

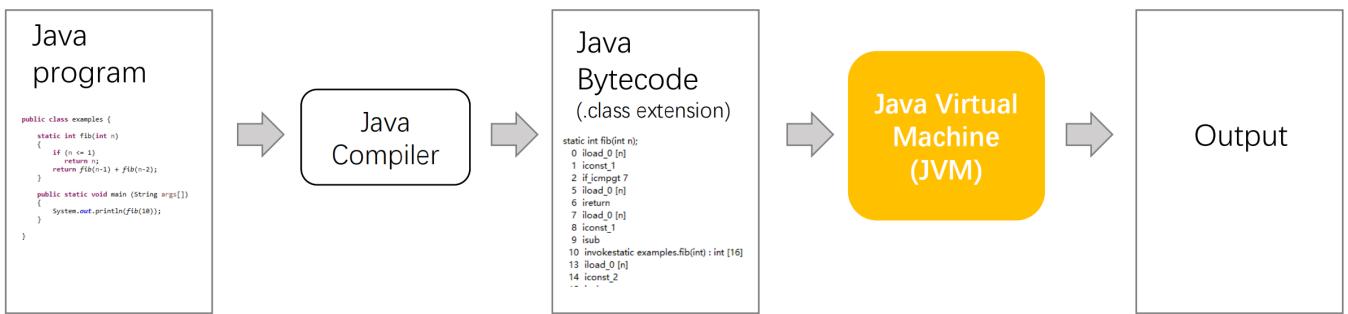
CS209 Final Review(1-9)

Lecture 1

Software and Hardware

- **System Software:** OS Device Driver Application Software

Java review and JVM



- 编译成 `.class` 文件之后，无论在什么OS系统上都可以通过JVM运行。
- 用较低版本的JDK编译的 `.class` 文件可以在高版本的JRE上运行，但是在高版本的JDK编译的 `.class` 文件就不可以在低版本的JRE上运行，因为可能会包含更多的包和功能。
- JVM对多语言（Java、Kotlin、JRuby、Groovy）都有支持，编译成了Bytecode之后都可以通过JVM运行。

组件	职责	包含的内容	适用场景
JVM	执行 Java 字节码	JVM 本身	运行 Java 程序的核心
JRE	运行 Java 程序	JVM + 类库 + 运行时工具	仅运行 Java 程序
JDK	开发 Java 程序	JRE + 编译器 + 开发工具	开发和运行 Java 程序

只有JDK可以把一个 `.java` 文件转换成 `.class` 文件，如果 `.class` 文件中运用了java.utils这些外部库，即便编译结束了JVM也无法执行，需要JRE才可以

JVM包括三部分

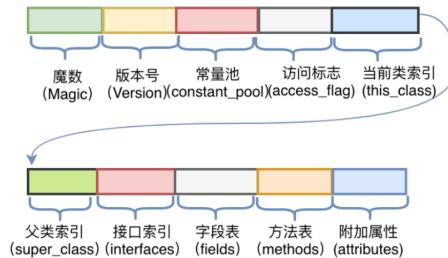
- a. Class Loader Subsystem

主要步骤：

1. Loading

- a. JVM 根据类的全限定名 (**fully qualified name**) 找到对应的字节码文件
 - 例如, `java.lang.String` 这样的类名会指向具体的 `.class` 文件
- i. **Bootstrap ClassLoader**: 首先搜索 `JAVA_HOME/lib` 路径。
- ii. **Extension ClassLoader**: 搜索 `JAVA_HOME/lib/ext` 目录。
- iii. **Application ClassLoader**: 搜索由 `classpath` 环境变量 (或 `-cp` 参数) 指定的路径。
- iv. 如果仍未找到, 则抛出 `ClassNotFoundException` 异常。

- b. 将类的结构信息存储到方法区 (**Method Area**)



- i. **类信息**: 包括类名、访问修饰符 (如 `public/private`)、父类、实现的接口, 以及静态变量等。
- ii. **字段信息**: 包括字段名、字段类型、访问修饰符等。
- iii. **方法信息**: 方法名、参数类型、返回值类型、访问修饰符, 以及方法对应的字节码等。
- iv. **运行时常量池**: 包含字符串引用、`final` 常量, 以及符号引用
- c. JVM 创建一个 `java.lang.Class` 对象
 - i. 用于访问存储在方法区的类信息。
 - ii. 这个对象提供了操作类元数据的方法, 例如通过反射机制获取类的信息。

2. Linking

- a. **验证 (Verify)** : 验证字节码是否符合 Java 语言规范 (Java Specification) 。
- b. **准备 (Preparation)** : 为类的 **静态变量 (static variables)** 分配内存, 并将其初始化为JVM 的默认值。不会被赋予编程中指定的值。`int` 为 `0` `boolean` 为 `false` 。
- c. **解析 (Resolve)**
 - 将 **常量池 (Constant Pool)** 中的 **Symbolic References** 解析为 **Direct References**
 - **Symbolic References** : 符号引用是逻辑名称, 如 `java/lang/System.out` 或 `println(Ljava/lang/String;)V`。

- **Direct References**: 直接引用是具体的内存地址, `System.out` 是指向一个 `PrintStream` 对象的地址。`println` 是指向方法字节码的内存地址。
- 确定类或方法的具体实现, 对于类名或方法名, 解析阶段会找到它的实现 (包括其内存位置)。
- **Resolve**适用于不涉及多态 (无需运行时绑定) 的所有方法。

3. Initialization

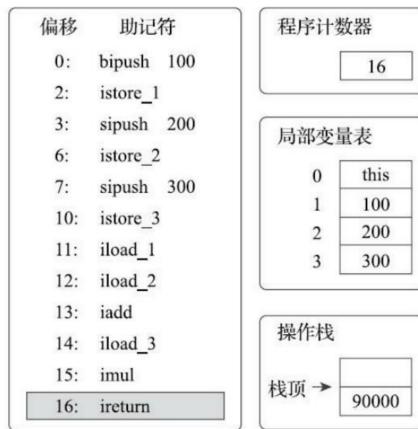
- a. 初始化静态变量 (Initialize Static Variables) : 为类的静态变量赋予编程中指定的值。
 - 注意: 在之前的 **准备阶段 (Preparation)** 中, 静态变量已经被分配内存并赋予默认值, 但只有到了初始化阶段, 静态变量才会被赋予实际的值。
- b. 执行类中定义的 `static{}` 块。
- `.class` 文件中有classpath定位 `.jar` 包 **Class Loader** 会负责Locate and Load to memory

b. Runtime Data Area

- **Method Area(方法区)**: 类的结构信息 所有方法的字节码 静态变量 **Runtime Constant Pool** (字符串引用 符号引用)
- **Heap Area(堆区)**: 对象实例 (通过 `new` 创建) 实例变量 (非静态变量) 字符串常量池 (字符串 "Hello")
- **Stack Area(栈区)** : `this` 关键字指代的对象 **Runtime Constant Pool Reference**
 - 方法调用时的栈帧:
 - 局部变量表: 包括方法的参数和局部变量。
 - 操作数栈: 存储计算过程中临时的操作数。
 - 方法返回地址。
- **关键点:** 运行时常量池 是方法区的一部分, 用于存储编译期和运行时解析的常量。字符串常量池 是堆的一部分, 专门存储字符串字面量。

c. Execution Engine

- 执行过程



1. 每次调用方法时，会为该方法创建一个新的 **栈帧（Stack Frame）**，并将其压入 **JVM 栈**
2. 方法执行结束后，栈帧会从 JVM 栈中弹出。
3. 通过局部变量引用定位堆中的对象
4. 在调用一个静态方法或访问一个静态字段时，执行引擎会从方法区中获取信息
 - 执行引擎需要从 **方法区（Method Area）** 中查找类的元数据（如字段、方法签名等）。

