

## 3.3 Java 进阶功能

---

- 修饰符
- 文件读写
- 线程管理
- 正则表达式
- 其他常用 API



# 目录

1

修饰符

2

文件读写

3

线程管理

4

正则表达式

5

其他常用 API

# 修饰符

- **修饰符**<sup>[修飾子]</sup>可以在我们定义某个**变量、方法或类**（或代码块）时用来为其增加某种特性。
- 比如，我们之前学过的 **public**、**private** 等被称为 *访问修饰符*；**static** 是表示静态的修饰符。我们可以将这些修饰符写在某个定义之前以表示其特性。
- Java 中的常见修饰符如下：
  - 访问修饰符：private、protected、public；
  - abstract
  - static
  - final

## Note !

同一个修饰符，修饰变量（包括成员变量或局部变量）、方法或类可能会有不同的作用。

# abstract 修饰符

- **abstract 修饰符**用于表达修饰目标是“抽象的”。
- abstract 修饰符修饰类时，表示该类是**抽象类**，不能实例化，有一些方法需要子类实装。
- abstract 修饰符修饰方法时，表示该方法是**抽象方法**，没有实装，需要子类重写实装。
- abstract 修饰符不能修饰变量。



# static 修饰符

- **static 修饰符**用于表达修饰目标是“静态的”。
- 如果 static 修饰符修饰一个成员变量、方法或内部类，则表示它们是**静态的**变量、方法或类，属于某一个**类**而不是某一个特定的**对象**。
- static 修饰符不能修饰局部变量。
- static 修饰符修饰的类代码块会在类初次被访问时执行。

# final 修饰符

- **final** 修饰符用于表达修饰的目标是“不可变的”。
- 如果 final 修饰符修饰某个变量，则这个变量只能被赋值 **1** 次。再次尝试赋值会导致语法错误。
  - 但是修饰局部、成员或静态变量会有一些小区别，我们之后会介绍。
- 如果 final 修饰符修饰某个方法，则表示该方法不能被子类**重写**。
  - final 不能修饰抽象方法。
- 如果 final 修饰符修饰某个类，则表示该类不能被继承。

# final 局部变量

- **final** 修饰的局部变量只能被赋值 1 次，**包括**声明时的初始化。
  - 如果声明时初始化了，那么此后就不能赋值。
  - 如果声明时没有初始化，那么此后只能赋 1 次值。
- 如果尝试改变一个已经被赋过值的 final 变量会产生语法错误：

```
1 final int a = 10;  
2 a = 20; // => 报错  
3 final int b;  
4 b = 20;  
5 b = 3; // => 报错
```

# final 成员变量

- **final** 修饰的成员变量除了像局部变量一样只能被赋值 1 次之外，还有一个特殊要求：它必须在**构造器结束前**被赋值。
- 也就是说，要么在初始化时给它赋值；要么必须在**所有构造器**里给它赋值。否则，会产生语法错误：

```
1 class A {  
2     final int a;  
3  
4     A() {  
5         a = 0; // 如果去掉这行会报错  
6     }  
7 }
```



# final 静态变量

- **final** 修饰的静态变量除了不能被改变之外，还必须在初始化时（或在静态代码块）被赋值。
- 在 Java 中，经常使用 final static 的变量表示**常量**[定数]。

```
1 public class Math {  
2     public final static double PI = 3.1415927;  
3     public final static double E = 2.7182818;  
4 }
```

## Note



Java 常量通常使用  
大写蛇形命名。

# 其他修饰符

修饰符	修饰对象	效果
strictfp	类、方法	严格按照 IEEE 754 的标准进行浮点数运算。
transient	成员变量	在序列化时不会被写入序列。
synchronized	类、方法、代码块	代码不会被多个线程同时执行。
volatile	成员变量	线程不会缓存该变量，而是直接在主存中存取。
native	方法	该方法由其他语言实现。
default	接口中的方法	该方法提供默认实现。

# 修饰符的书写顺序

- 在一个声明有多个修饰符时，**先写访问修饰符**，再写其他修饰符。  
(语法上也可以写在后面，但是不建议这么写。) 除了访问修饰符外的修饰符可以按**任意顺序**书写。相同修饰符**不能重复**。

## Example ✓

```
private final static int a;  
private static final int b;  
public strictfp abstract void c();
```

## Example ✗

```
static public int a;  
final static final int b;
```



Q & A

*Question and answer*

# 目录

1

修饰符

2

文件读写

3

线程管理

4

正则表达式

5

其他常用 API

# 文件读写

- 我们之前学习了在控制台中读写的方法（`System.out.println`）。但在实际的程序中，用户不会直接使用控制台访问程序。我们经常需要直接读写**硬盘**上的文件。
- Java 为我们提供了读写文件的 **API**。我们可以分成两个部分学习：
  1. 管理文件的 API，用于寻找、打开某个硬盘上的文件；
    - `java.io.File`。
  2. 其次是读写文件的 API，用于向打开的文件里写入数据或读取数据。
    - `java.io.InputStream`、`java.io.OutputStream`。



# File

- Java 提供了 **File** 作为代表一个文件的类。
- File 类同时也能用于代表硬盘中的一个目录[ディレクトリ]（文件夹[フォルダ]）。
- 提供文件的创建、查找、删除等基本文件操作功能。

# File 对象的创建

- 要创建一个文件类，直接使用 File 类的构造器：

```
File file = new File("test.txt");
```

- 构造器接受一个字符串作为参数，代表文件的**路径**[パス]（目录 + 文件名）。
- 可以接受**绝对或相对路径**。

# 绝对路径和相对路径

- **绝对文件路径**[絶対パス]是文件的完整路径，从盘符（Windows）或根目录（macOS）开始：
  - D:\class\java\test.txt
  - ~\class\java\test.txt（在 Java 里可以写 /class/java/test.txt）
- **相对文件路径**[相対パス]是相对于当前程序的文件路径（在 eclipse 中是相对于工程根目录）：
  - test.txt
  - java\test.txt
  - ../test.txt

