尚硅谷_JavaSE_day23

学习目标

- [ ] 了解Java9-Java17的新特性

Java9-17新特性

1 JDK版本的选择

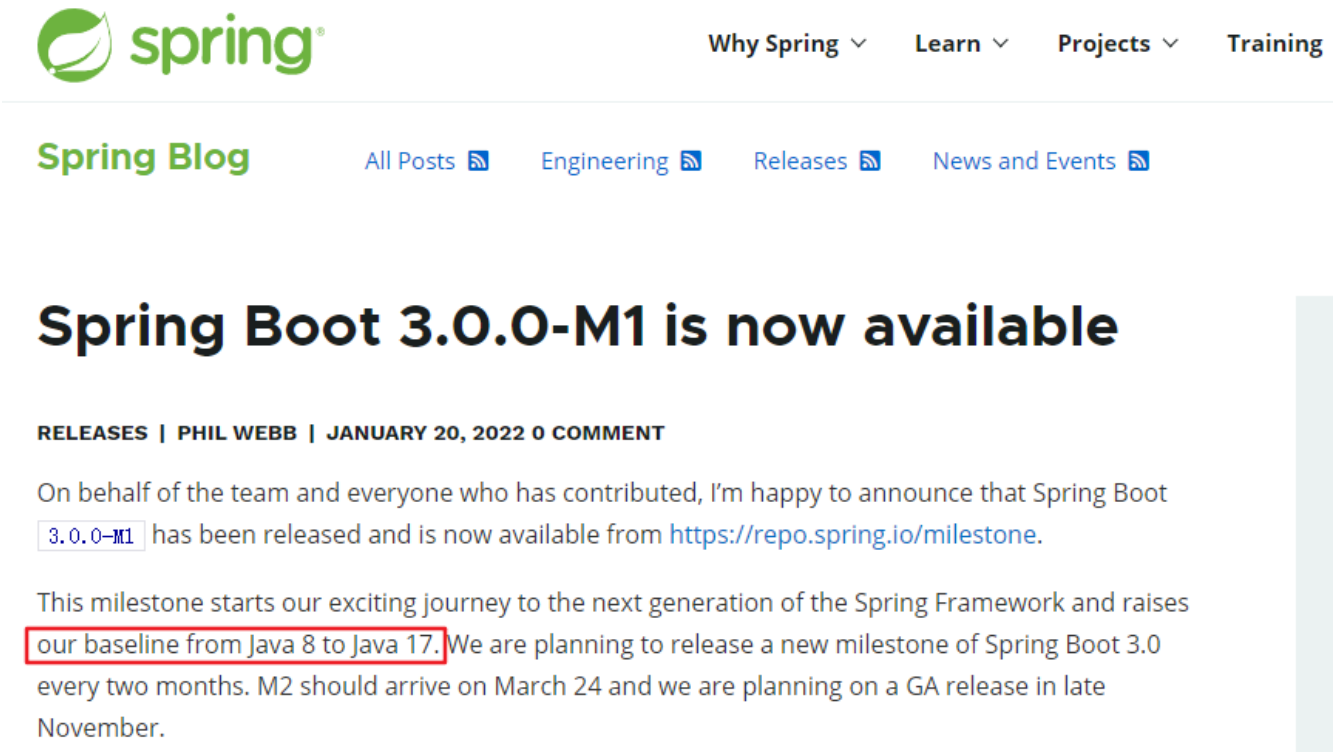
历经曲折的Java 9在4次跳票后，终于在2017年9月21日发布。从Java 9这个版本开始，Java 的计划发布周期是6个月，这意味着Java的更新从传统的以特性驱动的发布周期，转变为以时间驱动的发布周期，并逐步地将Oracle JDK原商业特性进行开源。针对企业客户的需求，Oracle将以3年为周期发布长期支持版本（Long Term Support，LTS），最近的LTS版本就是Java 11和Java17了，其他都是过渡版本

在Java 17正式发布之前，Java开发框架Spring率先在官博宣布，Spring Framework 6和Spring Boot 3计划在2022年第四季度实现总体可用性的高端基线：

1、Java 17+(来自 Spring Framework 5.3.x 线中的 Java 8-17)