示例：

```

1  public class Example {
2      public static void main(String[] args) {
3          int a = 5;
4          int b = 10;
5          int c = add(a, b);
6          System.out.println(c);
7      }
8      public static int add(int x, int y) {
9          return x + y;
10     }
11 }
```

- a. JVM 加载 `Example` 类，找到 `main` 方法，创建栈帧并压入 JVM 栈。
- b. 方法内的局部变量 `a` 和 `b` 被存储在栈帧的 **局部变量表** 中。
- c. 调用 `add` 方法时，创建新的栈帧，参数 `x` 和 `y` 存储在新栈帧的局部变量表中。
- d. `x + y` 的计算过程在 **操作数栈** 上完成，结果返回给调用栈帧。
- e. `main` 方法的栈帧通过 `System.out.println` 输出结果。
- f. 方法执行完成后，栈帧从 JVM 栈中弹出。

■ 特点

- 执行引擎通过局部变量表中的引用定位堆中的对象，同时从方法区中查找类和方法的元数据。

```
1  public class Example {  
2      static String staticVar = "Static Variable";  
3  
4      public static void main(String[] args) {  
5          Example obj = new Example();  
6          obj.printMessage();  
7      }  
8  
9      public void printMessage() {  
10         System.out.println(staticVar);  
11     }  
12 }
```

• 方法调用

- 当调用 `obj.printMessage()` 时：
 - 执行引擎从局部变量表中获取 `obj` 的引用，定位到堆中的对象。
 - 执行引擎从方法区获取 `printMessage` 方法的字节码，并执行它。
 - `printMessage` 方法中访问了静态变量 `staticVar`，执行引擎从方法区中找到该变量的值，并将其打印。

■ 线程相关

• 线程独立的内存区域

- 程序计数器（Program Counter Register）：记录当前线程正在执行的字节码指令地址。
- JVM 栈：每个方法调用都会在栈中生成一个栈帧，用于存储局部变量表、操作数栈等信息。
- 本地方法栈（Native Method Stack）：当方法调用本地（非 Java）代码时使用。

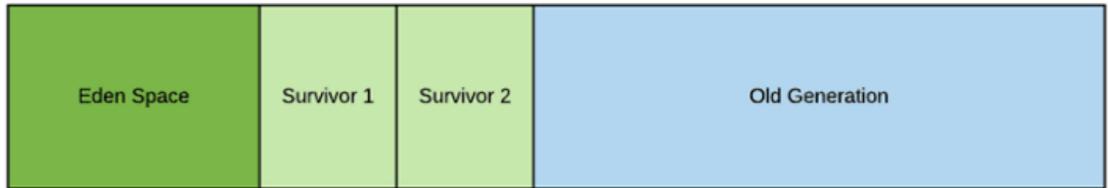
• 线程共享的内存区域

- 堆（Heap）：存储所有对象实例和数组。
- 方法区（Method Area）：存储类的元数据、静态变量、运行时常量池等。

■ 主要组件

- Interpreter：逐行将 bytecode 转换为机器码

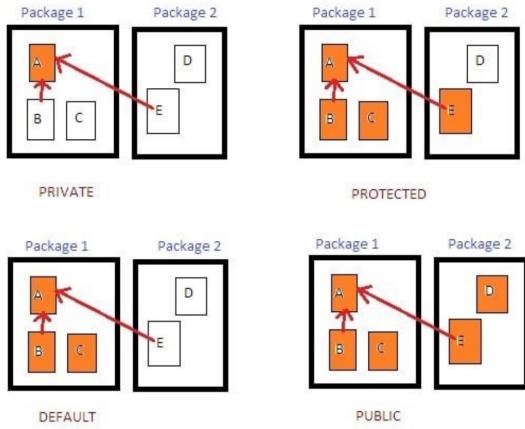
- **JIT Compiler**：如果检测到某一个bytecode的代码块被多次翻译（**Hot Spot code**），直接将翻译后的机器码储存在本地，后续可以直接执行
- **Garbage collector**：自动回收不用的变量聚焦于 **Heap Area**
 - **Reference Counting** 缺点：无法处理循环引用
 - **Reachability Analysis**：从 **GC Roots** 起始点出发，进行引用链追踪。如果一个对象在引用链中不可达，则被视为垃圾 优点：可以处理循环引用
 - **GC Roots** 一般包括：栈中的局部变量 方法区中类的静态属性 方法区中常量引用
 - **Naïve Algorithm**（简单算法）：扫描整个堆内存（Heap Memory），标记所有无法访问的对象，并回收这些对象所占用的空间。
 - **效率低**：扫描整个堆内存需要耗费大量时间，尤其在堆内存非常大的情况下。
 - **内存碎片化**：回收不可达对象后，剩余的存活对象分布不连续，会导致内存碎片化，降低内存使用效率。
 - **Improved Algorithm**



- 堆内存被划分为两大区域：
 - a. **Young Generation (年轻代)**：存储新创建的对象。**Eden Space**（伊甸区）：新对象最初分配在这里。**Survivor 1** 和 **Survivor 2**（幸存区）：存放在年轻代垃圾回收中存活下来的对象。
 - b. **Old Generation (老年代)**：存储生命周期较长的对象。
- 当年轻代的 Eden 空间满了时，会触发 Minor GC。Minor GC 只针对年轻代进行回收，存活的对象会被转移到 Survivor 区或晋升到老年代。
- 在 JVM 的堆内存管理中，为 Young Generation 对象设定了 threshold，对象在堆中每存活一次 Minor GC，年龄会增加 1，当对象的年龄达到该阈值时，就会被移动到 Old Generation
- 老年代的垃圾回收被称为 **Major GC**

Software design principles & OOP concepts

- **OO concepts:** **Encapsulation** **Abstraction** **Inheritance** **Polymorphism**



Lecture 2

Generics

- Primitive Type: byte short int long float double char boolean
- 泛型的好处：
 - i. 如果没有泛型，我们可以把任何类型都放进一个数据结构里，然后取出来的时候explicit type cast，但是这样容易引发RE，而如果用泛型一个数据结构只接受同样的Type，这样如果有问题会CE，问题越早发现越好。
 - ii. 泛型无需Explicit Type Cast

Example	Term
List<E>	Generic type
E	Formal type parameter (类型形参) Type variable
List<String>	Parameterized type
String	Actual type parameter (类型实参)
List	Raw type (原始类型)

