
   class Inner<S> {
      S s;
   }
}
```

不可能以部分原始类型(“罕见”类型)访问 Inner：

```
Outer.Inner<Double> x = null; // illegal Double d = x.s;
```

因为 Outer 本身是原始的，因此它的所有内部类包括 Inner 也是原始的，所以不可能向 Inner 传递任何类型参数。

原始类型的超类类型(以及超接口类型)是指定类或接口的超类类型(以及超接口类型)的擦除。

原始类型 C 的构造函数(§8.8)、实例方法(§8.4、§9.4)或非静态字段(§8.3)的类型，如果不是从超类或超接口继承而来，就是在泛型类或接口 C 中擦除其类型。

原始类型 C 的继承实例方法类型或非静态字段的类型(其中成员声明在类或接口 D 中)是 C 的超类型中命名为 D 的成员的类型。

原始类型 C 的静态方法或静态字段的类型与泛型类或接口 C 中的类型相同。

将类型参数传递给未从其超类或超接口继承的原始类型的非静态成员类或接口会导致编译时错误。

试图将参数化类型的成员类或接口用作原始类型会导致编译时错误。

这意味着对“稀有”类型的禁止扩展到限定类型被参数化的情况，但我们尝试使用内部类作为原始类型:

```
Outer<Integer>.Inner x = null; // illegal
```

这与上面讨论的情况相反。对于这种半生不熟的类型，没有实际的理由。在遗留代码中，没有使用类型参数。在非遗留代码中，我们应该正确使用泛型类型并传递所有所需的类型参数。

只允许使用原始类型作为对遗留代码兼容性的让步。不建议在 Java 编程语言中引入泛型后编写的代码中使用原始类型。Java 编程语言的未来版本可能不允许使用原始类型。

为了确保始终标记潜在的违反类型规则的行为，对原始类型成员的某些访问将导致编译时未检查警告。当访问原始类型的成员或构造函数时，编译时未检查警告的规则如下:

- 在字段赋值时：如果字段访问表达式（§15.11）中的基本类型是原始类型，则如果擦除更改字段的类型，则会出现编译时未检查警告。
- 调用方法或构造函数时：如果要搜索的类或接口的类型（§15.12.1）是原始类型，则如果擦除更改了方法或构造函数的任何形式参数类型，则会出现编译时未检查警告。
- 从字段读取或创建原始类型的类实例时，如果形式参数类型在擦除下没有更改

（即使返回类型和/或 throws 子句更改），方法调用不会出现编译时未检查警告。

请注意，上述未加检查的警告不同于可能来自窄化引用转换(§5.1.6)、未加检查的转换(§5.1.9)、方法声明(§8.4.1、§8.4.8.3)和某些表达式(§15.12.4.2、§15.13.2、§15.27.3)的未加检查的警告。

这里的警告涵盖了遗留用户使用通用库的情况。例如，标准库声明了一个泛型类 Foo<T extends String>，其中字段 f 的类型为 Vector<T>，但消费者将一个整数数组赋给 e.f，其中 e 的原始类型为 Foo。遗留消费者收到一个警告，因为它可能对泛化库的泛化消费者造成堆污染(§4.12.2)。

(注意，遗留使用者可以从标准库中将 Vector<String>赋值给自己的 Vector 变量，而不会收到警告。也就是说，Java 编程语言的子类型规则(§4.10.2)使原始类型的变量可以被赋值为该类型的任何参数化实例。)

来自未检查转换的警告涵盖了双重情况，即通用消费者使用遗留库。例如，标准库的一个方法具有原始返回类型 Vector，但是使用者将方法调用的结果分配给 Vector<String>类型的变量。这是不安全的，因为原始向量可能具有与 String 不同的元素类型，但仍然允许使用未检查的转换来支持与遗留代码的接口。来自未检查转换的警告表明，泛化使用者可能会在程序的其他地方遇到堆污染的问题。

例子 4.8-1. 原始类型

```
class Cell<E> {
   E value;

   Cell(E v) { value = v; }
   E get()                      { return value; }
   void set(E v) { value = v; }

   public static void main(String[] args) {
      Cell x = new Cell<String>("abc");
      System.out.println(x.value);                  //  OK, has type Object
      System.out.println(x.get());                  //  OK, has type Object
      x.set("def");                                 //  unchecked warning
      }
   }
```

例子 4.8-2. 原始类型和继承

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

class NonGeneric {
   Collection<Number> myNumbers() { return null; }
}
abstract class RawMembers<T> extends NonGeneric
                             implements Collection<String> {
   static Collection<NonGeneric> cng =
```

```
    new ArrayList<NonGeneric>();

  public static void main(String[] args) {
    RawMembers rw = null;
    Collection<Number> cn = rw.myNumbers();
                            // OK
    Iterator<String> is = rw.iterator();
                            // Unchecked warning
    Collection<NonGeneric> cnn = rw.cng;
                            // OK, static member
    }
}
```

在这个程序中(不打算运行)，RawMembers<T>继承这个方法:

Iterator<String> iterator()

从 Collection<String>这个超接口。原始类型 RawMembers 从 Collection 继承 iterator(), 擦除 Collection<String>, 这意味着 RawMembers 中 iterator()的返回类型是 Iterator. 因此，尝试将 rw.iterator()赋值给 Iterator<String>需要进行未检查的转换，因此会发出编译时未检查的警告。

相反，RawMembers 从非泛型类继承 myNumbers()，该类的擦除也是非泛型的。因此，RawMembers 中的 myNumbers()的返回类型不会被擦除，并且尝试将 rw.myNumbers()赋值给 Collection<Number>不需要未检查的转换，因此不会发出编译时未检查的警告。

类似地，静态成员 cng 即使在通过原始类型对象访问时也保留其参数化类型。注意，通过实例访问静态成员被认为是糟糕的风格，不建议使用。

这个例子揭示了原始类型的某些成员不会被擦除，即类型被参数化的静态成员，以及从非泛型超类型继承的成员。

原始类型与通配符密切相关。两者都基于存在类型。原始类型可以被认为是通配符，其类型规则故意是不健全的，以适应与遗留代码的交互。 历史上，原始类型先于通配符;它们首次在 GJ 中被介绍，并在 1998 年 10 月的 ACM 面向对象编程、系统、语言和应用会议(OOPSLA 98)上，由 Gilad Bracha、Martin Odersky、David Stoutamire 和 Philip Wadler 撰写的论文《为过去创造安全的未来:为 Java 编程语言添加泛型》中进行了描述。