## Note



不要忘了在 Java 字符串里的“\”符号需要转义！



# 相对路径

## Case 1

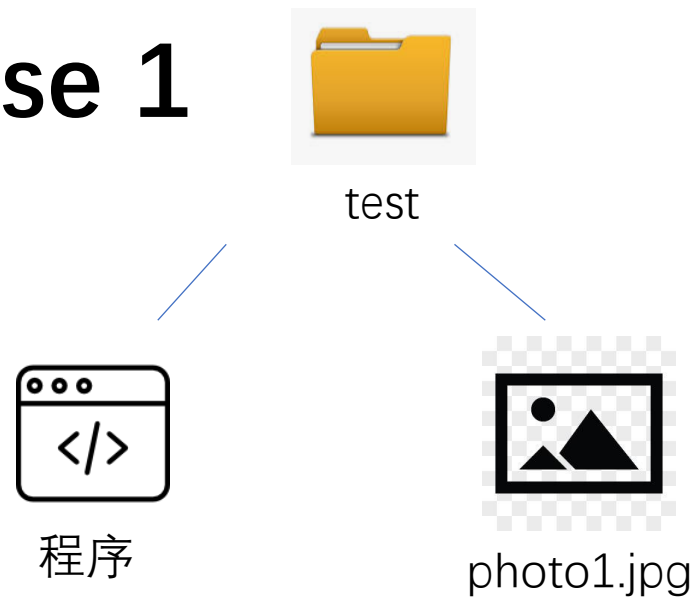


photo1 的相对路径:  
**photo1.jpg**

## Case 2

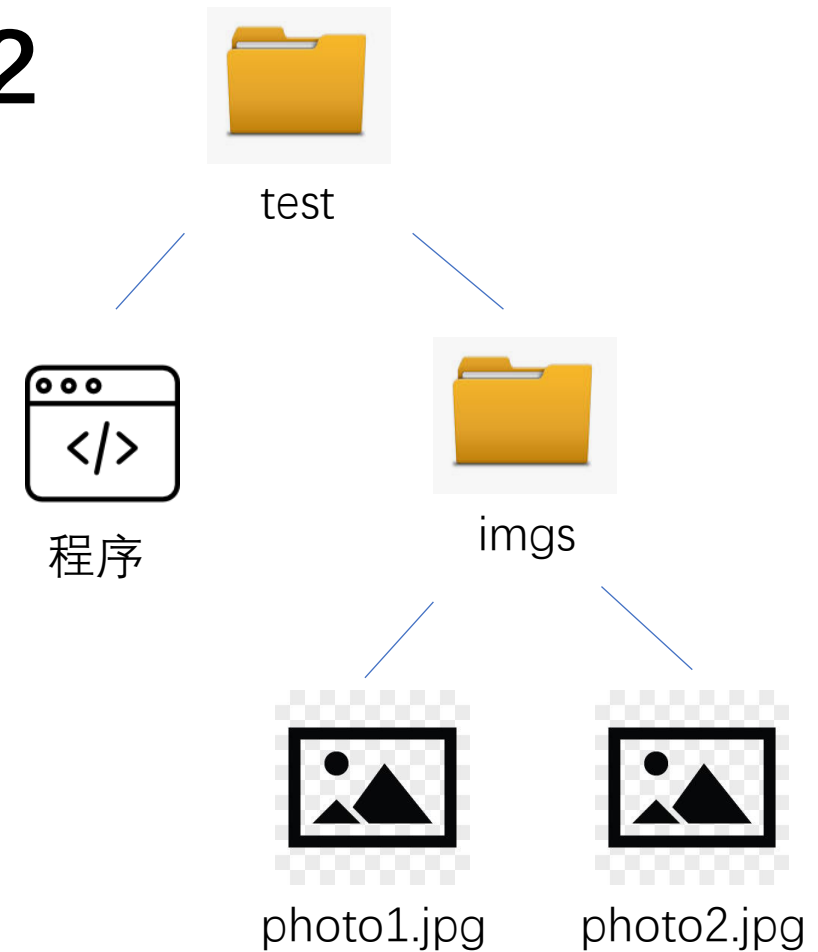


photo1 的相对路径:  
**imgs\photo1.jpg**

接次页 ➞

接前页

## Case 3

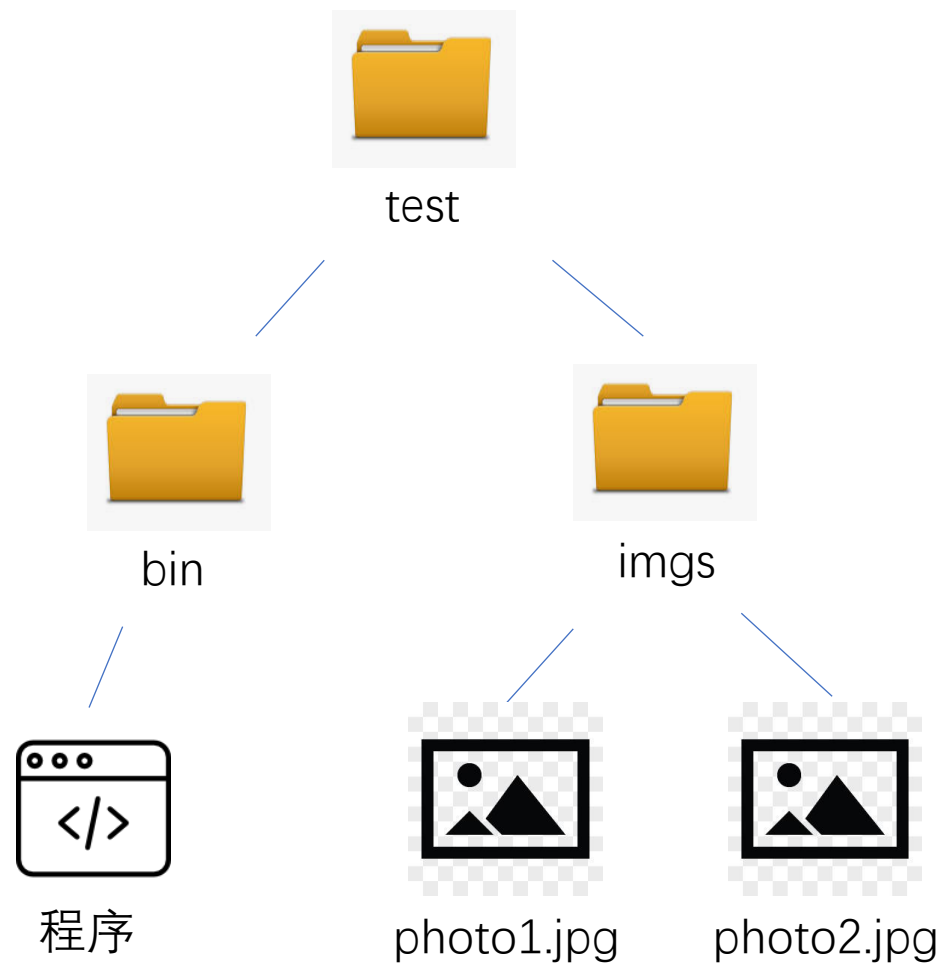


photo1 的相对路径:  
`..\imgs\photo1.jpg`