Raw Type 只会warning但不会error的原因：

- 向后兼容 确保JDK5.0之前的代码还可以运行
- 在编译的时候会Type Erasure 把Parameterized type转化成Raw Type(Object)，所以对于JVM来说，没有Generics

```

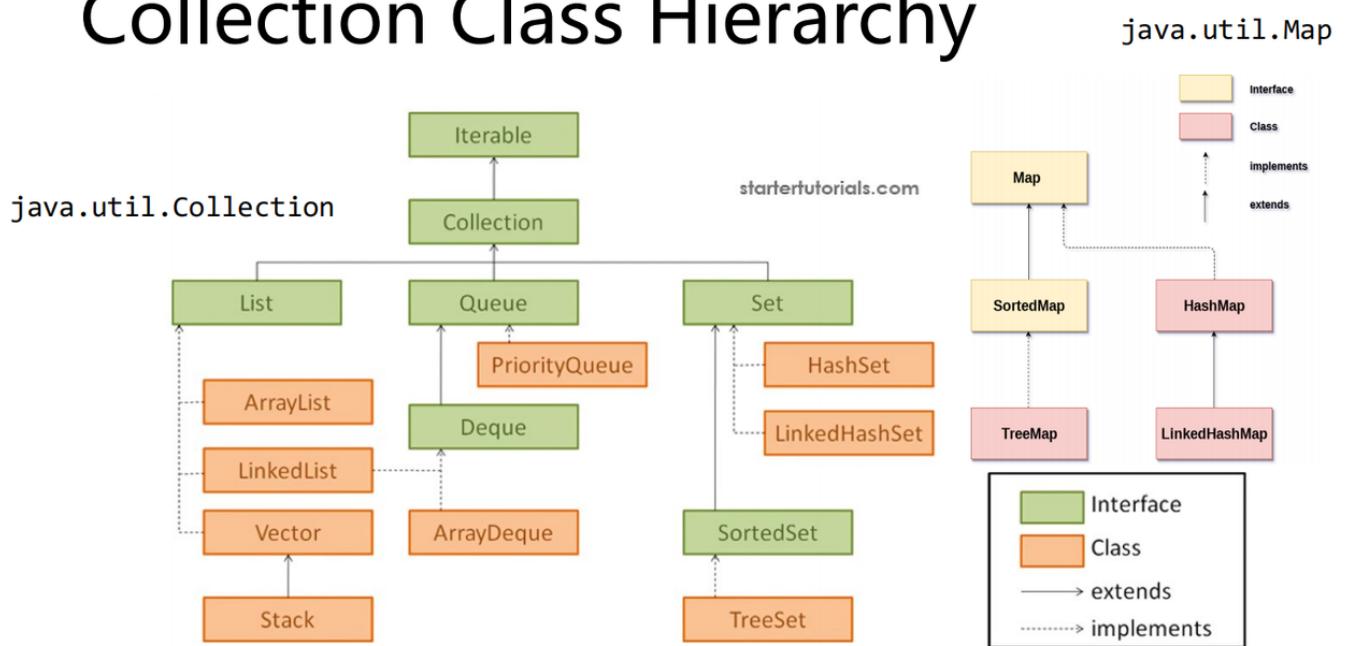
1  class ClassName extends SuperClass implements I1, I2, I3
2  class Box<T extends Animal & Runnable & Swimmable>
  
```

- 普通类和Generics都只能继承一个Class，但是普通类implements接口，而Generics都用extends，并且一个用`,`间隔，一个用`&`，并且class必须放在第一个，接口继承接口用extends
- Number是Integer的父类，但是Pair<Number>并不是Pair<Integer>的父类。这个时候就需要用Wildcards（通配符），Pair<?>是Pair<Integer>、Pair<String>等一系列Pair共同的父类，所以不能用Pair<Number>，而是要用Pair<? extends Number>

Collections

Interfaces

Collection Class Hierarchy



```

1  public interface Iterable<T> {
2      Iterator<T> iterator();
3  }

```

- `Iterable<T>` 是一个可以被迭代的接口：
 - 定义了一个方法：`Iterator<T> iterator()`，返回一个 `Iterator<T>` 对象。
 - 任何实现了 `Iterable<T>` 的类都可以使用 `foreach` 循环（增强型 `for` 循环）进行遍历。
- `Iterator<T>` 是实现具体迭代功能的接口：
 - 提供迭代的具体方法，比如检查是否有下一个元素（`hasNext()`）、返回当前元素并移动到下一个元素（`next()`）、以及删除当前元素（`remove()`）。

```

1 Set<Integer> s1 = new HashSet<>(Set.of(1, 2, 3, 4));
2 Set<Integer> s2 = new HashSet<>(Set.of(3, 4, 5, 6));
3 // 子集判断
4 boolean isSubset = s1.containsAll(s2); // false
5 Set<Integer> union = new HashSet<>(s1);
6 union.addAll(s2); // union = [1, 2, 3, 4, 5, 6]
7 Set<Integer> intersection = new HashSet<>(s1);
8 intersection retainAll(s2); // intersection = [3, 4]
9 Set<Integer> difference = new HashSet<>(s1);
10 difference.removeAll(s2); // difference = [1, 2]
11 List<Integer> c = List.of(1, 2, 2, 3, 4, 4, 5);
12 Set<Integer> noDups = new HashSet<>(c); // noDups = [1, 2, 3, 4, 5]

```

```

1 List<Integer> a = new ArrayList<>(List.of(1, 2, 3));
2 List<Integer> b = new ArrayList<>(List.of(4, 5));
3 a.addAll(b); // 合并 输出: [1, 2, 3, 4, 5]
4 a.subList(1, 4).clear(); // 删除索引 1 至 3 的元素 输出: [1, 5]
5 List<Integer> c = new ArrayList<>(List.of(1, 2, 3, 4, 5));
6 List<Integer> partView = c.subList(1, 4); // 提取 [2, 3, 4]
7 List<Integer> part = new ArrayList<>(partView); // 创建独立副本
8 partView.clear(); // 清空子列表
9 System.out.println(c); // 输出: [1, 5] c和partView指向同一个对象
10 System.out.println(part); // 输出: [2, 3, 4]

```

```

1 Map<String, Integer> a = new HashMap<>(Map.of("A", 1, "B", 2, "C", 3));
2 Map<String, Integer> b = Map.of("B", 4, "C", 5);
3 boolean isSubMap = a.entrySet().containsAll(b.entrySet());
4 System.out.println("Is SubMap: " + isSubMap); // false
5 Set<String> commonKeys = new HashSet<>(a.keySet());
6 commonKeys retainAll(b.keySet());
7 System.out.println("Common Keys: " + commonKeys); // [B, C]
8 a.keySet().removeAll(b.keySet());
9 System.out.println("After Removing Keys: " + a); // {A=1}

```

Map

- 当执行 `put` 操作 (如 `map.put("key", "value")`) 时:
 - i. **计算哈希值:** 调用键的 `hashCode()` 方法。
 - ii. **确定数组索引:** 根据公式 `hash & (length - 1)` 计算存储桶的索引。
 - iii. **检查冲突:**

- 一个节点储存4个值 **HashCode** **Key** **Value** **Next**(对应同一个HashCode的链表)
- 如果目标存储桶为空，则直接插入新节点。
- 如果目标存储桶不为空，则遍历链表检查是否存在相同键：
 - 如果找到相同键，则更新值。
 - 如果未找到相同键，则将新节点添加到链表（或树）。
- HashMap** 无序 **TreeMap** 自然顺序或者指定的顺序 **LinkedHashMap** 插入顺序

Set

- HashSet** 无序 **TreeSet** 自然顺序或者指定的顺序 **LinkedHashSet** 插入顺序

Algorithms

- Comparable<? super T>** 的原因是如果其父类实现了Comparable也可以

```

1  public static <T extends Comparable<? super T>> T max(Collection<? extends
T> coll) {
2      Iterator<? extends T> i = coll.iterator();
3      T candidate = i.next();
4      while (i.hasNext()) {
5          T next = i.next();
6          if (next.compareTo(candidate) > 0) {
7              candidate = next;
8          }
9      }
10     return candidate;
11 }
```

- Comparable和Comparator([示例](#))

i. Comparable Interface:

- 方法：** `compareTo(T o)` 是类本身实现的一个方法，用于比较当前对象与另一个对象。
- 内部定义排序规则：** 实现了 **Comparable** 的对象可以自己定义如何与其他对象比较。

ii. Comparator Interface:

- 方法：** `compare(T o1, T o2)` 是一个独立的方法，用于比较两个对象。
- 外部定义排序规则：** `Comparator` 通常被实现为一个单独的类，或者作为匿名类/函数传递。

- Collection有很多static方法 比如 `max fill reverse shuffle`等

```
1 List<String> strings;
```

```
2 Collections.sort(strings);  
3 Collections.shuffle(strings);
```