2、Jakarta EE 9+（来自Spring框架5.3.x 线中的 Java EE 7-8）

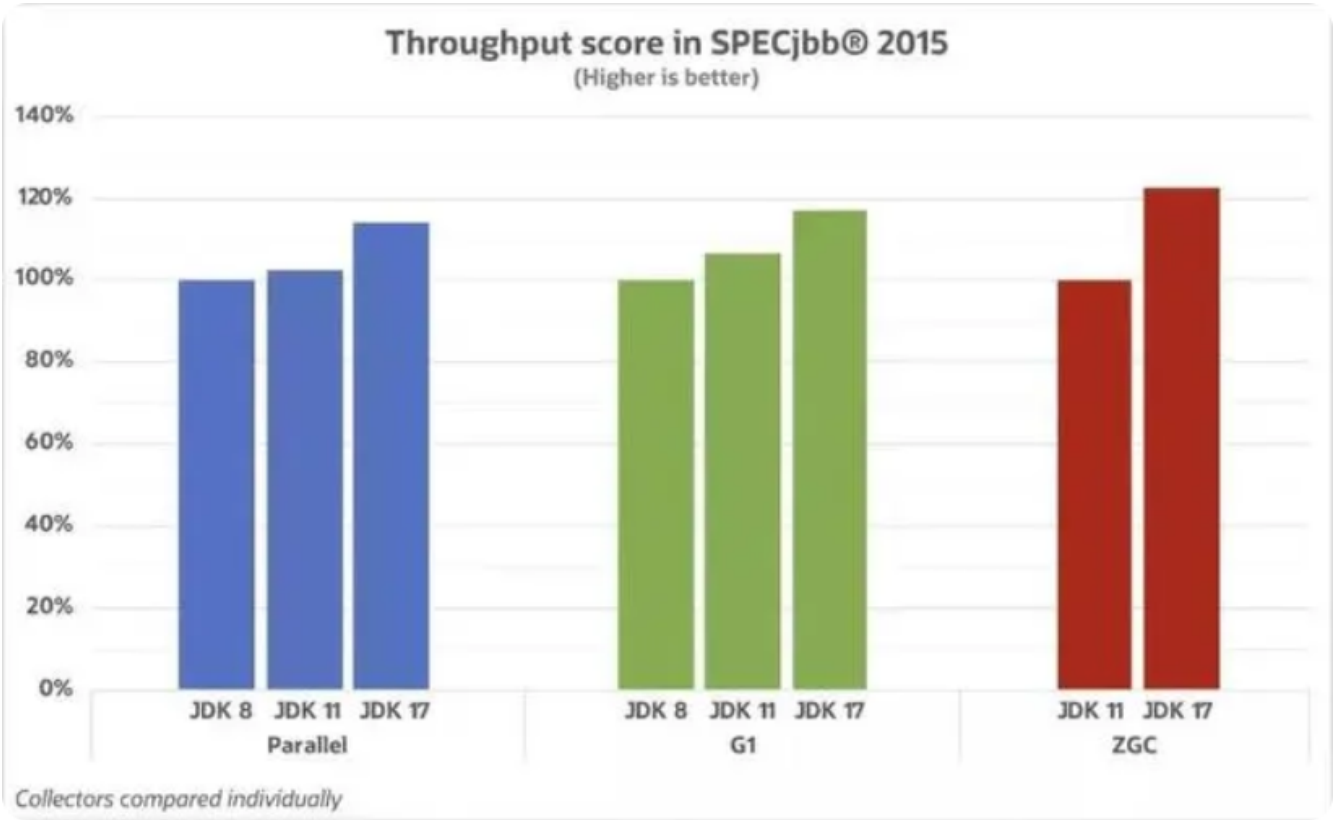
3.Spring 官方说明：`https://spring.io/blog/2022/01/20/spring-boot-3-0-0-m1-is-now-available`