4.9 交集类型

交集类型形式为 T_1 &⋯& T_n (n > 0)，其中 T_i (1 ≤ i ≤ n)为类型。

交集类型可以由类型参数边界(§4.4)和转换表达式(§15.16)派生;它们也出现在捕获转换(§5.1.10)和最小上限计算(§4.10.4)的过程中。

交集类型的值是指 1≤ i ≤ n 条件下所有类型 T_i 的值。

每个交集类型 T_1 &⋯& T_n 引入一个概念类或接口，用于识别交集类型的成员，如下

所示:

- 对于每个 $T_i(1\leqslant i\leqslant n)$，让 C_i 成为 $T_i <: C_i$ 的最特定的类或数组类型。那么对于任何 $i(1\leqslant i\leqslant n)$，必须存在某个 C_k 使 $C_k <: C_i$，否则将发生编译时错误。
- 对于 $1\leqslant j\leqslant n$，如果 T_j 是一个类型变量，则让 T_j' 是一个接口，其成员与 T_j 的 public 成员相同;否则，如果 T_j 是一个接口，则让 T_j' 为 Tj。
- 如果 C_k 是 Object，则产生一个概念接口;否则，产生一个拥有直接超类类型 C_k 的概念类。这个类或接口有直接的超接口类型 T_1'，…，T_n' 和出现在交集类型的包中的声明。

交集类型的成员是它产生的类或接口的成员。

> 有必要详细讨论交集类型和类型变量边界之间的区别。每一种类型变量的边界都会产生一种交集类型。这种交集类型通常很简单，只包含一个类型。边界的形式是有限制的(只有第一个元素可以是类或类型变量，并且边界中只能出现一个类型变量)，以防止出现某些尴尬的情况。但是，捕获转换可能导致创建边界更一般的类型变量，例如数组类型。

4.10子类型化

子类型和超类型的关系是类型上的二元关系。

类型的超类型是通过在直接超类型关系上的自反和传递闭包获得的，写做 $S >_1 T$，该关系由本节后面给出的规则定义。S:> T 表示 S 和 T 之间存在超类型关系。

如果 S： > T 并且 S≠T，S 是 T 的一个超类型，写做 S > T。

类型 T 的子类型是 U 的所有类型，因此 T 是 U 的超类型，并且是空类型。我们用 T <: S 表示类型 T 和 S 之间存在的子类型关系。

如果 T <: S 并且 S≠T，T 是 S 的子类型。

如果 $S >_1 T$，T 是 S 的直接子类型，写做 $T <_1 S$。

子类型不通过参数化类型扩展：T <: S 并不表示 C<T> <: C<S>。

4.10.1 原生类型中的子类型

以下规则定义了原生类型之间的直接超类型关系:

- double $>_1$ float
- float $>_1$ long
- long $>_1$ int

- int >₁ char
- int >₁ short
- short >₁ byte

4.10.2 类和接口类型中的子类型

给定一个非泛型类或接口声明 C，类型 C 的直接超类型如下:

- C 的直接超类类型(§8.1.4)，如果 C 是一个类。
- C 的直接超接口类型(§8.1.5，§9.1.3)。
- Object 类型，如果 C 是一个没有直接超接口类型的接口(§9.1.3)。

  给定具有类型参数 $F_1,...,F_n$ ($n > 0$)的泛型类或接口 C，原始类型 C(§4.8)的直接超类型如下:

- 擦除(§4.6)C 的直接超类类型, 如果 *C* 是一个类。
- 擦除 C 的直接超接口。
- 类型 Object, 如果 C 是一个没有直接超接口类型的接口。

  给定具有类型参数 $F_1,...,F_n$ (n > 0)的泛型类或接口, 参数化类型 $c<T_1,...,T_n>$的直接超类型如下，对每一个 T_i(1 ⩽ i ⩽ n)是一个类型:

- 替换 $[F_1:=T_1,...,F_n:=T_n]$ 应用于 C 的直接超类型,如果 C 是一个类。
- 替换 $[F_1:=T_1,...,F_n:=T_n]$ 应用于 C 的直接超接口类型。
- $C<S_1,...,S_n>$, 其中 S_i 包含 T_i (1 ⩽ i ⩽ n) (§4.5.1)。
- 类型 Object, 如果 C 是没有直接超接口类型的接口。
- 原始类型 C。

给定泛型类型声明 $C<F_1,...,F_n>$ (n > 0), 参数化类型 $C<R_1,\cdots,R_n>$ 的直接超类型是参数化类型 $C<X_1,...,X_n>$的直接超类型，其中至少有一个 R_i (1 ⩽ i ⩽ n)是通配符类型参数，这是对 $C<R_1,...,R_n>$应用捕获转换的结果(§5.1.10)。

交集类型 T_1 & ... & T_n 的直接超类型是 T_i (1 ⩽ i ⩽ n)。

类型变量的直接超类型是其边界中列出的类型。

类型变量是其下界的直接超类型。

空类型的直接超类型是除空类型本身之外的所有引用类型。

4.10.3 数组类型中的子类型

以下规则定义了数组类型之间的直接超类型关系:

- 如果 S 和 T 都是引用类型，那么 S[] $>_1$ T[] 当且仅当 S $>_1$ T。
- Object $>_1$ Object[]
- Cloneable $>_1$ Object[]
- java.io.Serializable $>_1$ Object[]
- 如果 P 是原生类型，那么:
  - Object $>_1$ P []
  - Cloneable $>_1$ P []
  - java.io.Serializable $>_1$ P []

4.10.4 最小上界

一组引用类型的最小上界或“lub”是一种共享超类型，它比任何其他共享超类型都具体(也就是说，没有其他共享超类型是最小上界的子类型)。这种类型，lub(U_1, ..., U_k)定义如下。

如果 k = 1, 那么 lub 就是类型本身:lub(U) = U。

否则:

- 对每一个 U_i ($1 \leqslant i \leqslant k$):

  假设 ST(U_i) 是 U_i 的超类集合。

  假设 EST(U_i), 是 U_i 的擦除超类集合:

  EST(U_i) = { |W| | W 在 ST(U_i)中}，其中|W| 是 W 的擦除。

  > 计算擦除超类型集的原因是为了处理类型集包括泛型类型的几个不同参数化的情况。
  >
  > 例如，给定 List<String>和 List<Object>，简单的将集合 ST(List<String>) = {List<String>, Collection<String>, Object } 和 ST(List<Object>) = {List<Object>, Collection<Object>, Object } 相交会产生一个集合 {Object}, 这样我们就忘记了上界可以被认为是一个 List。
  >
  > 相反地, 将集合 EST(List<String>) = {List, Collection, Object} 和集合 EST(List<Object>) = {List, Collection, Object} 相交产生{List, Collection, Object}, 这最终将使我们能够产生 List<?>。

- 假设 EC 是 U_1 ... U_k 的擦除候选集合，为所有集合 EST(U_i) ($1 \leqslant i \leqslant k$)的交集。

- 假设 MEC, 是 U_1 ... U_k的最小擦除候选集:

  MEC = { v | v 在 EC 中, 对于所有 EC 中的 W≠V, W <: V 不成立}

  因为我们希望推断出更精确的类型，所以希望过滤掉任何是其他候选项的超类型的候选项。这就是计算 MEC 所完成的工作。在我们的例子里，设置 EC = { List, Collection, Object }, 所以 MEC = { List }。下一步是为 MEC 中被擦除的类型恢复类型参数。

- 对于 MEC 中任何泛型类型的元素 G:

  设置 G 的相关参数化 Relevant(*G*)为:

  Relevant(G) = { v | 1 ⩽ i ⩽ k: v 在 ST(U_i) 中并且 V = G<...> }

  在我们的例子中，MEC 的唯一的泛型元素是 List, Relevant(List) = { List<String>, List<Object> }。现在我们将为 List 寻找一个既包含 String 又包含 Object 的类型参数(§4.5.1)。

  这是通过下面定义的最小包含参数化(lcp)操作来完成的。第一行将集合上的 lcp()定义为由集合元素组成的列表上的操作，例如 Relevant(List)。下一行将此类列表上的操作定义为列表元素的成对减少。 第三行是 lcp()在参数化类型对上的定义，这反过来又依赖于最小包含类型参数（lcta）的概念。lcta()是为所有可能的情况定义的。

设 G 的"候选"参数化，candidate (G)，是泛型类型 G 的最具体的参数化，包含 G 的所有相关参数化:

Candidate(G) = lcp(Relevant(G))

lcp()，包含最少的参数化，定义如下:

- lcp(S) = lcp(e_1,⋯, e_n) 其中 e_i (1 ⩽ i ⩽ n) 在 S 中
- lcp(e1, ⋯, en) = lcp(lcp(e_1,e_2),e_3,⋯,e_n)
- lcp(G<X_1,⋯,X_n>, G< Y_1,⋯,Y_n>) = G<lcta(X_1, Y_1),..., lcta(X_n,Y_n)>
- lcp(G< X_1,⋯,X_n >) = G<lcta(X_1), ..., lcta(X_n)>

lcta(), 包含最少的参数化，定义如下: (假设 U 和 V 是类型)

- lcta(U, V) = U 如果 U = v, 否则? extends lub(U, V)
- lcta(U, ? extends V) = ? extends lub(U, V)
- lcta(U, ? super V) = ? super glb(U, V)
- lcta(? extends U, ? extends V) = ? extends lub(U, V)
- lcta(? extends U, ? super V) = ?

- lcta(? super U, ? super V) = ? super glb(U, V)
- lcta(U) = ? 如果 U 的上界是 Object, 否则? extends lub(U,Object)

glb()定义在 §5.1.10。

• 定义 lub(U_1,⋯,U_k) 为:

Best(W_1) & ... & Best(W_r)

其中 W_i (1 ⩽ i ⩽ r) 是 MEC 的元素, U_1,⋯,U_k的最小擦除候选集;

如果这些元素中的任何一个是泛型的，我们使用候选参数化(以便恢复类型实参):

Best(X) = Candidate(X) 如果 X 是泛型; 否则等于 X。

严格地说，这个 lub()函数只近似于最小上界。

形式上，可能存在其他类型的 T 使得所有的 U_1,⋯,U_k是 T 的子类型，T 是 lub(U_1, ..., U_k)的子类型。然而，Java 语言的编译器必须实现如上所述的 lub()。

lub()函数可能产生无限类型。这是允许的，Java 语言的编译器必须识别这种情况，并使用循环数据结构适当地表示它们。

> 无限类型的可能性源于对 lub()的递归调用。熟悉递归类型的读者应该注意，无限类型与递归类型是不同的。

4.10.5 类型投影

合成类型变量是编译器在捕获转换(§5.1.10)或推理变量解析(§18.4)期间引入的类型变量。

有时需要找到类型的接近超类型，在这种情况下，该超类型没有提到某些合成类型变量。这是通过应用到类型上的向上投影实现的。

类似地，可以应用向下投影来查找类型的接近子类型，该子类型没有提到某些合成类型变量。因为这种类型并不总是存在，向下投影是一个偏函数。

这些操作将一组不应再被引用的类型变量作为输入，称为受限制的类型变量。当重复这个操作，受限制的类型变量集将隐式地传递给递归应用程序。

类型 T 相对于一组受限类型变量的向上投影定义如下:

- 如果 T 没有提到任何受限类型变量，那么结果是 T。
- 如果 T 是一个受限类型变量，那么结果就是 T 的上界向上的投影。

- 如果 T 是一个参数化的类类型或参数化的接口类型，$G<A_1,...,A_n>$，那么结果是 $G<A_1',..., A_n';>$，其中 $1 \leqslant i \leqslant n$，$A_i'$ 来自于 A_i，具体见下：
  - 如果 A_i 没有提到任何受限类型变量，那么 $A_i' = A_i$。
  - 如果 A_i 是一个提到受限类型变量的类型，那么假设 U 为 A_i 的向上投影。A_i' 是一个通配符，定义为三种情况：
    - 如果 U 不是 Object，如果 G 的第 i 个参数 B_i 的声明边界提到了 G 的类型参数，或者 B_i 不是 U 的子类型，那么 A_i' 是上界通配符 ? extends U。
    - 否则，如果 A_i 的向下投影是 L，那么 A_i' 是下界通配符? super L。
    - 否则，A_i 的向下投影未定义，A_i' 是无边界通配符?。
  - 如果 A_i 是一个提到受限类型变量的上界通配符，那么设 U 为通配符边界的向上投影。A_i' 是上界通配符? extends U。
  - 如果 A_i 是一个提到受限类型变量的下界通配符，那么如果通配符边界向下的投影是 L，那么 A_i' 是下界通配符，? super L；如果通配符边界的向下投影未定义，那么 Ai' 是无边界通配符?。
- 如果 T 是数组类型 S[]，则结果是一个数组类型，其组件类型是 S 的向上投影。
- 如果 T 是交集类型，那么结果也是交集类型。对于 T 的每个元素 S，其结果是 S 的向上投影。

类型 T 对一组受限类型变量的向下投影是偏函数，定义如下：

- 如果 T 没有提到任何受限类型变量，那么结果是 T。
- 如果 T 是一个受限类型变量，那么如果 T 有一个下界，如果这个边界的向下投影是 L，结果就是 L；如果 T 没有下界，或者边界的向下投影没有定义，那么结果就没有定义。
- 如果 T 是参数化的类类型或参数化的接口类型 $G<A_1,...,A_n>$，那么结果是 $G<A_1',..., A_n'>$，如果，对于 $1 \leqslant i \leqslant n$，类型参数 A_i' 来自 A_i；否则，结果未定义：
  - 如果 A_i 没有提到一个受限的类型变量，那么 $A_i'=A_i$。
  - 如果 A_i 是一个提到受限类型变量的类型，那么 A_i' 未定义。
  - 如果 A_i 是一个提到受限类型变量的上界通配符，那么如果通配符边界的向下投影是 U，那么 A_i' 是上界通配符，? extends U；如果通配符边界的向下投影未定义，那么 A_i' 未定义。
  - 如果 A_i 是一个提到受限制类型变量的下界通配符，则设 L 为通配符边界的

向上投影。A_i' 是下界通配符, ? super L.

- 如果 T 是数组类型, S[], 那么如果 S 的向下投影是 S', 结果是 S'[];如果 S 的向下投影未定义, 那么结果也未定义。
- 如果 T 是交集类型, 那么对于每一个元素 T 都定义了向下投影, 其结果是一个交集类型，其元素是 T 元素的向下投影; 如果对 T 的任何元素向下投影都没有定义，那么结果也未定义。

像 lub(§4.10.4)一样，向上投影和向下投影可能产生无限类型，这是由在类型变量边界上的递归导致的。

4. 11 类型用在哪

类型在大多数类型的声明和某些类型的表达式中使用。具体来说，有 17 种使用类型的类型上下文:

• 在声明中:

1. 类声明中的 extends 或 implements 语句中的类型 (§8.1.4, §8.1.5)
2. 接口声明中的 extends 语句中的类型 (§9.1.3)
3. 方法返回类型(§8.4.5, §9.4), 包括注解接口的元素类型 (§9.6.1)
4. 方法或构造函数的 throws 语句中的类型(§8.4.6, §8.8.5, §9.4)
5. 泛型类、接口、方法或构造函数的类型参数声明的 extends 语句中的类型 (§8.1.2, §9.1.2, §8.4.4, §8.8.4)
6. 类或接口的字段声明中的类型 (§8.3, §9.3), 包括枚举常量(§8.9.1)
7. 方法、构造函数或 lambda 表达式的形式参数声明中的类型 (§8.4.1, §8.8.1, §9.4, §15.27.1)
8. 方法接收参数的类型(§8.4)
9. 局部变量声明中的类型可以是语句(§14.4.2，§14.14.1，§14.14.2，§14.20.3)或模式(§14.30.1)。
10. 异常形参声明中的类型 (§14.20)
11. 记录类的记录组件声明中的类型 (§8.10.1)

• 在表达式中:

12. 显式构造函数调用语句、类实例创建表达式、方法调用表达式或方法引用表达式的显式类型参数列表中的类型 (§8.8.7.1, §15.9, §15.12, §15.13)
13. 在非限定的类实例创建表达式中，作为要实例化的类类型(§15.9)或作为要实例化的匿名类的直接超类类型或直接超接口类型 (§15.9.5)
14. 数组创建表达式中的元素类型（§15.10.1）
15. 强制转换表达式的强制转换操作符中的类型 (§15.16)
16. instanceof 类型比较操作符后面的类型 (§15.20.2)
17. 在方法引用表达式（§15.13）中，作为搜索成员方法的引用类型，或作为要构造的类类型或数组类型。

此外，类型用作:

- 上述任何上下文中数组类型的元素类型; 以及
- 在上述任何上下文中，参数化类型的非通配符类型参数或通配符类参数的边界。

最后，Java 编程语言中有三个特殊术语表示类型的使用:

- 无边界通配符 (§4.5.1)
- …在变量参数数量的类型(§8.4.1)中，表示数组类型
- 构造函数声明中的类型的简单名称(§8.8)，用来指示构造对象的类

类型在类型上下文中的含义是:

- §4.2, 用于原生类型
- §4.4, 用于类型参数
- §4.5, 用于参数化的类和接口类型，或作为参数化类型中的类型参数或作为参数化类型中的通配符类型参数的边界出现
- §4.8, 用于原始的类和接口类型
- §4.9, 用于类型参数边界内的交集类型
- §6.5, 用于非泛型类、接口和类型变量的类型
- §10.1, 对于数组类型

一些类型上下文限制了引用类型的参数化方式:

- 以下类型上下文要求，如果一个类型是参数化引用类型，它没有通配符类型参数:

- 类声明的 extends 或 implements 语句 (§8.1.4, §8.1.5)
- 接口声明的 extends 语句(§9.1.3)
- 在非限定的类实例创建表达式中，作为要实例化的类类型(§15.9)或作为要实例化的匿名类的直接超类类型或直接超接口类型 (§15.9.5)
- 在方法引用表达式(§15.13)中，作为查找成员方法的引用类型或作为构造的类类型或数组类型。