Lecture 3

3 Types of Programming Paradigms

- i. **Objected Oriented Programming(OOP 面向对象编程)** 核心概念：类 对象 封装 继承 多态
- ii. **Procedural Programming** 程序分解为多个过程，每个过程处理一部分，注重怎么做
- iii. **Functional Programming(函数式编程)**

Functional Programming

- i. **Functions first-class citizen** 可以作为函数参数 函数返回值 可以赋值给一个变量
- ii. **No Side Effects:** Pure Function 同样的输出同样的结果 不依赖外部环境
- iii. **Immutability** `x = x + 10` 会改变原来的x，但是 `return x + 10` 则不会
- iv. **Recursion** 通过递归来代替循环

Lambda Expressions

Syntax

```
1 (param1, param2) -> param1 > param2  
2 (param1, param2) -> {  
3     return param1 > param2; // 必须显式使用 return  
4 }  
5 (param1, param2) -> { param1 > param2; } // 会报错，因为缺少显式 return
```

Type Inference: 使用Lambda表达式的时候 编译器会进行Type Inference，这时候就要避免使用Raw Type而是使用Generics，如果用Raw Type给String排序，因为Object没有Length方法，就会报错。

Use Cases

- **Lambda 参数命名规则：** Lambda 表达式的参数不能与外部作用域中定义的局部变量同名。
- **外部变量的限制：**
 - Lambda 表达式只能访问外部作用域中定义的 **final 或有效 final 变量**。 (局部变量是受到限制的 实例变量和静态变量不受限制)

- 有效 final: 变量未显式声明为 `final`, 但它的值在整个作用域内没有改变, 被视为有效 final。

forEach 方法

- `forEach` 是 `Iterable` 接口的一个默认方法。它接收一个 `Consumer` 接口的实现, `Consumer` 是一个函数式接口, 只有一个抽象方法 `accept(T t)`。
- 每次迭代时, `forEach` 会将集合中的每个元素传递给 `Consumer` 的 `accept` 方法

Method References(区别)

i. Static Method `ClassName::staticMethod`

```
1 BiFunction<Integer, Integer, Integer> maxFunction = Math::max;
2 maxFunction.apply(10, 20); // 直接调用, 不需要实例
```

ii. Instance Method(Bound) `InstanceName::instanceMethod`

- 依赖实例 但是在定义的时候就已经绑定好了实例

```
1 String myString = "Hello";
2 // Bound 实例方法引用
3 Function<Integer, Character> boundMethod = myString::charAt;
4 // 调用时不需要提供实例 (因为 myString 已经绑定)
5 System.out.println(boundMethod.apply(1)); // 输出 'e'
```

iii. Instance Method(Unbound) `ClassName::instanceMethod`

```
1 Transformer transformer = String::toLowerCase;
2 transformer.transform("HELLO"); // "hello"
```

iv. Constructor

```
1 Supplier<Person> personSupplier = Person::new;
2 Supplier<Person> personSupplier = () -> new Person();
3 // 调用时会创建一个新实例
4 Person person = personSupplier.get();
5
6 Function<Integer, String[]> arrayFunction = String[]::new;
7 Function<Integer, String[]> arrayFunction = (size) -> new
String[size];
```

```
8 // 调用时会创建一个指定大小的数组  
9 String[] strings = arrayFunction.apply(10); // 创建一个长度为10的字符串  
数组
```

总结来说 1 3的区别在于 3存在一个Receiver而1不存在 比如说toLowerCase就需要一个 Receiver，一个去调用这个函数的实例，但是Min Max就不需要 不是 `5.Min(3)` 这样，就是单纯的 `Min(5,3)`

```
1 "HELLO".toLowerCase(); // "HELLO" 是 receiver
```

- 2 3都存在一个Receiver 但是区别在于 2的Receiver在定义的时候就是一个确定的实例，而3则是一个类

Java Functional Interfaces

- i. `Supplier<T>` : 表示一个无参数的函数，返回一个结果。方法: `T get()`

```
1 Supplier<String> supplier = () -> "Hello, World!";  
2 System.out.println(supplier.get()); // 输出: Hello, World!
```

- ii. `Consumer<T>` : 表示一个接受单一参数但无返回值的函数。方法: `void accept(T t)`

```
1 Consumer<String> consumer = s -> System.out.println(s);  
2 consumer.accept("Hello, World!"); // 输出: Hello, World!
```

- iii. `Function<T, R>` 表示一个接受单一参数并返回结果的函数。方法: `R apply(T t)`

```
1 Function<Integer, String> function = i -> "Number: " + i;  
2 System.out.println(function.apply(10)); // 输出: Number: 10
```

- iv. `Predicate<T>` : 表示一个返回布尔值的函数 方法: `boolean test(T t)`

```
1 Predicate<String> predicate = s -> s.isEmpty();  
2 System.out.println(predicate.test("")); // 输出: true
```

v. `BiFunction<T, U, R>` :表示一个接受两个参数并返回结果的函数。方法: `R`
`apply(T t, U u)`

```
1 BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
2 System.out.println(add.apply(5, 10)); // 输出: 15
```

vi. `UnaryOperator<T>` 和 `BinaryOperator<T>` :是 `Function` 的特例。

```
1 UnaryOperator<Integer> square = x -> x * x;
2 System.out.println(square.apply(5)); // 输出: 25
3 BinaryOperator<Integer> maxOperator = (a, b) -> a > b ? a : b;
4 int result = maxOperator.apply(5, 10); // 返回 10
```

Lecture 4

```
1 // From a Java Collection
2 List<String> list = new ArrayList<String>();
3 list.add("A");
4 Stream<String> stream = list.stream();
5
6 // Stream.generate需要一个Supplier函数式接口，生成的是一个无限流，只要调用就生成。
7 Stream<String> echos = Stream.generate(() -> "Echo");
8 Stream<Double> randoms = Stream.generate(Math::random);
9
10 // Stream.iterate 也是一个无限流，iterate(T seed, UnaryOperator<T> f) seed是
11 // 初始值，函数式接口决定怎么从前一个元素计算出后一个元素
12 Stream<Integer> evenNumber = Stream.iterate(2, n -> n + 2);
13 evenNumber.limit(10).forEach(System.out::println);
14
15 // Stream.of()
16 Integer[] array = new Integer[]{1, 2, 3}; // 有限流
17 Stream<Integer> istream = Stream.of(array);
18 Stream<String> sentence = Stream.of("This", "is", "Java", "2");
19
20 // Primitive Type Streams
21 IntStream stream0 = Arrays.stream(new int[]{1, 2, 3});
22 IntStream stream1 = IntStream.of(1, 2, 3, 5, 8);
23 IntStream stream2 = IntStream.range(5, 10);
24 Stream<String> sentences = Stream.of("This", "is", "Java", "2");
25 IntStream stream3 = sentences.mapToInt(String::length);
```

Intermediate Operation 中间操作是惰性的

```
1 Stream<T> filter(Predicate<? super T> predicate)
2 List.stream()
3     .filter(element -> element % 2 == 0)
4     .forEach(element -> System.out.print(element+ " "));
5
6 <R> Stream<R> map(Function<? super T, ? extends R> mapper)
7 List.stream() // Stream<String>
8     .map(Integer::parseInt) // String<Integer>
9     .forEach(System.out::println)
10
11 // sorted() 或者 sorted(Comparator<? super T> comparator)
12 pointList.stream()
13     .sorted((p1, p2) -> p1.x.compareTo(p2.x))
14     .forEach(System.out::println);
```