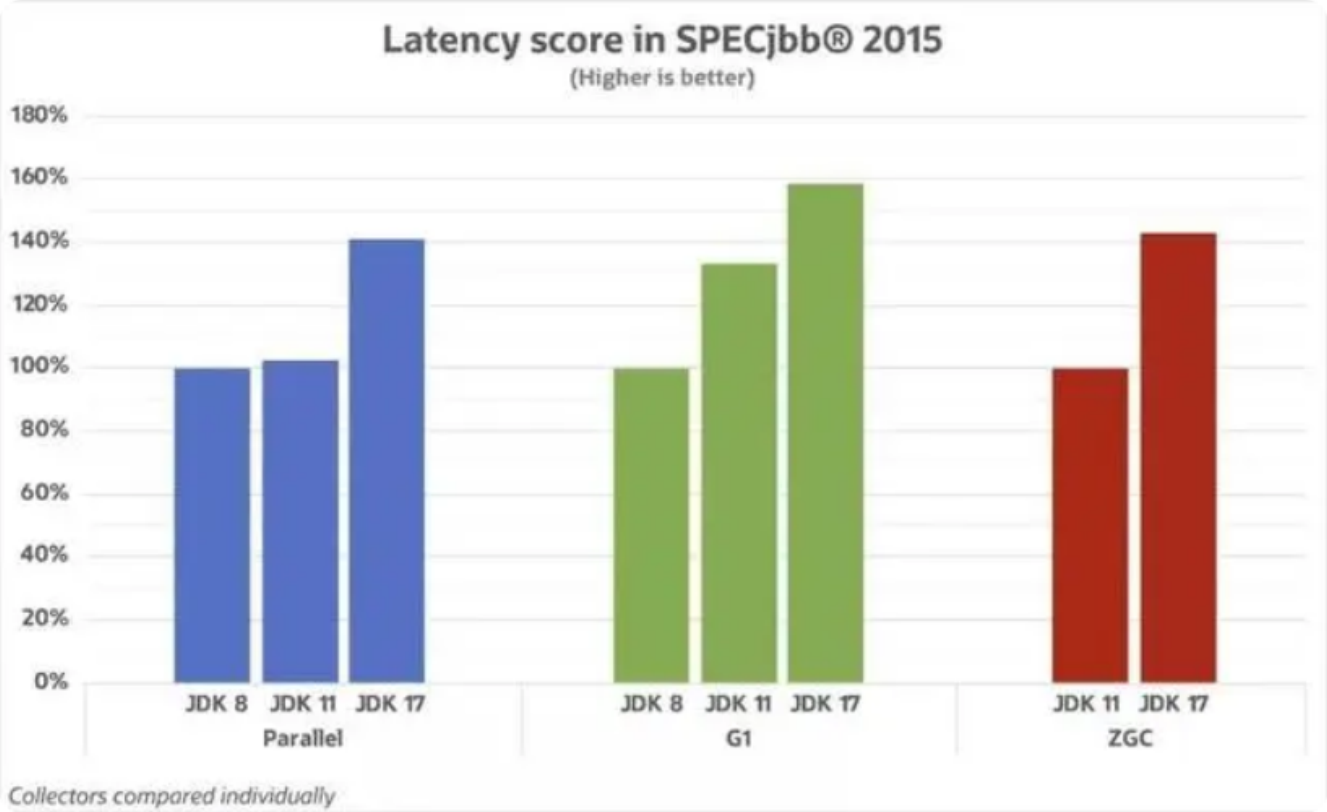
> 意味着：Springboot3.0 是需要用Java17和Spring6.0为基础建设。如果从企业选型最新Springboot3.0作为架构来说，它搭配jdk17肯定是标配了。 针对于Spring 6，官网的说明会弃用java8以9为最低版本，而且兼容tomcat10+。

4.JDK17针对于GC方面作出了优化,以及做了性能的提高

a.在吞吐量方面，Parallel 中 JDK 8 和 JDK 11 差距不大，JDK 17 相较 JDK 8 提升 15% 左右；G1 中 JDK 17 比 JDK 8 提升 18%；ZGC 在 JDK 11引入，JDK 17 对比JDK 11 提升超过 20%



b. 在 GC 延迟方面，JDK 17 的提升更为明显。在 Parallel 中 JDK 17 对比 JDK 8 和JDK 11 提升 40%；在 G1 中，JDK 11 对比 JDK 8 提升 26%，JDK 17 对比 JDK 8 提升接近 60%！ZGC 中 JDK 17 对比 JDK 11 提升超过 40%



> 从GC性能角度去看，JDK 11对比JDK 8延迟提升不到40%；反观JDK 17对比JDK 8 延迟提升 60%，吞吐量提升18%；可以看到JDK17的提升还是非常明显的

由于JDK对性能提升方面都是自动的，所以我们可以直接学习JDK新特性中的语法和API。我们要知道的是下面的语法不都是从JDK17才开始有的，但是JDK17都支持这些语法和API。

2 接口的私有方法

Java8版本接口增加了两类成员：

- 公共的默认方法
- 公共的静态方法

Java9版本接口又新增了一类成员：

- 私有的方法

为什么JDK1.9要允许接口定义私有方法呢？因为我们说接口是规范，规范时需要公开让大家遵守的

私有方法：因为有了默认方法和静态方法这样具有具体实现的方法，那么就可能出现多个方法由共同的代码可以抽取，而这些共同的代码抽取出来的方法又只希望在接口内部使用，所以就增加了私有方法。

```
package com.atguigu.inter;

public interface MyInter{
    private void m1(){
        System.out.println("接口中的私有方法");
    }
    private static void m2(){
        System.out.println("接口中的私有静态方法");
    }
}
```

3 钻石操作符与匿名内部类结合

自Java 9之后我们将能够与匿名实现类共同使用钻石操作符，即匿名实现类也支持类型自动推断

```
package com.atguigu.anonymous;

import java.util.Arrays;
import java.util.Comparator;

public class TestAnonymous {
    public static void main(String[] args) {
        String[] arr = {"hello","Java"};
        Arrays.sort(arr,new Comparator<>() {
            @Override
            public int compare(String o1, String o2) {
                return o1.compareToIgnoreCase(o2);
            }
        });
    }
}
```

Java 8的语言等级编译会报错：“'<>' cannot be used with anonymous classes。”Java 9及以上版本才能编译和运行正常。

4 try..catch升级

JDK 1.7引入了trywith-resources的新特性，可以实现资源的自动关闭，此时要求：

- 该资源必须实现java.io.Closeable接口
- 在try子句中声明并初始化资源对象
- 该资源对象必须是final的

```
try(IO流对象1声明和初始化;IO流对象2声明和初始化){
    可能出现异常的代码
}catch(异常类型 对象名){
  异常处理方案
}
```