此外，显式构造函数调用语句、类实例创建表达式、方法调用表达式或方法引用表达式(§8.8.7.1、§15.9、§15.12、§15.13)的显式类型参数列表中不允许有通配符类型参数。

• 以下类型上下文要求，如果一个类型是参数化引用类型，它只有无界通配符类型参数(即它是一个可具体化的类型):

- 作为数组创建表达式中的元素类型 (§15.10.1)
- 作为关系运算符 instanceof 后面的类型(§15.20.2)

• 以下类型上下文完全禁止参数化引用类型，因为它们涉及异常，且异常的类型是非泛型的 (§6.1):

- 作为方法或构造函数可以抛出的异常的类型(§8.4.6, §8.8.5, §9.4)
- 在异常参数声明中 (§14.20)

在使用类型的任何类型上下文中，都可以对表示原始类型的关键字或表示引用类型的简单名称的标识符进行注解。还可以通过在数组类型中所需嵌套级别的[的左侧写入注解来注解数组类型。在这些位置上的注解称为类型注解，在§9.7.4 中指定。下面是一些例子:

- @Foo int[] f; 注解原生类型 int
- int @Foo [] f; 注解数组类型 int[]
- int @Foo [][] f; 注解数组类型 int[][]
- int[] @Foo [] f; 注解数组类型 int[]，它是数组类型 int[][]的组件类型

声明中出现的某些类型上下文与许多声明上下文占据相同的语法空间(§9.6.4.1):

- 方法返回类型 (包括注解接口的元素类型)
- 类或接口的字段声明中的类型 (包括枚举常量)
- 方法、构造函数或 lambda 表达式的形式参数声明中的类型
- 局部变量声明中的类型
- 异常参数声明中的类型

- 记录类的记录组件声明中的类型

程序中相同的语法位置既可以是类型上下文，也可以是声明上下文，这是因为声明的修饰符直接位于声明实体的类型之前。§9.7.4 解释了在这样的位置上的注解如何被认为出现在类型上下文中或声明上下文中或两者中。

例子 4.11-1. 类型的使用

```
    import java.util.Random;
    import java.util.Collection;
    import java.util.ArrayList;

    class MiscMath<T extends Number> {
       int divisor;
       MiscMath(int divisor) { this.divisor = divisor; }
       float ratio(long l) {
          try {
              l /= divisor;
          } catch (Exception e) {
            if (e instanceof ArithmeticException)
               l = Long.MAX_VALUE;
            else
               l = 0;
            }
         return (float)l;
    }
    double gausser() {
         Random r = new Random();
         double[] val = new double[2];
         val[0] = r.nextGaussian();
         val[1] = r.nextGaussian();
         return (val[0] + val[1]) / 2;
      }
    Collection<Number> fromArray(Number[] na) {
         Collection<Number> cn = new ArrayList<Number>();
         for (Number n : na) cn.add(n);
         return cn;
    }
    <S> void loop(S s) { this.<S>loop(s); }
}
```

在本例中，类型用于以下声明:

- 字段，它们是类的类变量和实例变量(§8.3)和接口的常量(§9.3);类 MiscMath 里的字段 divisor 被声明为 int 类型
- 方法参数 (§8.4.1); 方法 ratio 的参数 l 定义为 long 类型
- 方法结果(§8.4); 方法 ratio 结果类型为 float, 方法 gausser 结果类型为 double
- 构造器参数 (§8.8.1); MiscMath 构造器的参数类型为 int
- 局部变量 (§14.4, §14.14); 方法 gausser 的局部变量 r 和 val 类型为 Random 和 double[] (double 数组)

- 异常参数(§14.20); catch 语句的异常参数 e 类型为 Exception
- 类型参数 (§4.4); 在这里，MiscMath 的类型参数是一个类型变量 T，类型 Number 作为其声明的边界
- 在任何使用参数化类型的声明中; 在参数化类型 Collection<Number>里，Number 类型用作类型参数 (§4.5.1)。

和在下列的表达式中:

- 创建类实例 (§15.9); 方法 gausser 的局部变量 r 由使用类型 Random 的类实例创建表达式初始化
- 泛型类(§8.1.2)实例创建(§15.9); Number 用作表达式 new ArrayList<number>()中的类型参数
- 创建数组(§15.10.1); 在这里，方法 gausser 的局部变量 val 由数组创建表达式初始化，该表达式创建了一个大小为 2 的 double 数组
- 泛型方法 (§8.4.4) 或构造函数 (§8.8.4) 调用 (§15.12); 在这里，方法循环使用显式类型参数 S 调用自身
- 强制类型转换 (§15.16); 这里 ratio 方法的返回语句使用强制类型转换为 float
- instanceof 操作符(§15.20.2); 在这里，instanceof 操作符测试 e 是否与 ArithmeticException 类型赋值兼容

4. 12 变量

一个变量是一个存储位置，它有一个相关的类型，有时称为它为编译时类型，它要么是一个原生类型(§4.2)，要么是一个引用类型(§4.3)。

变量的值通过赋值来改变 (§15.26)， 或者通过前缀或后缀自增(++)或自减(--)运算符来改变 (§15.14.2, §15.14.3, §15.15.1, §15.15.2)。

Java 编程语言的设计保证了变量值与其类型的兼容性，只要程序不产生编译时未检查的警告(§4.12.2)。在编译时,默认值(§4.12.5) 通常是兼容的，对变量的所有赋值都要检查赋值兼容性(§5.2)，但是，在涉及数组的单个情况下，要进行运行时检查(§10.5)。