## Case 4

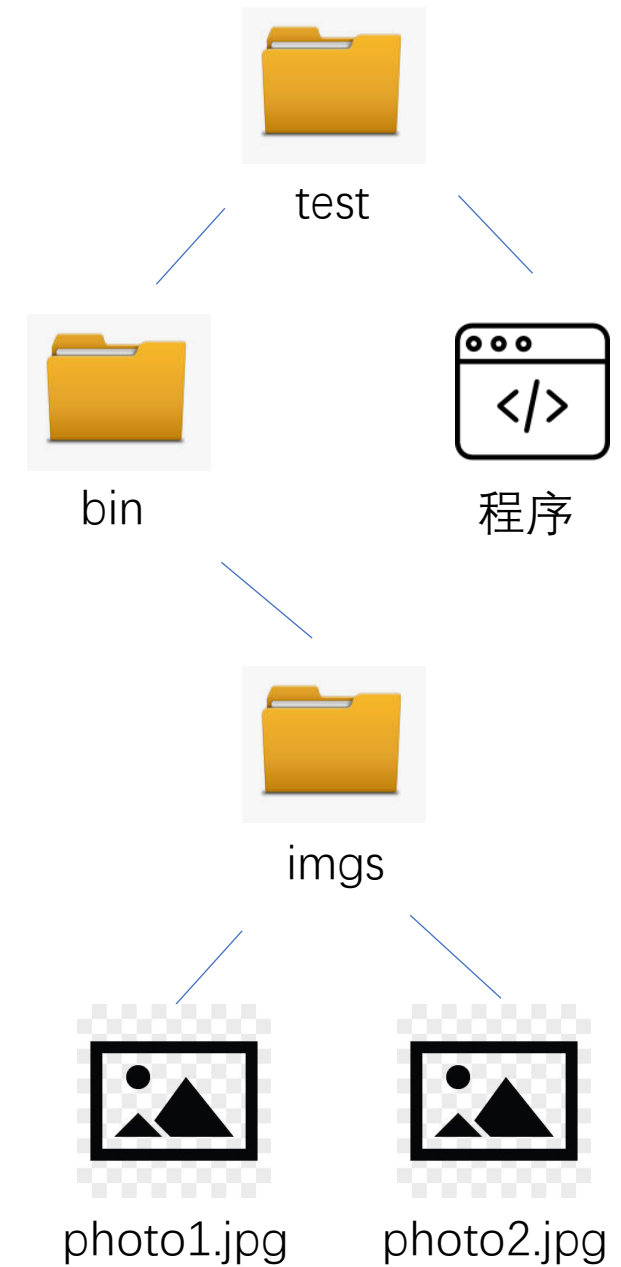


photo1 的相对路径: ?

# 管理文件的常用方法

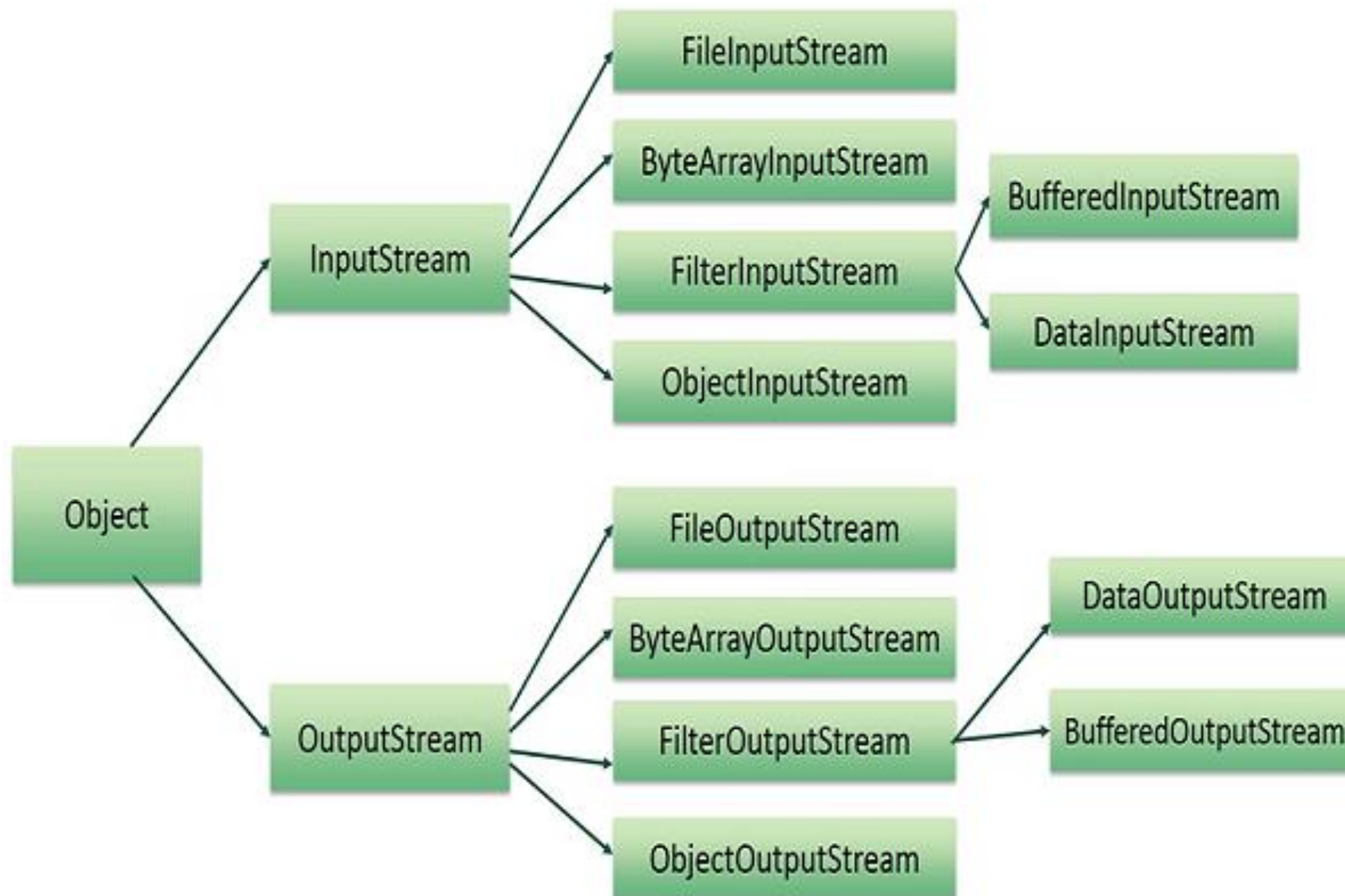
方法	功能
exists	判断文件（夹）是否存在
createNewFile	如果文件不存在，创建文件
mkdir	如果文件夹不存在，创建文件夹
delete	删除文件（夹）
getName	获得文件名
getAbsolutePath	获得文件（夹）的绝对路径
length	获得文件的大小
list	获得文件夹里的文件（夹）列表
canRead、canWrite	判断文件是否可读、可写