JDK1.9又对trywith-resources的语法升级了

- 该资源必须实现java.io.Closeable接口
- 在try子句中声明并初始化资源对象，也可以直接使用已初始化的资源对象

- 该资源对象必须是final的

```
IO流对象1声明和初始化;
IO流对象2声明和初始化;

try(IO流对象1;IO流对象2){
    可能出现异常的代码
}catch(异常类型 对象名){
  异常处理方案
}
```

```
package com.atguigu.trycatch;

import org.junit.Test;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;

public class TestTryCatch {

    /**
     * jdk7之前的IO异常处理方案
     */
    @Test
    public void test1() {
        FileWriter fw = null;
        try {
            fw = new FileWriter("io\\1.txt");
            fw.write("张三");
        } catch (IOException e) {
            throw new RuntimeException(e);
        } finally {
            if (fw != null) {
                try {
                    fw.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    /**
     * jdk7开始有的下面这种IO流异常处理方式
     * try(IO流对象1声明和初始化;IO流对象2声明和初始化){
     *     可能出现异常的代码
     * }catch(异常类型 对象名){
     *     异常处理方案
     * }
     *
     * jvm会自动刷新和关闭流对象
```

```
     */
    @Test
    public void test2(){
        try (FileWriter fw = new FileWriter("io\\2.txt");) {
            fw.write("张三");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Java 9 中，用资源语句编写try将更容易，我们可以在try子句中使用已经初始化过的资源
     * IO流对象1声明和初始化;
     * IO流对象2声明和初始化;
     * try(IO流对象1;IO流对象2){
     *     可能出现异常的代码
     * }catch(异常类型 对象名){
     *     异常处理方案
     * }
     *
     * jvm会自动刷新和关闭流对象
     */
    public static void test3()throws Exception{
        FileInputStream fis = new FileInputStream("io\\1.jpg");
        FileOutputStream fos = new FileOutputStream("io\\2.jpg");
        try(fis;fos){
            byte[] bytes = new byte[1024];
            int len;
            while((len = fis.read(bytes))!=-1){
                fos.write(bytes,0,len);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5 局部变量类型自动推断

jdk10之前，我们定义局部变量都必须要明确数据的数据类型，但是到了JDK10，出现了一个最为重要的特性，就是局部变量类型推断，顾名思义，就是定义局部变量时，不用先确定具体的数据类型了，可以直接根据具体数据推断出所属的数据类型。

```
  var 变量名 = 值;
```

```
package com.atguigu.var;

import java.util.ArrayList;
import java.util.Arrays;

/**
 * Description:
```

```
 *
 * @Author atguigu
 * @Create 2022/11/5 14:49
 * @Version 1.0
 */
public class TestVariable {
    public static void main(String[] args) {
        var a = 1;
        System.out.println("a = " + a);

        var s = "hello";
        System.out.println("s = " + s);

        var d = Math.random();
        System.out.println("d = " + d);

        var list = Arrays.asList("hello","world");
        for (var o : list) {
            System.out.println(o);
        }
    }
}
```

6 switch表达式

switch表达式在Java 12中作为预览语言出现，在Java 13中进行了二次预览，得到了再次改进，最终在Java 14中确定下来。另外，在Java17中预览了switch模式匹配。

传统的switch语句在使用中有以下几个问题。

（1）匹配是自上而下的，如果忘记写break，那么后面的case语句不论匹配与否都会执行。

（2）所有的case语句共用一个块范围，在不同的case语句定义的变量名不能重复。

（3）不能在一个case语句中写多个执行结果一致的条件，即每个case语句后只能写一个常量值。

（4）整个switch语句不能作为表达式返回值。

1、Java12的switch表达式

Java 12对switch语句进行了扩展，将其作为增强版的switch语句或称为switch表达式，可以写出更加简化的代码。

- 允许将多个case语句合并到一行，可以简洁、清晰也更加优雅地表达逻辑分支。
- 可以使用-> 代替：
    - ->写法默认省略break语句，避免了因少写break语句而出错的问题。
    - ->写法在标签右侧的代码段可以是表达式、代码块或 throw语句。
    - ->写法在标签右侧的代码块中定义的局部变量，其作用域就限制在代码块中，而不是蔓延到整个switch结构。
- 同一个switch结构中不能混用“→”和“:”，否则会有编译错误。使用字符“:”，这时fall-through规则依然有效，即不能省略原有的break语句。": "的写法表示继续使用传统switch语法。

案例需求：

请使用switch-case结构实现根据月份输出对应季节名称。例如，3～5月是春季，6～8月是夏季，9～11月是秋季，12～2月是冬季。

Java12之前写法：