4.12.1 原生类型变量

原生类型的变量总是保存该原生类型的原生值。

4.12.2 引用类型变量

类类型 T 的变量可以保存空引用，也可以保存对类 T 或作为 T 子类的任何类的实例的引用。

接口类型的变量可以保存空引用或实现该接口的任何类的实例的引用。

> 注意，不能保证变量总是引用其声明类型的子类型，而只能引用声明类型的子类或子接口。这是由下面讨论的堆污染的可能性导致的。

如果 T 是一个原生类型，那么类型为“T 的数组”的变量可以保存空引用或对任何类型为“T 的数组”的数组的引用。

如果 T 是引用类型，那么类型为“T 的数组”的变量可以保存空引用或对类型为“S 的数组”的任何数组的引用，其中类型 S 是类型 T 的子类或子接口。

Object[]类型的变量可以保存对任何引用类型的数组的引用。

Object 类型的变量可以保存空引用或对任何对象的引用，无论该对象是类的实例还是数组。

参数化类型的变量可能引用不属于该参数化类型的对象。这种情况称为堆污染。

堆污染只有在以下情况下才会发生:程序执行了一些涉及原始类型的操作，这会引起编译时未检查的警告(§4.8，§5.1.6，§5.1.9，§8.4.1，§8.4.8.3，§8.4.8.4，§9.4.1.2，§15.12.4.2)，或者程序通过原始或非泛型超类型的数组变量重命名一个非具体化元素类型的数组变量。

例如，代码:

```
List l = new ArrayList<Number>();
List<String> ls = l; // Unchecked warning
```

产生一个编译时未检查警告, 因为无论是在编译时(在编译时类型检查规则的限制内)还是在运行时，都无法确定变量 l 是否确实引用 List<String>。

如果执行上面的代码，就会产生堆污染，因为变量 ls(声明为 List<String>)所引用的值实际上并不是 List<String>。

在运行时无法识别问题，因为类型变量没有具体化，因此实例在运行时不携带用于创建它们的类型参数的任何信息。

在上面给出的一个简单示例中，在编译时识别情况并给出错误似乎应该很直观。然而，在一般(和典型)情况下，变量 l 的值可能是调用单独编译的方法的结果，或者它的值可能依赖于任意的控制流。因此，上面的代码是非典型的，而且确实是非常糟糕的风格。

此外，Object[]是所有数组类型的超类型这一事实意味着可能会发生不安全的混淆，从而导致堆污染。例如，以下代码可编译，因为它是静态类型正确的:

```
static void m(List<String>... stringLists) {
    Object[] array = stringLists;
    List<Integer> tmpList = Arrays.asList(42);
    array[0] = tmpList;     // (1)
    String s = stringLists[0].get(0); // (2)
```

}

堆污染发生在(1)处，因为 stringLists 数组中本应引用 List<String>的组件现在引用 List<Integer>。如果同时存在通用超类型(Object[])和不可具体化类型(形式参数的声明类型 List<String>[])，则无法检测这种污染。在(1)处没有未经检查的警告是合理的;然而，在运行时，ClassCastException 将在(2)处发生。

编译时未检查的警告将在上述方法的任何调用时被给出，因为 Java 编程语言的静态类型系统认为调用会创建一个元素类型为 List<String>的数组，它是不可实现(§15.12.4.2)。当且仅当方法的主体相对于变量数量参数是类型安全的，那么程序员可以使用 SafeVarargs 注解关闭调用时警告(§9.6.4.7)。由于如上所述的方法体会导致堆污染，因此使用注解为调用者禁用警告是完全不合适的。

最后，请注意，stringLists 数组可以通过 Object[]以外类型的变量进行别名，仍然可能发生堆污染。例如，数组变量的类型可以是 java.util.Collection[](一种原始元素类型),上面方法的主体将在编译时没有警告或错误，但仍然会造成堆污染。如果 Java SE 平台将 Sequence 定义为 List<T>的非泛型超类型，那么使用 Sequence 作为数组类型也会导致堆污染。

变量将始终引用一个对象，该对象是表示参数化类型的类的实例。

在上面的例子中，ls 的值始终是一个提供 List 表示的类的实例。

只有在不使用参数化类型的遗留代码与使用参数化类型的现代代码相结合时，才应该使用从原始类型的表达式赋值到参数化类型的变量。

如果不发生需要发出编译时未检查警告的操作，且不发生具有不可具体化元素类型的数组变量的不安全混淆，则不会发生堆污染。注意，这并不意味着堆污染只会在编译时出现未检查的警告时才会发生。有可能运行一个程序，其中一些二进制文件是由较老版本的 Java 编程语言的编译器生成的，或者来自显式地抑制未检查警告的源代码。这种做法充其量是不健康的。

相反，尽管执行的代码可能(也可能确实)引起编译时未检查的警告，但也有可能不会发生堆污染。实际上，良好的编程实践要求程序员满足自己的要求:尽管存在未检查的警告，但代码是正确的，不会发生堆污染。

4.12.3 变量类型

有八种变量类型:

1. 类变量是在类声明中使用关键字 static 声明的字段(§8.3.1.1)，或者在接口声明中使用或不使用关键字 static 声明的字段(§9.3)。

类变量是在类或接口准备好(§12.3.2)并初始化为默认值(§4.12.5)时创建的。当类或接口被卸载时，类变量也不再存在(§12.7)。

2. 实例变量是在类声明中不使用关键字 static(§8.3.1.1)声明的字段。

如果一个类 T 有一个作为实例变量的字段 a，那么一个新的实例变量 a 就会被创建并初始化为一个默认值(§4.12.5)，作为每个新创建的类 T 的对象或任何类 T 的子类

(§8.1.4)的对象的一部分。当作为实例变量字段的对象不再被引用时，在对象的任何必要的终结(§12.6)完成后，实例变量实际上就不复存在了。

3. 数组组件是未命名的变量，每当创建一个新的数组对象(§10，§15.10.2)时，就会创建并初始化为默认值(§4.12.5)。当数组不再被引用时，数组组件实际上将不复存在。

4. 方法参数(§8.4.1)命名传递给方法的参数值。

对于在方法声明中声明的每个参数，每次调用该方法时都会创建一个新的参数变量(§15.12)。用方法调用中相应的参数值初始化新变量。当方法体的执行完成时，方法参数不复存在

5. 传递给构造器的构造器参数 (§8.8.1) 名和参数值。

对于在构造函数声明中声明的每个参数，每当类实例创建表达式(§15.9)或显式构造函数调用(§8.8.7)时，都会创建一个新的参数变量。用来自创建表达式或构造函数调用的相应参数值初始化新变量。当构造函数体执行完成时，构造函数参数实际上不再存在。