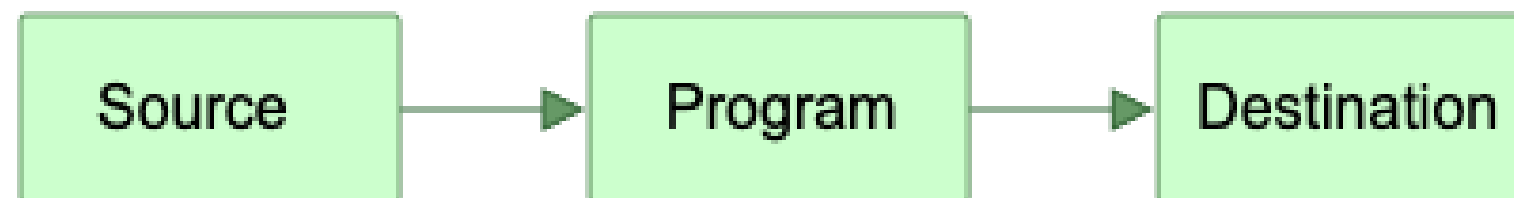
# Java I/O API

- Java **I/O** (Input / Output) API 用于输入和输出。



# I/O API 的功能

- I/O API 不仅提供文件读写，还提供从各种数据源读取数据并将数据写入数据目标的类。
- 典型的数据源和目标包括：
  - File、
  - Pipe、
  - 网络通信端口、
  - 控制台（System.in, System.out, System.err）



# I/O Streams

- I/O Stream 是 Java I/O 里的核心概念。
- 可以简单理解为将数据的输入和输出抽象化成为连续的操作。
- 又可分为两种 Stream:
  - Byte Stream: InputStream、OutputStream 的子类
    - 以 byte（字节）为单位进行读写。
    - 一般用于处理图片、音频等非文本数据。
  - Char Stream: Reader、Writer 的子类
    - 以文字（字符）为单位进行读写。
    - 一般用于处理文本数据
- 我们今天只介绍 Reader 和 Writer 的用法。



# 解读 System.out.println

- System.out.println -> 向标准输出写入数据：

System.out.println(xx)

System 类

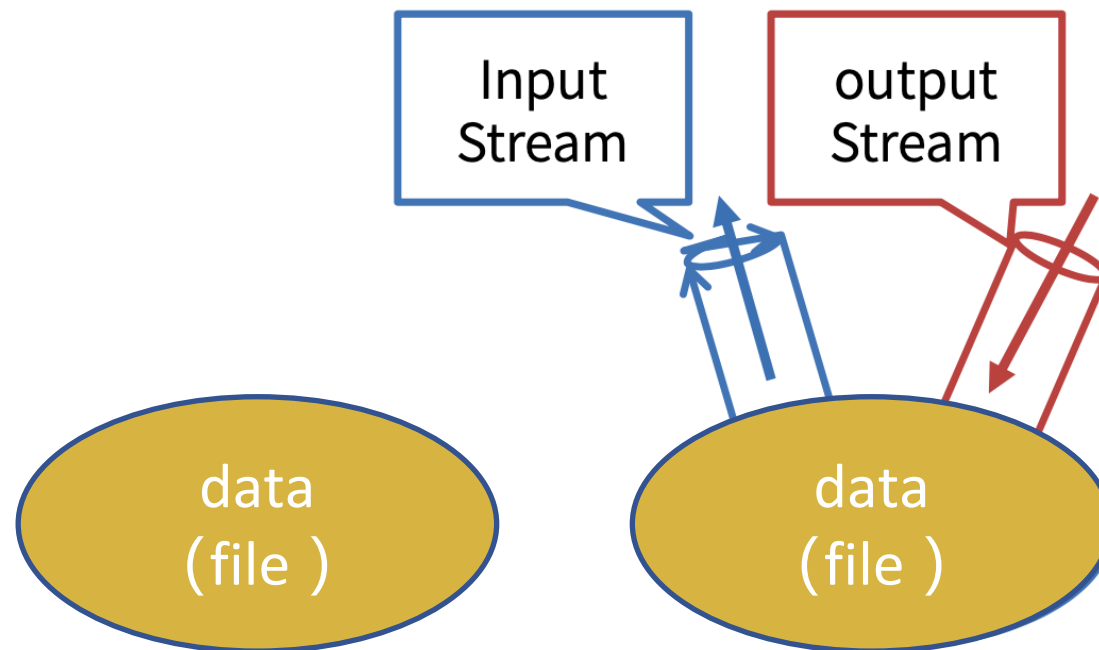
静态变量 out 里保存了一个  
PrintStream 的对象

内部拥有指向标准输出（通常是 console）的 OutputStream

println()  
println(String s)  
println(int i)  
等等 PrintStream 的方法

# I/O Stream 的用法

1. 创建文件对应的 File 对象。
2. 创建读（写）该文件的 Stream 类的对象。
3. 使用 Stream 对象的方法进行读（写）。
4. 使用 **close** 方法关闭 Stream 对象。