```
@Test
    public void test1() {
        int month = 3;
        switch (month) {
            case 3:
            case 4:
            case 5:
                System.out.println("春季");
                break;
            case 6:
            case 7:
            case 8:
                System.out.println("夏季");
                break;
            case 9:
            case 10:
            case 11:
                System.out.println("秋季");
                break;
            case 12:
            case 1:
            case 2:
                System.out.println("冬季");
                break;
            default:
                System.out.println("月份输入有误！");
        }
    }
```

Java12之后写法：

```
package com.atguigu.switchexp;

import org.junit.Test;

import java.util.Scanner;

public class TestSwitch12 {

    @Test
    public void test121() {
        int month = 3;
        switch(month) {
            case 3,4,5 -> System.out.println("春季");
            case 6,7,8 -> System.out.println("夏季");
            case 9,10,11 -> System.out.println("秋季");
            case 12,1,2 -> System.out.println("冬季");
            default -> System.out.println("月份输入有误！");
        }
```

```
    }

    @Test
    public void test122(){
        int month = 3;
        String monthName = switch(month) {
            case 3,4,5 -> "春季";
            case 6,7,8 -> "夏季";
            case 9,10,11 -> "秋季";
            case 12,1,2 -> "冬季";
//            default -> "error";
            default -> throw new IllegalArgumentException("月份有误! ");
        };
        System.out.println("monthName = " + monthName);
    }
    @Test
    public void test123(){
        int month = 3;
        switch(month) {
            case 3,4,5 : System.out.println("春季");//仍然会贯穿
            case 6,7,8 : System.out.println("夏季");
            case 9,10,11 : System.out.println("秋季");
            case 12,1,2 : System.out.println("冬季");
            default : System.out.println("月份输入有误! ");
        }
    }

    @Test
    public void test(){
        double balance = 0.0;
        Scanner input = new Scanner(System.in);
        boolean flag = true;
        while(flag) {
            System.out.println("1、存钱");
            System.out.println("2、取钱");
            System.out.println("3、查询");
            System.out.println("4、退出");
            System.out.print("请选择: ");
            int select = input.nextInt();
            switch (select){
                case 1->{
                    System.out.print("请输入存钱金额: ");
                    double money = input.nextDouble();
                    balance += money;
                }
                case 2->{
                    System.out.print("请输入取钱金额: ");
                    double money = input.nextDouble();
                    balance -= money;
                }
                case 3-> System.out.println("余额: " + balance);
                case 4 -> flag = false;
                default -> System.out.println("输入有误");
```

```
            }
        }
        input.close();
    }
}
```

2、Java13的switch表达式

Java 13提出了第二个switch表达式预览，引入了yield语句，用于返回值。这意味着，switch表达式（返回值）应该使用yield语句，switch语句（不返回值）应该使用break语句。

案例需求：根据星期值，获取星期名称。

Java13之前写法：

```
    @Test
    public void test2() {
        int week = 2;
        String weekName = "";
        switch (week) {
            case 1:
                weekName = "Monday";
                break;
            case 2:
                weekName = "Tuesday";
                break;
            case 3:
                weekName = "Wednesday";
                break;
            case 4:
                weekName = "Thursday";
                break;
            case 5:
                weekName = "Friday";
                break;
            case 6:
                weekName = "Saturday";
                break;
            case 7:
                weekName = "Sunday";
                break;
            default:
                System.out.println("Week number is between 1 and 7.");
                weekName = "Error";
        }
        System.out.println("weekName = " + weekName);
    }
```

Java13之后写法：

```
package com.atguigu.switchexp;

import org.junit.Test;
```

```
public class TestSwitch13 {
    @Test
    public void test1(){
        int week = 2;
        String weekName = switch(week) {
            case 1 -> "Monday";
            case 2 -> "Tuesday";
            case 3 -> "Wednesday";
            case 4 -> "Thursday";
            case 5 -> "Friday";
            case 6 -> "Saturday";
            case 7 -> {
                System.out.println("Weekend!");
                yield "Sunday";
            }
            default -> {
                System.out.println("Week number is between 1 and 7.");
                yield "Error";
            }
        };
        System.out.println("weekName = " + weekName);
    }
}
```

3、Java17的switch表达式

Java17==预览==了switch模式匹配，允许switch表达式和语句可以针对多个模式进行测试，每个模式都有特定的操作，这使得复杂的面向数据的查询能够简洁而安全地表达。

案例需求：根据传入数据的类型不同，返回不同格式的字符串。

不使用模式匹配：

```
    public static String formatterIf(Object o) {
        String str = "unknown";
        if (o instanceof Integer i) {
            str = String.valueOf(i);
        } else if (o instanceof Long l) {
            str = String.valueOf(l);
        } else if (o instanceof Double d) {
            str = String.valueOf(d);
        }
        return str;
    }
```

使用模式匹配：