6. 传递给 Lambda 表达式体(§15.27.2)的 Lambda 参数(§15.27.1)名称和参数值。

对于 lambda 表达式中声明的每个参数，每次 lambda 体实现的方法调用时都会创建一个新的参数变量(§15.12)。用来自方法调用的相应参数值初始化新变量。当 lambda 表达式体执行完成时，lambda 参数实际上不再存在。

7. 每次 try 语句的 catch 子句捕获异常时，都会创建一个异常参数(§14.20)。

用与异常相关的实际对象初始化新变量(§11.3，§14.18)。当与 catch 子句相关联的块执行完成时，异常参数实际上不再存在。

8. 局部变量(§14.4)是通过语句(§14.4.2、§14.14.1、§14.14.2、§14.20.3)和模式(§14.30.1)来声明的。模式声明的局部变量称为模式变量。

当控制流进入最近的封闭块(§14.2)、for 语句或 try-with- resources 语句时，语句声明的局部变量被创建。

语句声明的局部变量在执行语句时被初始化，前提是变量的声明器有一个初始化器。明确赋值规则(§16)防止语句声明的局部变量的值在未初始化或未赋值前被使用。

模式声明的局部变量在模式匹配时创建并初始化(§14.30.2)。作用域规则(§6.3)防止模式声明的局部变量的值被使用，除非模式匹配。

当局部变量的声明不在作用域中时，它就不再存在。

如果不是因为一个例外情况，语句声明的局部变量总是被认为是在执行语句时被创建的。这种例外情况涉及到 switch 语句(§14.11)，在 switch 语句中，控制可以进入一个块，但绕过局部变量声明语句的执行。然而，由于明确赋值规则(§16)的限制，通过这种被忽略的局部变量声明语句声明的局部变量，在通过赋值表达式明确赋值之前，不能使用它(§15.26)。

例子 4.12.3-1. 不同种类变量

```
class Point {
    static int numPoints;   //  numPoints is a class variable
    int  x, y;              //  x and y are instance variables
    int[] w = new int[10];  // w[0] is an array component
    int  setX(int x)  {     //  x is a method parameter
        int oldx  =this.x;  //  oldx is a local variable
        this.x = x;
        return oldx;
    }
    boolean equalAtX(Object o) {
        if (o instanceof Point p) // p is a pattern variable return
            this.x == p.x;
        else return false;
    }
}
```


4.12.4 final 变量

变量可以声明为 final。final 变量只能赋值一次。 如果一个 final 变量被赋值，除非它在赋值之前确定没有赋值(§16(明确赋值))，否则将会出现编译时错误。

一旦 final 变量被赋值，它总是包含相同的值。 如果 final 变量保存着一个对象的引用，那么该对象的状态可以通过对该对象的操作来改变，但是该变量将始终引用同一个对象。这也适用于数组，因为数组是对象;如果 final 变量持有对数组的引用，则可以通过对数组的操作更改数组的组件，但该变量始终指向同一个数组。

空 final 是一个声明没有初始化的 final 变量。

常量变量是原生类型或 String 类型的 final 变量，用常量表达式进行初始化(§15.29)。一个变量是否为常量变量可能与类初始化(§12.4.1)、二进制兼容性(§13.1)、可访问性(§14.22)和明确赋值(§16.1.1)有关。

三种变量明确声明为 final: 接口字段 (§9.3), 在 try-with-resources 语句中声明为资源的局部变量(§14.20.3)，以及在多捕获子句中声明的异常参数(§14.20)。唯一 catch 子句的异常参数从不会隐式声明为 final，但实际上可能是 final。

例子 4.12.4-1 final 变量

声明变量 final 可以作为有用的文档，说明其值不会改变，并有助于避免编程错误。在该程序

中:

```
class Point {
    int x, y;
    int useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    static final Point origin = new Point(0, 0);
}
```

类 Point 声明一个 final 类变量 origin。origin 变量保存对对象的引用，该对象是坐标为（0，0）的 Point 类的实例。变量 Point.origin 的值不会改变, 因此它总是指向同一个 Point 对象, 该对象是由它的初始化器创建的。但是，这个 Point 对象上的操作可能会改变它的状态——例如，修改它的 useCount，甚至是错误地修改它的 x 或 y 坐标。

某些未声明为 final 的变量被认为是有效的 final:

- 一个由语句声明的局部变量，其声明符有一个初始化器(§14.4)，或者一个由模式声明的局部变量(§14.30.1)，如果下列条件都为真，那么该变量就是有效的 final:
  - 它没有声明为 final。
  - 它从不出现在赋值表达式的左边 (§15.26). (注意，包含初始化器的局部变量声明符不是赋值表达式。)
  - 它从不作为前缀或后缀自增或自减操作符的操作数出现 (§15.14, §15.15).
- 由语句声明的局部变量，如果声明符没有初始化器，则该局部变量在以下条件均为真时被认为是有效的 final:
  - 它没有声明为 final。
  - 当它出现在赋值表达式的左边时，它肯定是未赋值的，而且在赋值之前也没有明确赋值;这就是说，在赋值表达式的右边(§16(明确赋值))之后，它是绝对没有赋值的，也不是明确赋值的。
  - 它从不作为前缀或后缀自增或自减操作符的操作数出现
- 为了确定方法、构造函数、lambda 表达式或异常参数(§8.4.1、§8.8.1、§9.4、§15.27.1、§14.20)是否为有效的 final，它们被当作一个声明符有一个初始化器的局部变量。

如果变量是有效的 final，那么在其声明中添加 final 修饰符将不会引入任何编译时错误。相反地，在合法程序中声明为 final 的局部变量或参数，如果去掉了 final 修饰符，就变成了有效的 final。

4.12.5 变量初始值

程序中的每个变量在使用它的值之前都必须有一个值:

- 每个类变量、实例变量或数组组件在创建时都使用默认值初始化 (§15.9, §15.10.2):
  - 对于类型 byte, 默认值是零，也就是(byte)0。
  - 对于类型 short, 默认值是零, 也就是(short)0。
  - 对于类型 int, 默认值是, 也就是 0。
  - 对于类型 long, 默认值是零, 也就是 0L。
  - 对于类型 float, 默认值是正零, 也就是 0.0f。
  - 对于类型 double, 默认值是正零, 也就是 0.0d。
  - 对于类型 char, 默认值是空字符,也就是'\u0000'。
  - 对于类型 boolean, 默认值是 false。
  - 对于所有引用类型(§4.3), 默认值是 null。
- 每个方法参数(§8.4.1)初始化为该方法的调用者(§15.12)提供的相应实参值。
- 每个构造函数形参(§8.8.1)初始化为相应的实参值，该实参值由类实例创建表达式(§15.9)或显式构造函数调用(§8.8.7)提供。
- 一个异常参数(§14.20)被初始化为表示异常的抛出对象(§11.3，§14.18)。
- 对于语句(§14.4.2、§14.14.1、§14.14.2、§14.20.3)所声明的局部变量，在使用它之前，必须先显式地给它一个值，方法是初始化(§14.4)或赋值(§15.26)，而这种方法可以用确定赋值的规则(§16)加以验证。

  模式(§14.30.1)声明的局部变量通过模式匹配(§14.30.2)隐式初始化。

例子 4.12.5-1. 变量初始值

```
class Point {
    static int npoints;
    int x, y;
    Point root;
}

class Test {
    public static void main(String[] args) {
        System.out.println("npoints=" + Point.npoints);
        Point p = new Point();
        System.out.println("p.x="  +  p.x  +  ",  p.y="  +  p.y);
        System.out.println("p.root=" + p.root);
```

```
        }
    }
```

程序打印:

```
npoints=0
p.x=0, p.y=0
p.root=null
```

说明了 npoints 的默认初始化，它发生在准备类 Point 的时候(§12.3.2)，以及 x、y 和 root 的默认初始化，它发生在新 Point 实例化的时候。关于类和接口的加载、链接和初始化的所有方面的完整描述，以及创建新的类实例的类的实例化的描述，请参见§12(执行)。

4.12.6 类型、类和接口

在 Java 编程语言中，每个变量和每个表达式都有一个可以在编译时确定的类型。类型可以是原生类型或引用类型。引用类型包括类类型和接口类型。 引用类型通过类型声明引入，类型声明包括类声明(§8.1)和接口声明(§9.1)。 我们经常使用术语类型来指代类或接口。

在 Java 虚拟机中，每个对象都属于某个特定的类:是在创建表达式中提到的产生该对象的类(§15.9)，或者是调用类的 Class 对象的反射方法来产生该类的对象，或者是字符串连接操作符+隐式创建的 String 类(§15.18.1)的对象。这个类被称为对象的类。 对象被称为其类及其所有超类的实例。

每个数组也有一个类。当调用数组对象的 getClass 方法时，将返回一个表示该数组所属类的类对象(Class 类的)(§10.8)。

变量的编译时类型总是声明的，表达式的编译时类型可以在编译时推导出来。编译时类型限制变量在运行时可以保存的值或表达式在运行时可以产生的值。如果运行时值是一个非空的引用，则它引用具有类的对象或数组，而该类必须与编译时类型兼容。

即使变量或表达式可能具有接口类型的编译时类型，也没有接口的实例。一个类型为接口类型的变量或表达式可以引用任何实现该接口的类(§8.1.5)的对象。

有时，一个变量或表达式被称为具有“运行时类型”。 这将引用变量或表达式在运行时的值所引用的对象的类，前提是该值不为空。

编译时类型和运行时类型之间的对应是不完整的，原因有二:

1. 在运行时，Java 虚拟机使用类加载器加载类和接口。每个类加载器都定义自己

的一组类和接口。因此，两个加载器可能加载相同的类或接口定义，但在运行时产生不同的类或接口。因此，如果类加载器不一致，编译正确的代码链接时可能失败。

参见论文《Java 虚拟机动态类型加载》，由 Sheng Liang 和 Gilad Bracha，在 OOPSLA '98 的会议记录，发表在 ACM SIGPLAN 通知，卷 33，第 10 号，1998 年 10 月，第 36-44 页，和 Java 虚拟机规范，Java SE 19 版的更多细节。

2. 类型变量(§4.4)和类型参数(§4.5.1)在运行时不会具体化。因此，同一个类或接口在运行时表示从编译时开始的多个参数化类型(§4.5)。具体来说，给定泛型类型(§8.1.2，§9.1.2)的所有编译时参数化共享一个单一的运行时表示。

在某些条件下，参数化类型的变量可能引用不属于该参数化类型的对象。这种情况被称为堆污染(§4.12.2)。变量将始终引用一个对象，该对象是表示参数化类型的类的实例。

例子 4.12.6-1. 变量的类型与对象的类

```
interface Colorable {
   void setColor(byte r, byte g, byte b);
}

class Point { int x, y; }

class ColoredPoint extends Point implements Colorable {
   byte r, g, b;
   public void setColor(byte rv, byte gv, byte bv) {
      r = rv; g = gv; b = bv;
   }
}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        p = cp;
        Colorable c = cp;

        }
     }
```

在这个例子中:

• 类 Test 的方法 main 的局部变量 p 的类型为 Point，并且被赋值为一个对类 Point 的新实例的引用。

• 局部变量 cp 的类型为 ColoredPoint，并且被赋值为一个对 ColoredPoint 类的新实例的引用。

• 将 cp 赋值给变量 p 导致 p 持有了一个 ColoredPoint 对象的引用。这是允许的，因为 ColoredPoint 是 Point 的子类，所以 ColoredPoint 类和 Point 类型是赋值兼容的(§5.2)。ColoredPoint 对象包含对 Point 的所有方法的支持。除了它的特定字段 r, g 和 b 之外，它

还有类 Point 的字段，即 x 和 y。

• 局部变量 c 有它自己的类型以及接口类型 Colorable ，因此它能持有任何实现 Colorable 接口的类对象的引用； 特别地，它能持有 ColoredPoint 的引用。

形如 new Colorable()的表达式是非法的，因为不能创建接口的实例，只能创建类的实例。然而，表达式 new Colorable() { public void setColor... } 是合法的，因为它声明了实现 Colorable 接口的匿名类 (§15.9.5)。