# 文件读写所需类

- 最简单的文件读写处理主要用到以下三类：
  - **File** 类，用来指定需要读写的文件；
  - **FileReader / FileWriter** 类，用来直接对文件读写字符；
  - **BufferedReader / BufferedWriter** 类，用来提高读写效率，并添加一些方便的读写方法。

**BufferedReader**  
read  
readLine

**Reader**  
read



# Reader / Writer 的创建

- 直接使用 FileReader / FileWriter 只能一个一个字符读写，一般将它放入一个 BufferedReader / BufferedWriter 中方便我们按行读写。
- 以 BufferedReader 为例：

```
1 File file = new File("test.txt");  
2 FileReader fileReader = new FileReader(file);  
3 BufferedReader reader = new BufferedReader(fileReader);
```

- 此时，FileReader 的构造器可能抛出一个 FileNotFoundException，需要进行异常处理。

# BufferedReader 的使用

- BufferedReader 提供的常用方法包括：

方法	功能
<code>read</code>	读取一个字符。
<code>readLine</code>	读取一行字符。
<code>lines</code>	读取很多行数据。返回值是一个字符串的 <code>Stream</code> ，可以转化为列表。
<code>reset</code>	再次从头开始读取。
<code>skip</code>	跳过指定数量的字符。

# BufferedWriter 的使用

- BufferedWriter 提供的常用方法包括：

方法	功能
<code>write(int c)</code>	写入一个字符。
<code>write(String s)</code>	写入一个字符串。
<code>newLine</code>	换行。
<code>flush</code>	刷新输出缓存。



# close 方法和异常处理

- 在使用完 Reader / Writer 后，一定要把所有 Reader / Writer 都用 close 方法关闭：

```
fileReader.close();  
reader.close();
```

- 但为了异常处理，我们可能会把读写的代码用 try-catch 语句括起来。如果发生了异常，catch 语句可能会直接跳出方法，跳过 close 的语句。
- 为了保证 close 方法可以关闭这些类，我们可以使用 finally 语句 / try-with-resource 语句。

# 使用 try-with-resource 语句处理异常

- 使用 try-with-resource 语句可以非常轻松地处理文件读写时出现的各种异常。Java 会在 try 代码块结束时自动关闭所有声明的 Reader / Writer, 不管是否出现异常:

```
1 try (  
2     FileReader fileReader = new FileReader(file);  
3     BufferedReader reader = new BufferedReader(fileReader)  
4 ) {  
5     System.out.println(reader.readLine());  
6 } catch (Exception e) {  
7     System.out.println("Exception occurred while reading the file: " + file);  
8     e.printStackTrace();  
9 }
```