Terminal Operations

- i. 标志着流的结束：终端操作是流操作的最后一步，一旦执行，流不再可用。
- ii. 返回非流类型的结果：终端操作的结果可能是：基本数据类型. 引用类型 `void`
- iii. 触发流的执行（Eager execution）：流的中间操作是Lazy Execution。
 - 只有终端操作会触发整个流的执行。减少不必要的计算 如果Terminal Operations是`findFirst()`, 找到一个之后就不需要做剩余元素的filter了，减少了大量计算

i. `reduce()`

```
1 Optional<T> reduce(BinaryOperator<T> accumulator)
2 int sum = arrayList.stream().reduce((a, b) -> a + b).orElse(0);
3 int sum = arrayList.stream().reduce(0, (a, b) -> a + b); // 0既是初始值  
也是默认值
```

ii. `collect()` `<R, A> R collect(Collector<? super T, A, R>`

在 `Collector<T, A, R>` 中，输入流元素 (`T`) 通过 `Collector` 被逐步累积到一个容器 (`A`) 中。最终，容器 (`A`) 被转换成目标结果 (`R`)。

```
1 List<String> items = Arrays.asList("apple", "banana", "orange");
2 List<String> result = items.stream()
    .collect(Collectors.toList());
```

- `T` 是 `String`。 `A` 是 `ArrayList<String>` `R` 是 `List<String>` (最终的结果)。

流在终端操作后会被消费掉，无法再次使用 因此，在代码中如果需要对同一个流调用两个 `.collect()` 操作，可能需要用类似以下的方式重新生成流：

```

1 Stream<String> stream1 = Stream.of("a", "bb", "cc", "ddd");
2 Map<String, Integer> map =
3     stream1.collect(Collectors.toMap(Function.identity(),
4         String::length));
5
6 Stream<String> stream2 = Stream.of("a", "bb", "cc", "ddd");
7 String joined = stream2.collect(Collectors.joining("$"));
8 // T 是 String A 是 StringBuilder R 是 String

```

groupingBy(Classifier, Downstream Collectors)

```

1 Stream<String> stream = Stream.of("1a", "1bb", "1c", "2a", "2a",
2 "2bb");
3 Map<Character, Set<String>> group =
4     stream.collect(Collectors.groupingBy(s->s.charAt(0),
5             Collectors.mapping(s->s.substring(1),
6                 Collectors.toSet()))); // {1=[bb, a, c], 2=[bb, a]}
7
8 Map<Character, Set<String>> group =
9     stream.collect(Collectors.groupingBy(s->s.charAt(0),
10                Collectors.toSet())); // {1=[1a, 1bb, 1c], 2=
11 [2a, 2a, 2bb]}

```

从前往后执行先分类后转换成set `Collectors.mapping(Function<T, V>, Collector)` 本身也是一个下流收集器但是多做了一层映射

Optional<T>

i. `Optional.of()` && `Optional.ofNullable()`

```

1 Optional<String> valueOptional = Optional.ofNullable(null);
2 Optional<String> valueOptional = Optional.of("Hello");
3 Optional<String> optional = Optional.of("Hello");
4 if (optional.isPresent()) System.out.println("Value is present");
5 optional.ifPresent(value -> System.out.println("Value: " + value));

```

```

6  String result = optionalString.orElse("Null"); // 原本有值就返回原本的
    否则返回"Null"
7  String result = optionalString.orElseGet(() ->
    System.getProperty("user.dir"));
8  String result =
    optionalString.orElseThrow(IllegalStateException::new);

```

ii. `map()` && `flatMap()`

- `Get` 方法本质是一个 `Function<T, V>` `Set` 方法本质是一个 `BiConsumer<T, V>`
- `T` 都是这个类对象本身
- `flatMap()` 可以有效处理多层 `Option<>` 的嵌套

Lecture 5

(1) Chinese Telegraph Code, CTC (2) ASCII (3) GB2312, GBK, GB18030 中文占
2Byte (4) Unicode

Java中的char只能表示 U+0000 to U+FFFF (BMP Basic Multilingual Plane), 超出的部分
U+10000 到 U+10FFFF 被称为 **Supplementary Characters**, 需要用int或者string或者

```
1  char[] tree = Character.toChars(0x1F332); // len = 2
```

1. UTF-8(Unicode Transformation Format – 8-bit)

- 在每字节的高位部分加入头部标志 (header bits)

Character Range	Encoding
0...7F	0a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
80...7FF	110a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
800...FFFF	1110a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
10000...10FFFF	11110a ₂₀ a ₁₉ a ₁₈ 10a ₁₇ a ₁₆ a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀

- 使用最少 1 字节, 但根据字符复杂度, 可以使用 2、3 或 4 字节。中文一般占用3个字节(但是其实表示的话只需要两个字节 因为真正的有效位只有16位) 向后兼容 ASCII。
2. UTF-16: 使用至少 2 字节, 但字符复杂时可使用 4 字节。不兼容 ASCII。
 3. UTF-32: 始终使用 4 字节, 直接表示代码点。不兼容 ASCII。

Byte Streams & Character Streams

Byte Stream (字节流)

- 主要用于处理二进制数据（例如图像、音频文件、视频文件）。基于 `InputStream` 和 `OutputStream` 类及其子类。局限性：对于处理 Unicode 文本（如字符数据）不够方便，需要额外的编码和解码。

Character Stream (字符流)

- 用于处理 Unicode 文本数据。基于 `Reader` 和 `Writer` 类及其子类。直接支持 Unicode 编码。

文件读入

- 对于 Character Stream 来说，主要依赖于系统默认的编码方式，中文系统是 `GBK`，英文系统是 `UTF-8`，但是也可以自己指定

```
1  FileReader("src\io\test.txt", Charset.forName("gb2312"))
```

- “计”的 UTF-8 编码是 `E8 BA 91`，占用 3 个字节。通过 Character Stream 读取进来之后用 2 个字节的 char 储存，“计”的 Unicode 码点是 `8B A1`。（因为实际有效的就只有 16 位）
- 对于 Byte Stream 来说，通过使用 `InputStreamReader` 和 `FileInputStream` 来指定编码的方式

```
1  new InputStreamReader(new FileInputStream("file.txt"), "UTF-8");
```

FilterInputStream(Decorator Design Pattern)

```
1  InputStream file = new FileInputStream("src/test.zip"); // 文件读取 -> 缓冲 ->
   解压
2  InputStream bfile = new BufferedInputStream(file);
3  InputStream zfile = new GZIPInputStream(bfile);
```

Scanner

```
1  File file = new File("example.txt");
2  Scanner scanner = new Scanner(file);
3  while (scanner.hasNextLine()) { // Line 和 Int 这些都一样
4      String line = scanner.nextLine(); // 读取一行文本
5      System.out.println("Line: " + line);
6  }
```

```
7 scanner.close();
```

PrintWriter

```
1 PrintWriter out = new PrintWriter("employee.txt", "UTF-8");
2 out.printf("%d", Int);
```

I/O from Command Line

a. System.in

Byte Stream 而不是 Character Stream 但是 Byte Stream 来读取十分的不方便，还要做后续的转换，所以用如果想要 Reader (Character Stream) 需要用 InputStreamReader 来进行衔接

```
1 Scanner scanner = new Scanner(System.in); // 使用标准输入流
2
3 BufferedReader br = new BufferedReader(new
4   InputStreamReader(System.in));
5 String str = "";
6 while (!str.equals("quit")) {
7   str = br.readLine(); // 按行读取用户输入
8   System.out.println(str); // 输出用户输入
9 }
```

b. System.out (PrintStream)