```
    public static String formatterSwitch(Object o) {
        return switch (o) {
            case Integer i -> String.valueOf(i);
            case Long l  -> String.valueOf(l);
            case Double d -> String.valueOf(d);
            case String s -> String.valueOf(s);
            default    -> o.toString();
        };
    }
```

直接在switch上支持Object类型，这就等于同时支持多种类型，使用模式匹配得到具体类型，大大简化了代码量，这个功能还是挺实用的，期待转正。

7 文本块

预览的新特性文本块在Java 15中被最终确定下来，Java 15之后我们就可以放心使用该文本块了。

1、Java13文本块

JDK 12引入了Raw String Literals特性，但在其发布之前就放弃了这个特性。这个JEP与引入多行字符串文字（文本块）在意义上是类似的。Java 13中引入了文本块（预览特性），这个新特性跟Kotlin中的文本块是类似的。

现实问题

在Java中，通常需要使用String类型表达HTML，XML，SQL或JSON等格式的字符串，在进行字符串赋值时需要进行转义和连接操作，然后才能编译该代码，这种表达方式难以阅读并且难以维护。

文本块就是指多行字符串，例如一段格式化后的XML、JSON等。而有了文本块以后，用户不需要转义，Java能自动搞定。因此，**文本块将提高Java程序的可读性和可写性。**

目标

- 简化跨越多行的字符串，避免对换行等特殊字符进行转义，简化编写Java程序。
- 增强Java程序中字符串的可读性。

举例

会被自动转义，如有一段以下字符串：

```
<html>
  <body>
      <p>Hello, 尚硅谷</p>
  </body>
</html>
```

将其复制到Java的字符串中，会展示成以下内容：

```
"<html>\n" +
"    <body>\n" +
"        <p>Hello, 尚硅谷</p>\n" +
"    </body>\n" +
"</html>\n";
```

即被自动进行了转义，这样的字符串看起来不是很直观，在JDK 13中，就可以使用以下语法了：

```
"""
<html>
  <body>
      <p>Hello, world</p>
  </body>
</html>
""";
```

使用“”“作为文本块的开始符和结束符，在其中就可以放置多行的字符串，不需要进行任何转义。看起来就十分清爽了。

文本块是Java中的一种新形式，它可以用来表示任何字符串，并且提供更多的表现力和更少的复杂性。

（1）文本块由零个或多个字符组成，由开始和结束分隔符括起来。

- 开始分隔符由三个双引号字符表示，后面可以跟零个或多个空格，最终以行终止符结束。
- 文本块内容以开始分隔符的行终止符后的第一个字符开始。
- 结束分隔符也由三个双引号字符表示，文本块内容以结束分隔符的第一个双引号之前的最后一个字符结束。

以下示例代码是错误格式的文本块：

```
String err1 = """""";//开始分隔符后没有行终止符

String err2 = """  """;//开始分隔符后没有行终止符
```

如果要表示空字符串需要以下示例代码表示：

```
String emp1 = "";//推荐
String emp2 = """
   """;//第二种需要两行，更麻烦了
```

（2）允许开发人员使用“\n”“\f”和“\r”来进行字符串的垂直格式化，使用“\b”“\t”进行水平格式化。如以下示例代码就是合法的。

```
String html = """
    <html>\n
      <body>\n
        <p>Hello, world</p>\n
      </body>\n
    </html>\n
    """;
```

（3）在文本块中自由使用双引号是合法的。

```
String story = """
   Elly said,"Maybe I was a bird in another life."

Noah said,"If you're a bird , I'm a bird."
 """;
```

2、Java14文本块

Java 14给文本块引入了两个新的转义序列。一是可以使用新的\s转义序列来表示一个空格；二是可以使用反斜杠“\”来避免在行尾插入换行字符，这样可以很容易地在文本块中将一个很长的行分解成多行来增加可读性。

例如，现在编写多行字符串的方式如下所示：

```
String literal = "人最宝贵的东西是生命，生命对人来说只有一次。" +
        "因此，人的一生应当这样度过：当一个人回首往事时，" +
        "不因虚度年华而悔恨，也不因碌碌无为而羞愧；" +
        "这样，在他临死的时候，能够说，" +
        "我把整个生命和全部精力都献给了人生最宝贵的事业" +
        "——为人类的解放而奋斗。";
```

在文本块中使用“\”转义序列，就可以写成如下形式：

```
String text = """
    人最宝贵的东西是生命，生命对人来说只有一次。\
    因此，人的一生应当这样度过：当一个人回首往事时，\
    不因虚度年华而悔恨，也不因碌碌无为而羞愧；\
    这样，在他临死的时候，能够说，\
    我把整个生命和全部精力都献给了人生最宝贵的事业\
    ——为人类的解放而奋斗。
    """;
```

8 instanceof模式匹配

instanceof的模式匹配在JDK14、15中预览，在JDK16中转正。有了它就不需要编写先通过instanceof判断再强制转换的代码。

案例需求：

现有一个父类Animal及它的两个子类Bird和Fish，现在要判断某个对象是Bird实例对象还是Fish实例对象，并向下转型然后调用各自扩展的方法。