Q & A

*Question and answer*



# 目录

1

修饰符

2

文件读写

3

线程管理

4

正则表达式

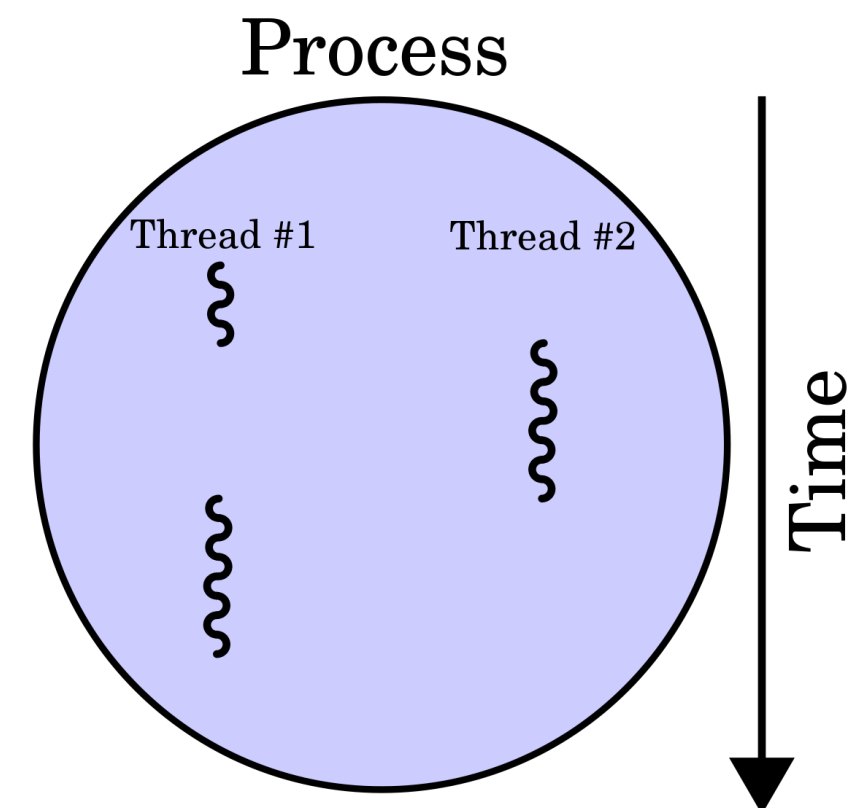
5

其他常用 API



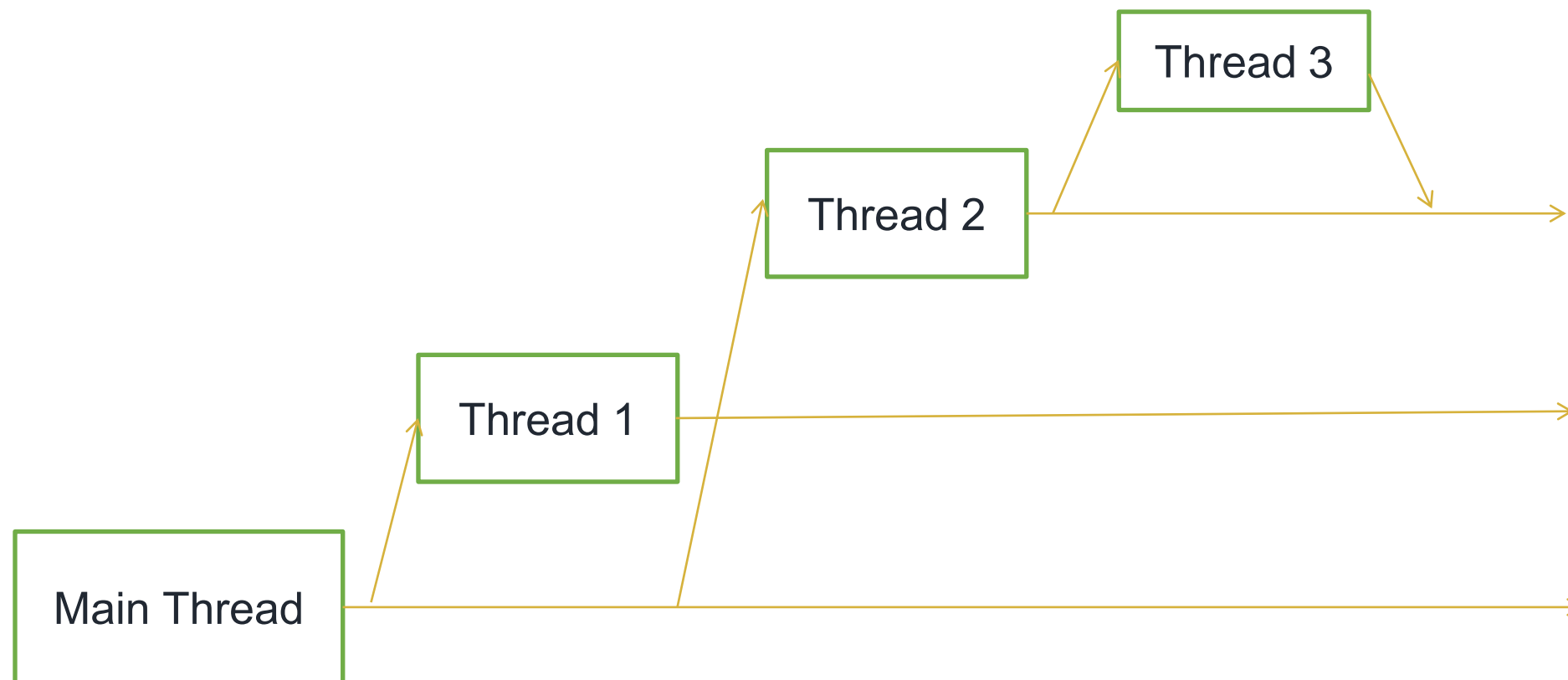
# 进程和线程

- 在计算机里，每当我们想要运行一个程序，操作系统会为它分配一个**进程**[プロセス]。多个进程可以同时运行，系统会自动帮助我们自动安排它们的运行时间。
- 但有时我们也需要在同一个程序内部同时进行多个任务。此时，就应该使用**线程**[スレッド]。和进程一样，操作系统可以自动帮我们同时分配多个线程的运行时间：
- 想一想，我们什么时候应该用到多线程？



# Java 中的线程

- 当我们开始运行一个程序的时候，所有的代码都运行在一个默认的线程上，也就是所谓的**主线程**`[メインスレッド]`。
- 如果我们想让一个任务和主线程同时运行，我们可以创建一个新的线程，把任务交给这个线程执行。



# 创建线程

- **Thread** 类是 Java 中代表线程的类。
- 要创建线程，使用 Thread 的构造方法创建一个线程对象。查阅文档，我们会发现 Thread 的构造方法接受一个 Runnable 参数：

```
Thread(Runnable target)
```

Allocates a new Thread object.

- Runnable 是一个函数式接口，只有一个方法 run：

```
1 package java.lang;
2
3 @FunctionalInterface
4 public interface Runnable {
5     public abstract void run();
6 }
```

接次页 



- 因此，要创建线程，我们需要先有一个类实装 `Runnable` 接口，然后把这个类的实例对象传入 `Thread` 的构造器。
- 由于 `Runnable` 是一个函数式接口，最简便的方法便是利用 `lambda` 表达式 (← § 3.2.2) 了：

```
1 Thread thread = new Thread(() -> {  
2     System.out.println("Do something 1");  
3     System.out.println("Do something 2");  
4     System.out.println("Do something 3");  
5 });
```



# 运行线程

- 注意，我们到目前为止只是**创建**了一个线程，它还并没有开始**运行**。要让一个线程开始运行，我们需要使用它的 `start` 方法：

```
1 Thread thread = new Thread(() -> {  
2     System.out.println("Do something 1");  
3     System.out.println("Do something 2");  
4     System.out.println("Do something 3");  
5 });  
6 thread.start(); // Do something 1, 2, 3
```

# 运行多个线程

- 我们使用线程的主要目的是为了同时执行多个任务。我们可以创建多个不同的线程，看看它们会不会同时运行。

Try 

MultiThread.java

Note 

线程实际执行的顺序并不一定等同于我们使用 start 的顺序，而是由操作帮我们决定的。

# 线程的方法

- 除了 `start` 方法，`Thread` 还提供了其他一些方法以供我们控制线程的动作。常见的方法如下：

方法	功能
<code>start</code>	开始运行线程。
<code>setPriority</code>	设置线程的优先级。
<code>join</code>	等待该线程结束。
<code>interrupt</code>	中断线程。
<code>isAlive</code>	判断线程是否正在运行。
<i>static</i> <code>sleep</code>	让当前线程暂停一段时间。