- 本质上是 Byte Stream, 但它通过内部使用的 Character Stream 模拟了字符流的功能，允许我们直接处理文本数据。(类似 PrintWriter). .println .printf .format 等方法
- 通过 System.setOut(out); 可以重定向

```
1 PrintStream out = new PrintStream(new File("test.txt"));
2 System.setOut(out); // 把System.out 的输出重定向到新的位置
3 System.out.println("Hello!"); //就会输出到文件
```

c. System.err 输出报错信息

System.console

- 通过调用 `System.console()` 返回一个 `console` 对象，调用这个 `console` 对象来读入 `readline()` 输出 `printf` `println` 还提供一些高级功能 比如输入密码的时候不显示具体输入

```

1 java.io.Console console = System.console();
2 String name = console.readLine("Enter your name: ");
3 console.printf("Hello, %s!\n", name);
4 char[] password = console.readPassword("Enter your password: ");
5 console.printf("Password entered: %s\n", new String(password));

```

Lecture 6

Persistence and Serialization

- Serialization:** `Object` -> `Byte Stream`

```

1 Student student = new Student("Alice", "CS", 20); // 设置存储字节流的位置
2 FileOutputStream fos = new FileOutputStream("student.ser");
3 ObjectOutputStream oos = new ObjectOutputStream(fos); // 序列化对象
4 oos.writeObject(student); // 将对象写入文件
5 oos.close();

```

- Deserialization:** `Byte Stream` -> `Object`

```

1 FileInputStream fis = new FileInputStream("student.ser");
2 ObjectInputStream ois = new ObjectInputStream(fis); // 反序列化对象
3 Student student2 = (Student) ois.readObject(); // 从文件中读取对象
4 ois.close();

```

Serializable Interface

- 标记接口（Marker Interface）：
 - `Serializable` 是一个没有方法或字段的空接口。它的作用是作为标记，告诉 Java 编译器和 JVM，某个类的对象可以被序列化。实现了 `Serializable` 接口的类不需要添加任何额外方法。
- 默认序列化机制：
 - JVM 提供了默认的序列化机制，使用 `ObjectOutputStream` 和 `ObjectInputStream` 类即可完成序列化和反序列化操作。

- i. `SerialVersionUID` 是类的 `private static final long` 字段用于标识类的版本，检查一致性需要显式定义和维护
- ii. `Serial Number` 由JVM内部自动管理，为每一个对象都创建一个 `Serial Number`，如果一个对象内部定义了一个对象，两个对象都会有一个序列号，比如两个 `ClassRoom` 对象里都有同一个 `Student` 对象，则序列化的时候两个 `Student` 是完全一样的
 - 用于标识序列化过程中单个对象的引用关系 无需开发者指定 避免重复存储对象，维护引用关系

Transient:如果直接序列化会将所有类的元数据以及实例信息每个字段全部储存，通过 `transient`关键字可以使得某些字段不被序列化。

```

1  public class StringList implements Serializable {
2      private transient int size = 0;
3      private transient Node head = null;
4      private static class Node { // 没有实现 Serializable 接口，默认情况下它
不可序列化
5          String data; Node next; Node previous;
6      }
7      private void writeObject(ObjectOutputStream oos) throws IOException
{
8          oos.defaultWriteObject(); // 写入类的元数据和非Transient字段
9          oos.writeInt(size);
10         for (Node n = head; n != null; n = n.next) {
11             oos.writeObject(n.data);
12         }
13     }
14 }
```

Malicious Attack: 可以重写 `readObject` 来检查序列化之后的数据是否被篡改 加入一些基本的检查来检验是否数据被修改

Working with Files

```

1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3
4  Path p1 = Paths.get("resources");
5  Path p2 = Paths.get(args[0]);
6  Path p3 = Paths.get(URI.create("file:///D:/CS209A/sample123.txt"));
7
8  Path rp = Paths.get("C:\\\\Users\\\\admin\\\\test\\\\..\\\\CS209A_Lectures");
```

```

9 System.out.format("rp2 normalize: %s%n", rp.normalize());
//C:\Users\admin\CS209A_Lectures

```

通过树状结构来储存路径 / 和 C:\ 开头为绝对路径 储存在根结点

Method Invoked	Returns in the Solaris OS	Returns in Microsoft Windows
toString	/home/joe/foo	C:\home\joe\foo
getFileName	foo	foo
getName(0)	home	home
getNameCount	3	3
subpath(0, 2)	home/joe	home\joe
getParent	/home/joe	\home\joe
getRoot	/	C:\

- toAbsolutePath(): 将相对路径转换为绝对路径。不检查文件或目录是否真实存在。
(Java中的\是强行转义符 所以需要\\)

```

1 Path cp = Paths.get("resources\\..\\resources\\math.txt");
2 System.out.println(cp.toAbsolutePath()); //
C:\Users\admin\CS209A_Lectures\resources..resources\math.txt

```

- toRealPath(): 返回路径的真实路径，同时检查文件或目录是否存在。自动去除路径中的冗余部分 (如 . 和 ..)。如果文件不存在，抛出 NoSuchFileException。

```

1 Path cp2 = Paths.get("resources\\..\\resources\\notexist.txt");
2 System.out.println(cp2.toRealPath()); // Throws NoSuchFileException
3 static Path createFile(Path path, FileAttribute<?> attrs)
4 static Path createDirectory(Path path, FileAttribute<?> attrs)
5 static Path copy(Path from, Path to, CopyOption)
6 static Path move(Path from, Path to, CopyOption)
7 static long copy(InputStream from, Path to, CopyOption)
8 static long copy(Path from, OutputStream to, CopyOption)
9 Files.list(Path) // 只列出第一层级的文件夹和文件
10 Files.walk(Path) // 遍历所有 按照DFS的顺序输出

```

- 用Scanner | Files.lines 读取Path下的文件

```

1 Path file = Paths.get("resources", "math.txt");
2 Scanner in = new Scanner(file);
3 while (in.hasNext()) {System.out.println(in.nextLine());}
4
5 try (Stream<String> stream = Files.lines(file)) {
6     stream.forEach(System.out::println);
7 } catch (IOException e) {e.printStackTrace();}

```

Exception Handling

Definition

- **Checked**代表可以预见的外部环境问题。所以需要开发者去捕获去处理（假设他会发生）
Unchecked指的是运行时错误
- **Error (unchecked)**: 也可以捕获但是不推荐 内部错误或资源耗尽等严重错误，由 JVM 抛出，程序通常无法继续运行。内存不足 栈溢出
- **Exception**
 - RuntimeException (unchecked)**: 可以用 `try-catch` 或者 `throw` 抛出 抛出都可以 `NullPointerException` `IndexOutOfBoundsException` 通常是编程内部错误，需要修正代码逻辑，比如空指针下标越界
 - Other Exception (checked)** : 用 `try-catch` 或者 `throw` 抛出 一定要抛出不然会CE `IOException` `SQLException` 外部环境问题，比如文件不存在，SQL出现错误等，需要去捕获处理

Process

- 异常沿着方法调用栈向上传递，直到找到匹配的 `catch` 块。如果没有任何方法处理异常，最终由 JVM 处理，通常会导致程序终止。[\(Example\)](#)
- **抛出**：方法无法处理或问题需要更高层决策 **捕获**：方法知道如何恢复或处理问题
- 多异常处理

```

1 #1 catch (FileNotFoundException | SQLException | SocketException e)
2 #2 catch (Exception e)