```
package com.atguigu.instanceofnew;

abstract class Animal {
}
class Bird extends Animal {
    public void fly(){
        System.out.println("fly~~");
    }
}
class Fish extends Animal {
    public void swim(){
```

```
        System.out.println("swim~~");
    }
}
```

之前我们调用一个对象中的方法,我们会先判断类型,如果调用子类特有方法,我们需要向下转型,如下面代码:

```
    public static void old(Animal animal){
        if (animal instanceof Bird) {
            Bird bird = (Bird) animal;
            bird.fly();
        } else if (animal instanceof Fish) {
            Fish fish = (Fish) animal;
            fish.swim();
        }
    }
```

从JDK14开始,我们不需要单独强转,直接省略强转的过程

```
   public static void now(Animal animal) {
        if (animal instanceof Bird bird) {
            bird.fly();
        } else if (animal instanceof Fish fish){
            fish.swim();
        }
    }
```

9 Record类

Record类在JDK14、15预览特性，在JDK16中转正。

record是一种全新的类型，它本质上是一个 final类，同时所有的属性都是 final修饰，它会自动编译出get、hashCode、比较所有属性值的equals、toString 等方法，减少了代码编写量。使用 Record 可以更方便的创建一个常量类。

1.注意:

- Record只会有一个全参构造
- 重写的equals方法比较所有属性值
- 可以在Record声明的类中定义静态字段、静态方法或实例方法。
- 不能在Record声明的类中定义实例字段;
- 类不能声明为abstract;
- 不能显式的声明父类，默认父类是java.lang.Record类
- 因为Record类是一个 final类，所以也没有子类等。

```
package com.atguigu.record;

public class TestRecord {
    public static void main(String[] args) {
        Triangle t = new Triangle(3, 4, 5);
        System.out.println(t);
        System.out.println("面积: " + t.area());
        System.out.println("周长: " + t.perimeter());
        System.out.println("边长: " + t.a() + "," + t.b() + "," + t.c());

        Triangle t2 = new Triangle(3, 4, 5);
```

```
        System.out.println(t.equals(t2));
    }
}

record Triangle(double a, double b, double c) {
    public double area() {
        if (a > 0 && b > 0 && c > 0 && a + b > c && b + c > a && a + c > b) {
            double p = (a + b + c) / 2;
            return Math.sqrt(p * (p - a) * (p - b) * (p - c));
        }
        throw new IllegalArgumentException("不是合法的三角形");
    }

    public double perimeter() {
        return a + b + c;
    }
}
```

10 密封类

其实很多语言中都有 密封类 的概念，在Java语言中,也早就有 密封类 的思想，就是final修饰的类，该类不允许被继承。而从JDK15开始,针对 密封类 进行了升级。

Java 15通过密封的类和接口来增强Java编程语言，这是新引入的预览功能并在Java 16中进行了二次预览，并在Java17最终确定下来。这个预览功能用于限制超类的使用，密封的类和接口限制其他可能继承或实现它们的其他类或接口。

```
【修饰符】 sealed class 密封类 【extends 父类】【implements 父接口】 permits 子类{

}
【修饰符】 sealed interface 接口 【extends 父接口们】 permits 实现类{

}
```

- 密封类用 sealed 修饰符来描述，
- 使用 permits 关键字来指定可以继承或实现该类的类型有哪些
- 一个类继承密封类或实现密封接口，该类必须是sealed、non-sealed、final修饰的。
- sealed修饰的类或接口必须有子类或实现类

```
package com.atguigu.sealed;

import java.io.Serializable;

sealed class Graphic /*extends Object implements Serializable*/ permits Circle,Rectangle,
Triangle {

}
final class Triangle extends Graphic{

}
non-sealed class Circle extends Graphic{

}
```

```
sealed class Rectangle extends Graphic permits Square{

}
final class Square extends Rectangle{

}
```

```
package com.atguigu.sealed;

import java.io.Serializable;

public class TestSealedInterface {
}
sealed interface Flyable /*extends Serializable*/ permits Bird {

