循环与解构：

循环：

while 和do While：

其实这个跟java差不多了，如下：

```
fun whileLoop(){
    var count = 0;
    while (count<5){
        count++;
        println(count)
    }
}

fun dowhileLoop() {
    var count = 0;
    do {
        println(count);
        count++;
    } while (count <=5)
}
```

循环没有java的for in的接结构体，最常用的几种形式如下：

for In：

```
fun loop1() {
    for (i in 1..10) {
        println(i);
    }
}
```

类似于java的for in循环，如上例子，就是输出1~10区间的10个元素。再比如有基于它的封装：

```
fun loop2() {
    for (i in 1 until 10) {
        println(i)
    }
}
```

这里会输出 1~9 9个元素，until（直到）关键字，方法内部会调用 .. 方法，最终都会调用到内部的rangeTo方法：

```
public operator fun rangeTo(other: Int): IntRange
```

如果想递减输出10~1怎么办呢？ 则用这样的关键字（downTo）：

```
fun loop3() {
    for (i in 10 downTo 1) {
        println(i)
    }
}
```

如果想类似于java的 i+2呢？ kotlin为我们提供了step关键字，让我们可以自定义间隔，下面例子我们让间隔变为2：

```
  fun loop4() {
    for (i in 1..10 step 2) {
        println(i)
    }
    //或
    for (i in 10 downTo 1 step 2) {
        println(i)
    }
}
```

还有更简单的循环写法：

```
fun loop5() {
    repeat(10) {
        println(it)
    }
}
```

内部封装了 until区间，所以结果是输出1~9.

for循环常用来遍历list，我们写个小例子：

```
fun loop6() {
    val list = arrayListOf<String>("a", "b", "c", "d")
    for (str in list) {
        println(str)
    }
}
```

那如何获取下标呢？这个我们在解构中在细讲。

循环的嵌套：

先写一个父级1~2 ，子级1~2的循环：

```
fun loopWithLoop() {
    for (i in 1..2) {
        println(" ${i}父级开始")
        for (j in 1..2) {
            println("${j}子级开始")
            println("${j}子级结束")
        }
        println(" ${i}父级结束")
    }
}
```

输出结果为：

```
 1父级开始
1子级开始
1子级结束
2子级开始
2子级结束
 1父级结束
 2父级开始
1子级开始
1子级结束
2子级开始
2子级结束
 2父级结束
```

break语句：执行该语句的时候会结束循环：

下面我在子级循环中加一个break语句：

```
for (j in 1..2) {
            println("${j}子级开始")
            if(i>1){
                break
            }
            println("${j}子级结束")
        }
```

输出结果为：

```
 1父级开始
1子级开始
1子级结束
2子级开始
2子级结束
 1父级结束
 2父级开始
1子级开始
 2父级结束
```

显然，在父级的第二次循环开始的时候，子循环直接满足条件结束。

continue语句：同java 忽略本次循环剩下的语句：

下面我们把上面的break直接换成 continue： 输出结果为：

```
  1父级开始
1子级开始
1子级结束
2子级开始
2子级结束
  1父级结束
  2父级开始
1子级开始
2子级开始
  2父级结束
```

显然，在满足条件了以后，子循环还是在继续，只是没有了判断条件以后的语句打印。

retuan语句：同java 执行该语句的时候直接结束方法：

改为return 以后结果为：

```
  1父级开始
1子级开始
1子级结束
2子级开始
2子级结束
  1父级结束
  2父级开始
1子级开始
```

显然，当满足条件以后，一切都安静了。

kotlin中还能这么用：

我给外层的父循环加一个标签，在使用上述操作符的时候，就可以直接操作父类循环了：

```
fun loopWithLoop() {
    parent@ for (i in 1..2) {
        println(" ${i}父级开始")
        for (j in 1..2) {
            println("${j}子级开始")
            if (i > 1) {
                //让外层循环直接结束
                break@parent
            }
            println("${j}子级结束")
        }
        println(" ${i}父级结束")
    }
}
```

输出结果为：

```
 1父级开始
1子级开始
1子级结束
2子级开始
2子级结束
 1父级结束
 2父级开始
1子级开始
```

解构：

kotlin解构，可以让我们将一个变量分别拆解成多个变量分别赋值：假如我有一个User类，里面有两个变量：

```
class User(var age: Int, var name: String) {
    operator fun component1() = age
    operator fun component2() = name
}
```

通过operator fun component1() 的声明可以将该变量解构出去，我们在代码中这么用：

```
fun main(args: Array<String>) {
    var man = User(48, "pony")
    //解构，声明了几个，参数就能拿到几个
    val (age, name) = man;
    println(age)
    println(name)
}
```

我们看看反编译的代码实现：

```
public final class User {
   private int age;
   @NotNull
   private String name;

   public final int component1() {
      return this.age;
   }

   @NotNull
   public final String component2() {
      return this.name;
   }

   //.....

   public User(int age, @NotNull String name) {
      Intrinsics.checkParameterIsNotNull(name, "name");
      super();
      this.age = age;
      this.name = name;
   }
}
  public static final void main(@NotNull String[] args) {
      Intrinsics.checkParameterIsNotNull(args, "args");
      User man = new User(48, "pony");
      //实际上是这样
      int age = man.component1();
      String name = man.component2();
      System.out.println(age);
      System.out.println(name);
   }
```

其实就是在类中声明了名为component的方法，返回相应的属性。

应用：

常常用于集合类的遍历取值：

遍历map：

```
fun loop7() {
    val map = mapOf<String, Int>("key1" to 1, "key2" to 2)
    for ((key, value) in map) {
        println("key:${key} value${value}")
    }
}
```

遍历List：

```
  val list = arrayListOf<String>("a", "b", "c", "d")
    for ((index, str) in list.withIndex()) {
        println("第${index}个元素是${str}")
    }
```

list中，我们遍历的时候，通过解构，就能同时拿到元素和对应的下标了，注意 list 后面需要接withIndex方法，这个方法返回 IndexedValue的一个迭代器，看其内部源码，果然就是跟我们定义的解构方式一样：

```
public final data class IndexedValue<out T> public constructor(index:
kotlin.Int, value: T) {
    public final val index: kotlin.Int /* compiled code */

    public final val value: T /* compiled code */

    public final operator fun component1(): kotlin.Int { /* compiled code
*/ }

    public final operator fun component2(): T { /* compiled code */ }
}
```