```

- method2是不太好的，没有区分具体的异常类型，可能导致某些特定异常的处理逻辑丢失。
- **finally 子句的作用**

- **保证清理操作:** `finally` 子句中的代码无论是否抛出异常，都会被执行。并且如果抛出了异常并且没有对应的catch，`finally` 会在向上层抛出异常之前执行，然后抛出异常，再中断当前这一层的任务。**所以 try-catch-finally 块后面的代码不会被执行**
- **try-with-resources** 文件操作 数据库连接 网络连接

```

1 // Origin
2 Resource res = null;
3 try {res = new Resource();} finally {
4     if (res != null) res.close(); // 手动关闭资源
5 }
6 // New
7 try (var in = new Scanner(new
8     FileInputStream("/usr/share/dict/words"));) {
9     while (in.hasNext()) {
10         System.out.println(in.next().toUpperCase());
11     }
12 } // 此处会自动关闭资源，无需手动调用 in.close() 无论顺利执行还是报错 都会关闭
13 // var是一个局部变量类型自动推断的工具 (c++的auto)

```

Lecture 7

- **Process && Thread**
 - 一个程序比如微信 WPS都是一个 `Process`，一个 `Process` 包含多个 `Thread`，OS系统为不同的 `Process` 分配空间和资源，同一个 `Process` 中的 `Thread` 共享内存和资源

Creating & Starting Threads

- JVM是一个单独的 `Process` `Main thread` 会在程序开始时自动创立，主进程有能力创建额外的进程
- a. **继承 `Thread` 类 (不推荐)** :重写 `Thread` 类中的 `run()` 方法。Java 不支持多继承，如果继承其他类，就无法再继承 `Thread`
- b. **实现 `Runnable` 接口 (推荐)** :在实现类中重写 `run()` 方法。使用 `Thread` 类的构造方法，将 `Runnable` 对象传入并启动线程。
- 通过new一个实现 `Runnable` 接口的类，然后用 `start()` 方法来运行而不是 `run()`
 - 如果用 `run()` 来运行，则不会用新创立的线程来跑，都是由 `main` 线程来负责执行，并且是顺序执行而不是并行 `start()` 的含义就是创立一个新的线程然后运行 `run()` 并且是非阻塞的。
- **Java 的 `Thread` 类有一个构造函数，接受一个实现了 `Runnable` 接口的对象作为参数：** `Thread(Runnable target)`

```
1 Runnable cat = () -> System.out.println("Cat thread");
2 Thread thread = new Thread(cat);
3 thread.start(); // 启动线程
```

Synchronization

- 因为数据资源的共享 如果两个进程同时对数据进行修改，因为读取写入不同步的问题，会导致数据的紊乱。

a. Synchronized()

- 存在一个 `intrinsicLock` 同一个对象的所有 `synchronized` 方法共享这一个内在锁
- 通过 `wait()` 和 `notifyAll()` 来实现 `condition`

```
1 // Method1 锁定部分数据 锁类似于token 只有拿到这个token的线程才能对临界区的代码具有执行的权限
2 private final Object balanceLock = new Object(); // 定义一个单独的锁对象
3 synchronized (balanceLock) { // 只锁住余额操作
4 }
5 // Method2 直接在函数名字前面添加synchronized关键字
6 public synchronized void withdraw(double amount) {
7 }
```

b. ReentrantLock()

```
1 Lock lock = new ReentrantLock();
2 lock.lock();
3 try {} finally {
4     lock.unlock();
5 }
```

o Condition(暂时解开锁 让别的线程操作 直到满足我的条件再切回控制权)

- 每个Condition都必须有对应的lock，一个lock可以有多个condition 通过 `newCondition()` 来绑定
- 通过调用 `condition.await()` 来实现带条件的线程锁
- 另一线程在同一 `Condition` 对象上调用 `signalAll()` 或 `signal()` 方法通知时，该进程会继续执行

```

1 // Synchronized
2 while (balance < amount) { // 如果余额不足
3     wait(); // 等待其他线程存入足够的金额
4 }
5 notifyAll(); // 另外一个线程 使用
6
7 // ReentrantLock
8 private Lock balanceLock;
9 private Condition sufficientCondition;
10 sufficientCondition = balanceLock.newCondition();
11 balanceLock.lock();
12 try {
13     while (balance < amount) { // 如果余额不足
14         sufficientCondition.await(); // 等待其他线程存入足够的金额
15     }
16     ...
17 } finally {balanceLock.unlock();}
18 sufficientCondition.signalAll(); // 另一个线程使用 如果signal之后了发现仍未满足条件 原本condition会继续await

```

State

- a. NEW: `Thread thread = new Thread();`
- b. RUNNABLE: `Thread.start()` 线程已准备好运行，但是否运行完全取决于CPU是否可
- c. BLOCKED: 当想要获取 `lock`，但这个 `lock` 在别的Thread的手里，就进入 BLOCKED 状态
- d. WAITING
 - `Object.wait()` 或者 `Thread.join()` (等待某个线程完成之后继续，否则就等待，比如当前线程是A，然后正在执行的线程是B，在A中执行 `B.join()`，A就会等待B线程执行完之后继续)
 - 直到 `notifyAll()` `signalAll()`
- e. TIMED_WAITING: `Thread.sleep(1000)`
- f. TERMINATED: `run()` exits normally or exits due to uncaught exception

Thread-safe Collections

- 普通的collection的子类 (`ArrayList`、`HashMap`) 不具备线程安全的功能，因为兼顾性能安全会大幅拖慢单进程的效率。`Vector` 和 `Hashtable` 是线程安全的，但是效率低下

1. CopyOnWrite

- 例子: `CopyOnWriteArrayList` `read >> write`
- `Read` 不阻塞 `Write` 不阻塞 `Read` 但是会通过 `reentrant lock` 阻塞 `Write`

2. Collections using Lock(Segmented Locking)

- 把一个collection拆分成多个部分 分开 lock 从而提升并发能力

例子：BlockingQueue

- 阻塞行为：
 - 队列满时添加元素： put() 生产者线程试图向满的队列添加元素时会阻塞，直到队列有空间。
 - 队列空时移除元素： take() 消费者线程试图从空的队列移除元素时会阻塞，直到队列有元素。
- 底层实现机制：使用 ReentrantLock 和 Condition 提供高效的线程间协调。
- 作用：阻塞队列为多线程之间的任务协调提供了一个高效的方式。

3. Compare-And-Swap(CAS) Collections

- 创建该变量的一个本地副本，保存其当前值（old value）。根据当前变量值，计算目标的 new value。比较变量的当前值（指的不是计算出来的值 而是重新读取该变量现在储存的值）和 old value 是否相同。
 - 相同，没有其他线程修改，则更新为 new value 不同，变量已被其他线程修改，则重试操作

Tasks and Thread Pools

线程池的优势

性能优化：减少线程创建和销毁的开销。通过线程复用提高系统吞吐量。

资源管理：限制线程的数量，防止因线程过多导致系统资源耗尽。

任务调度：提供统一的任务提交接口（如 ExecutorService），方便任务的管理和调度。

解耦逻辑与执行：程序员只需关注任务的实现，任务的执行由线程池负责。

Runnable 与 Callable 的对比

特性	Runnable	Callable
返回值	无返回值	有返回值，通过泛型 <V> 指定
抛出异常	不支持抛出受检异常	支持抛出受检异常
核心方法	void run()	V call() throws Exception
使用场景	不需要返回值的异步任务	需要返回值或可能抛出异常的异步任务
常见组合工具	Thread、Executor	ExecutorService、Future、CompletableFuture

ExecutorService

- `ExecutorService` 是一个Interface，`ThreadPoolExecutor` 实现了 `ExecutorService` 这个接口
- 可以将多个任务提交到 `ExecutorService`，由其管理任务的执行。（`submit()` 或 `execute()`）
 1. 使用 `submit()` 提交任务：返回一个 `Future` 对象，用于获取任务的执行结果或状态。

```
1 Future<T> submit(Callable<T> task)
2 Future<?> submit(Runnable task) // 虽然是Runnable但也会返回一个null
```