# 并发问题

- 由于线程的实际运行顺序由操作系统自动帮我们决定，多个线程同时运行可能会产生很多无法预期的问题，也就是所谓的**线程并发**[並列处理]问题。
- 比如，如果我们尝试在不同线程中修改同一个变量的值，产生的结果可能与我们的预期不符。



- 想一想，为什么会产生这样的结果？



# 原子性

- 线程无法如我们预期运行的很大一部分原因是因为某一段代码没有满足**原子性**[不可分性・アトミック性]。原子性是指某一段代码在运行时应该永远**连续**运行，中间不会插入别的代码的实行。
- 比如刚刚的例子中，我们希望每次变量增加的过程中没有别的代码被运行。
- 在 Java 中，我们可以使用 **synchronized** 关键字帮助我们控制某段代码在同一时间内只能由 **1** 个线程运行。
- **synchronized** 修饰的同一个对象的方法在同一时间只能由 **1** 个线程实行。



# synchronized 关键字的其他用法

- 除了修饰方法，synchronized 关键字还有别的使用方法帮助我们控制线程并发。我们这里只做简单总结：

- 修饰代码块：

```
1 synchronized(obj) {  
2     // codes  
3 }
```

使用对象对这段代码块“上锁”。同一时间，用同样对象上锁的方法只能有一个运行。最常见的用法是使用“this”或“类名.class”当做“锁”。

- 修饰普通方法。同一时间同一个对象的方法只能由一个线程执行。
- 修饰静态方法。同一时间这个方法只能由一个线程执行。

Q & A

*Question and answer*



# 目录

1

修饰符

2

文件读写

3

线程管理

4

正则表达式

5

其他常用 API



# 正则表达式

- **正则表达式**[正規表現] (Regular Expression, **Regex**) 是一种特殊的语言, 它可以用来**匹配**满足某些条件的字符串。
- 每一个正则表达式都是一个字符串。比如, 以下字符串可以匹配满足邮箱格式 (ABC@XXX.YZ) 的字符串:

```
[\\w\\-\\.\\_]+@[\\w\\-\\.\\_]+\\. [A-Za-z]+
```

- 正则表达式主要有两种实际的用法:
  - 判断某个字符串是否满足条件: **表单验证**[フォーム検証]。
  - 从某个字符串里找出所有满足条件的子串: **爬虫**[クローラ]。

# 正则表达式的基本语法

- 如果不包含特殊字符，一个普通的字符串将匹配它本身。

## Example ✓

正则表达式: `Apple`

匹配	不匹配
Apple	Banana
	apple
	chocolate

# 特殊字符：.

- 通配符 “.” 将匹配 1 个任意字符。

## Example ✓

正则表达式: `Ap.le`

匹配	不匹配
Apple	Banana
Apkle	apple
Ap3le	Aple
Ap~le	Appple

# 特殊字符： \*

- “a<sup>\*</sup>” 将匹配 0 个以上的 “a” 字符。

## Example ✓

正则表达式: `Ap*le`

匹配	不匹配
Apple	Banana
Aple	apple
Ale	Apkple
Appppple	AppIE



# 特殊字符的结合

- 可以将前两种语法组合：“.”将匹配 **0** 个以上的**任意**字符。

## Example ✓

正则表达式: `A.*le`

匹配	不匹配
Apple	Banana
Abbble	apple
Aabcle	AppIE
Ale	pppAle

## 特殊字符：()

- 可以使用圆括号 “**()**” 配合其他符号：“**(abc)\***” 将匹配 **0** 个以上的 **abc** 字符。

### Example ✓

正则表达式：

```
A(pp)*le
```

匹配	不匹配
Apple	Banana
Ale	apple
Appppppple	Appple

# 特殊字符: +

- “a+” 将匹配 1 个以上的 “a” 字符。

## Example ✓

正则表达式: `A(pp)+le`

匹配	不匹配
Apple	Banana
Appppple	apple
Appppppple	Ale

# 特殊字符：|

- “a|b” 将匹配 “a” 字符或 “b” 字符。

## Example ✓

正则表达式: `A(p|q|r)*le`

匹配	不匹配
Apple	Banana
Aqqle	apple
Apqrrle	Assle
Ale	Apqrsle



# 特殊字符：[ ]

- “[abc]” 将匹配 “a”、“b” 或 “c” 字符。

## Example ✓

正则表达式: `A[pqrst]+le`

匹配	不匹配
Apple	Banana
Apppppple	apple
Asle	Ale
Aqrtle	Awwwle

# [ ] 的简便写法

- “[a-z]” 将匹配小写字母； “[A-Z]” 将匹配大写字母； “[0-9]” 将匹配数字。

## Example ✓

正则表达式: `A[a-z0-9]*le`

匹配	不匹配
Apple	Banana
Ale	apple
A123abcle	APPlE

# 特殊字符： \

- 和 Java 类似，“\” 是转义字符，可以用来表达一些特殊匹配，或是匹配一些无法直接书写的特殊字符：

转义序列	匹配内容
\n	换行
\w	英文字母
\d	数字
\s	空白字符
\.	.
\+	+
\(	(
\\	\

# 其他正则表达式语法

- 正则表达式所有语法一览：  
 <https://quickref.me/regex>
- 正则表达式可视化：  
 <https://regexper.com/>



# Java 中的正则表达式

- Java 标准包中的 Pattern 类提供了正则表达式的相关方法。
- 要判断某个字符串是否匹配某个正则表达式，可以使用 Pattern 的 Pattern.matches 方法（也可以是用 String 的 matches 方法）：

```
1 // => true
2 System.out.println(Pattern.matches("A[pqr]+le", "Apple"));
3 // => false
4 System.out.println(Pattern.matches("A[pqr]+le", "Awle"));
```

- 要从某个字符串中找出所有匹配的子串会稍微复杂一些，需要用到 Matcher 类的 find 和 group 方法。



# 转义字符的注意事项

- 正则表达式中的转义字符 “\” 在 Java 中不能直接书写，需要转义为 “\\”。
- 这也就意味着要匹配字符 “\”，Java 中要写成 “\\\\”.....