}
non-sealed class Bird implements Flyable{

}
```

11 其他

陆续在新版本变化的API有很多，因篇幅问题不能一一列举。

Java 9带来了很多重大的变化，其中最重要的变化是Java平台模块系统的引入。众所周知，Java发展已经超过20年，Java和相关生态在不断丰富的同时也越来越暴露出一些问题。

（1）当某些应用很简单时。夸张地说，如若仅是为了打印一个“helloworld”，那么之前版本的JRE中有一个很重要的rt.jar（如Java 8的rt.jar中有60.5M），即运行一个“helloworld”，也需要一个数百兆的JRE环境，而这在很多小型设备中是很难实现的。

（2）当某些应用很复杂，有很多业务模块组成时。我们以package的形式设计和组织类文件，在起初阶段还不错，但是当我们有数百个package时，它们之间的依赖关系一眼是看不完的，当代码库越来越大，创建越复杂，盘根错节的“意大利面条式代码”的概率会呈指数级增长，这给后期维护带来了麻烦，而可维护性是软件设计和演进过程中最重要的问题。

（3）一个问题是classpath。所有的类和类库都堆积在classpath中。当这些JAR文件中的类在运行时有多个版本时，Java的ClassLoader就只能加载那个类的某个版本。在这种情形下，程序的运行就会有歧义，有歧义是一件非常坏的事情。这个问题总是频繁出现，它被称为“JAR Hell”。

（4）很难真正对代码进行封装，而系统并没有对不同部分（也就是 JAR 文件）之间的依赖关系有明确的概念。每个公共类都可以被类路径下的其他类访问到，这样就会在无意中使用了并不想被公开访问的API。

模块就是为了修复这些问题存在的。模块化系统的优势有以下几点。

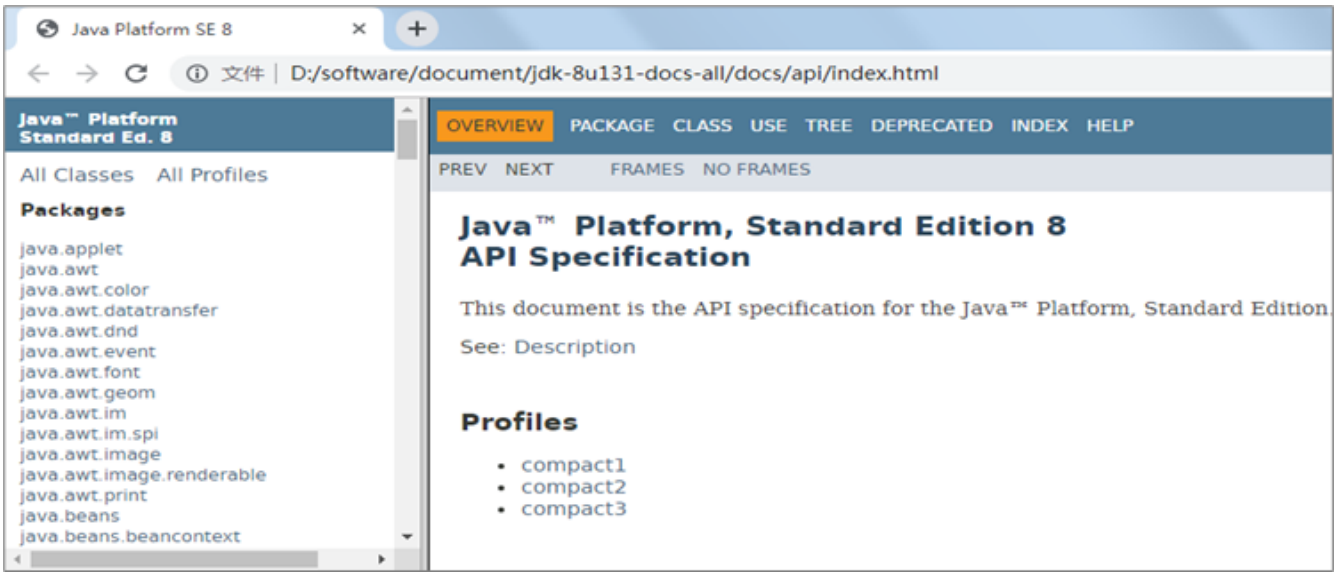
- 模块化的主要目的是减少内存的开销。
- 只需要必要模块，而非全部JDK模块，可简化各种类库和大型应用的开发和维护。
- 改进Java SE平台，使其可以适应不同大小的计算设备。
- 改进其安全性、可维护性。用模块管理各个package，其实就是在package外再裹一层，可以通过声明暴露某个package，不声明默认就是隐藏。因此，模块化系统使代码组织上更安全，因为它可以指定哪些部分可以暴露，哪些部分需要隐藏。
- 更可靠的配置，通过明确指定的类的依赖关系代替以前易错的路径（class-path）加载机制。模块必须声明对其他模块的显示依赖，并且模块系统会验证应用程序所有阶段的依赖关系：编译时、链接时和运行时。假设一个

模块声明对另一个模块的依赖，并且第二个模块在启动时丢失，JVM检测到依赖关系丢失，在启动时就会失败。在Java 9之前，当使用缺少的类型时，这样的应用程序只会生成运行时错误而不是启动时错误。

Java 9是一个庞大的系统工程，从根本上带来了一个整体改变，包括JDK的安装目录，以适应模块化的设计。

大家可以发现在Java 9之后，API的组织结构也变了。

原来Java 8的API，包是顶级的封装，Java 8的API结构如图所示。



而Java 9的API，模块是顶级的封装，模块之下才是包，如java.base模块，Java 9的API中Java SE部分的模块结构如图所示。