\	BACKSLASH
\\	REAL BACKSLASH
\\\	REAL REAL BACKSLASH
\\\\	ACTUAL BACKSLASH, FOR REAL THIS TIME
\\\\\\	ELDER BACKSLASH
\\\\\\\\	BACKSLASH WHICH ESCAPES THE SCREEN AND ENTERS YOUR BRAIN
\\\\\\\\\\	BACKSLASH SO REAL IT TRANSCENDS TIME AND SPACE
\\\\\\\\\\\\	BACKSLASH TO END ALL OTHER TEXT
\\\\\\\\\\\\\\...	THE TRUE NAME OF BA'AL, THE SOUL-EATER

Artist: xkcd

Q & A

*Question and answer*



# 目录

1

修饰符

2

文件读写

3

线程管理

4

正则表达式

5

其他常用 API

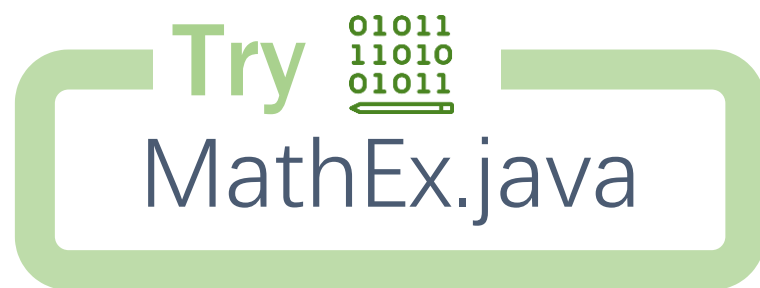


# Math

- **Math** 类提供了一些数学计算相关的静态变量和方法：

变量	含义
PI	圆周率
E	自然对数的底

方法	功能
max、min	求两个数的最大、最小值
abs	求绝对值
cos、sin、tan 等	求三角函数值
ceil、floor	向上、向下取整
pow	求乘方
exp、log	求自然对数的幂、对数
random、rint	产生随机数



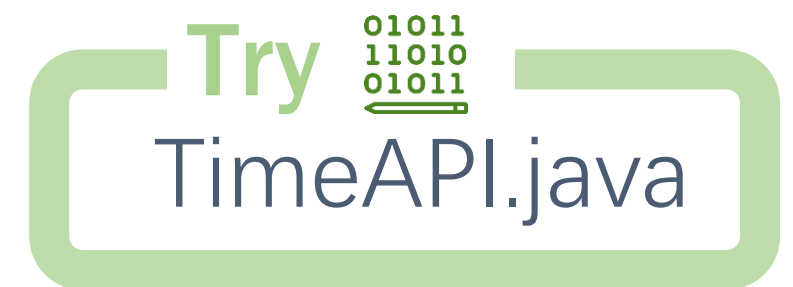
- 其他方法可以查阅文档：

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Math.html>

# 时间 API

- **java.time** 提供了和时间（日期）相关的功能。
- 以 **LocalDateTime** 类提供的功能为例：

方法	功能
<i>static</i> of	通过各种形式的输入创建一个表示日期时间的对象
<i>static</i> now	获得现在日期时间的对象
get	获得各种日期时间信息
plus	加上各种日期时间
with	设置各种日期时间的值



- 其他功能可以查阅文档：

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/time/package-summary.html>

# Scanner

- **Scanner** 类提供了简便的输入数据的方法。最常见的用途是用于处理控制台输入，以用于开发控制台应用或进行测试。
- 要使用 Scanner 处理控制台输入，将 **System.in** 传入它的构造器：

```
Scanner scanner = new Scanner(System.in);
```

## Note

Scanner 也属于 I/O Stream，不要忘了在结束使用后使用 **close** 方法关闭！

# Scanner 方法

- Scanner 提供了以下方法获得输入数据：

方法	功能
next	获得下一个项目（字符串）
nextLine	获得本行剩余内容
nextInt	获得下个整数
nextDouble	获得下个小数
nextBoolean	获得下个布尔值

- 注意：如果要用户输入多行数据，最好在每行输入结束之后使用 `nextLine` 方法换行。





# 字符串处理的问题

- 在编程时，经常需要反复修改某一个字符串的值：

```
String str = "";  
for (int i = 0; i < 10; i++) {  
    str += i + " ";  
}
```

- 由于 Java 中的字符串本质上是不可变的，我们其实是在不断**创建新的字符串对象**，会消耗很多额外资源。
- 当字符串需要被反复改变时，可以考虑使用 **StringBuilder** 类。

# StringBuilder

- **StringBuilder** 是一个保存字符串的类，它可以在不创建新对象的前提下对现有字符串进行修改。
- 可以用 `StringBuilder` 改写刚刚的代码：

```
1 StringBuilder sb = new StringBuilder();  
2 for (int i = 0; i < 10; i++) {  
3     sb.append(i).append(" ");  
4 }  
5 String str = sb.toString();
```

- `append` 方法只会修改现有对象，不会创建新的对象。

# StringBuilder 的方法

- StringBuilder 类还提供了许多其他方便的方法:

方法	功能
charAt	获得指定位置的字符
indexOf	找到某个子串的位置
insert	往某个位置插入
delete	删除范围内的字符串
replace	替换范围内的字符串
reverse	翻转字符串

Try  DynamicString.java

- 其他方法可以查阅文档:

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/StringBuilder.html>


# Optional

- **Optional** 类用于保存一个可能为空（null）的对象：

```
Optional<String> data = Optional.ofNullable(str);
```

- 使用 Optional 类可以提高代码可读性，预防忘记空检查的情况。它也能让我们更方便地书写一些操作可能为空的对象的代码：

```
1 // 只在数据不为空时输出它
2 data.ifPresent(s -> System.out.println("str is " + s));
3 // 输出数据，如果数据为空，输出 "No data"
4 System.out.println(data.orElse("No data"));
```

Try  OptionalTest.java

- 其他方法可以查阅文档：

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Optional.html>



Q & A

*Question and answer*

# 总结

## Sum Up

1. 修饰符: static、abstract、final。
2. 文件读写的方法。
3. 线程的基本概念和使用方法。
4. 正则表达式的基本使用方法。
5. 其他常用 API: Math、LocalDateTime、StringBuilder、Scanner、Optional



**THANK YOU!**