2. 使用 `execute()` 提交任务：不返回结果，仅用于执行任务。

- 内部包含一个线程池，用于复用线程资源，减少线程创建和销毁的开销。
 1. **FixedThreadPool**：有限的线程数量，适合固定数量的长期任务，避免频繁创建和销毁线程。
 2. **SingleThreadExecutor**：一个只有一个线程的线程池，所有任务按顺序执行
 3. **CachedThreadPool**：线程数没有固定限制。空闲线程的默认存活时间为 60 秒，超过会被销毁。

```
1 ExecutorService executorService = Executors.newFixedThreadPool(3);
2 ExecutorService executorService =
3 Executors.newSingleThreadExecutor();
4 ExecutorService executorService = Executors.newCachedThreadPool();
```

- 提交的任务存储在一个阻塞队列（`Blocking Queue`）中，等待线程池中的线程执行。
- `Callable` & `Runnable` & `Controlling Groups of Tasks`

```
1 ExecutorService executor = Executors.newFixedThreadPool(4);
2 Callable<String> task = () -> "Task 1";
3 Future<String> future = executor.submit(task);
4 String result = future.get(); // 可能会阻塞
5 executor.shutdown();
6
7 ExecutorService executor = Executors.newFixedThreadPool(4);
8 Runnable task = () -> System.out.println("Task 1");
9 executor.execute(task); // 用submit其实也行但是没必要
10 executor.shutdown();
```

```
11
12 <T> T invokeAny(Collection<? extends Callable<T>> tasks)
13 <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
14 // 无论原本的collection (Set Map List) 是什么都返回一个结果的list
```

Lecture 8

Socket Programming

- **ServerSocket**

```
1 ServerSocket serverSocket = new ServerSocket(1234); //在localhost的1234端口
    创建一个ServerSocket
2 Socket socket = serverSocket.accept();
3 while (true) {
4     Socket clientSocket = serverSocket.accept(); // 接收客户端连接
5     Scanner in = new Scanner(clientSocket.getInputStream());
6     PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
    true);
7     // 如果是False 需要手动out.flush() 调用out.println数据会先写进缓冲区 flush了
    之后才发送
8     // 发送最好要用println 遵守协议 如果用print 另外一边用readline的话 无法标识结
    尾
9     threadPool.execute(() -> handleClient(clientSocket));
10 }
```

- **Socket (Client Socket)**

```
1 Socket socket = new Socket("127.0.0.1", 1234); //连接Server
2 // 客户端的端口号是由操作系统自动分配的
3 Scanner in = new Scanner(socket.getInputStream());
4 PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
```

Getting Web Data

HTTP

- Port Number: **80 & 8080** 常见请求 **GET** **HEAD** (只请求header information) **POST** **PUT** **DELETE**
- 可以用正常的 **Client-Server Socket** 来进行请求 但是HTTP请求有自己的格式，需要手动format很麻烦

- 下面两个属于 **High Level API** 直接用Socket属于 **Low Level API**
- URLConnection 支持 HTTP (Hyper Text Transfer Protocol) 和 FTP (File Transfer Protocol)**

```

1 URL u = new URL("http://example.com");
2 URLConnection conn = u.openConnection();
3 HttpURLConnection httpConn = (HttpURLConnection) conn;
4 httpConn.setRequestMethod("GET");
5 int code = httpConn.getResponseCode(); // 用这个方法就会发送请求
6 String msg = httpConn.getResponseMessage();
7 InputStream istream = httpConn.getInputStream(); // 读取数据

```

HttpClient

```

1 HttpClient client = HttpClient.newHttpClient();
2 HttpRequest request = HttpRequest.newBuilder()
3     .uri(URI.create("http://example.com"))
4     .GET()
5     .build();
6 HttpResponse<String> response = client.send(request,
7     HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());

```

REST API (Representational State Transfer API)

资源导向：

- 每个 URI 表示一个资源。例如，<https://api.example.com/users/1> 表示用户 ID 为 1 的资源。一般都用名次的复数形式 标识资源的含义是一般都用名次作为URL 而不是 **FetchData** 这种
- 一个请求 **URL + Resources(Path) + Parameter**



Statelessness 每个请求都是独立的，不依赖于之前的请求。客户端每次请求中需提供必要的信息 (Token)

Client-Server Model: 客户端负责用户界面，服务器负责数据存储和逻辑处理。两者相互独立，方便扩展。

Uniform Interface: 使用统一的约定来操作资源，如标准的 HTTP 方法: `GET` `POST` (创建)

`PUT` (更新) `DELETE`

Return Type: 返回一个 JSON (JavaScript Object Notation) 或者 XML

Lecture 9

Overview

- **GUI(Graphical User Interface)**

- 抽象窗口工具包 (Abstract Window Toolkit, AWT) `Swing` `JavaFX`

- 使用 JavaFX

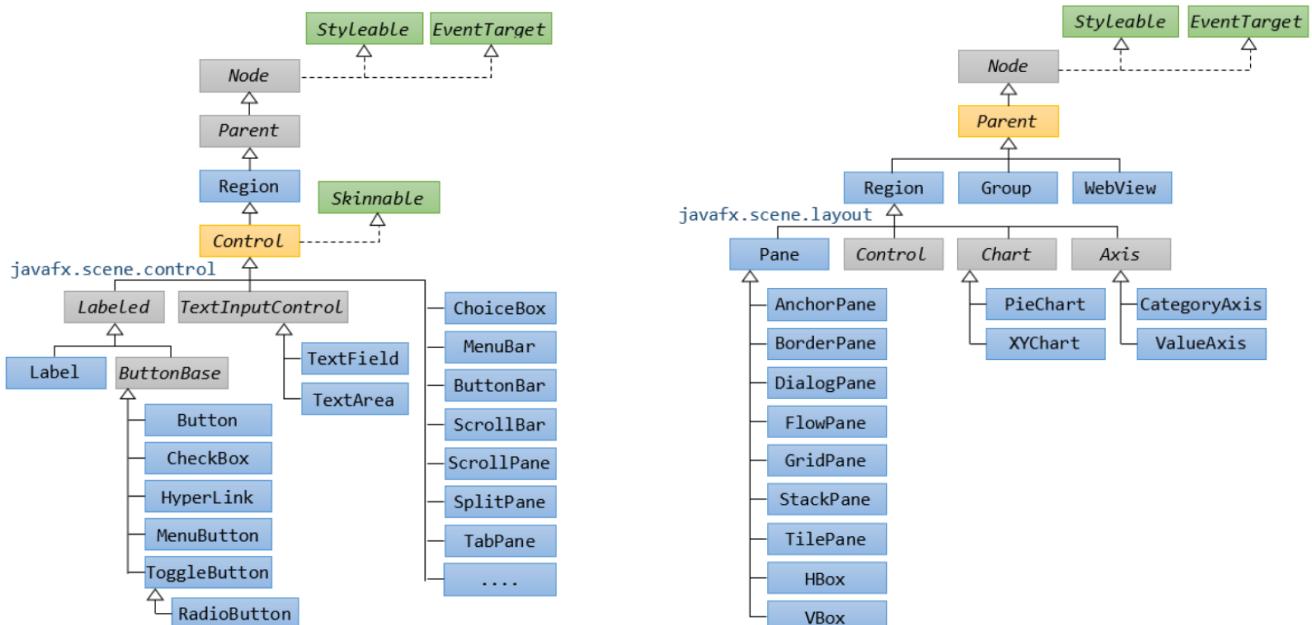
- 继承 `javafx.application.Application`，必须实现 `Application` 类的 `start()` 方法, `start()` 方法接受一个默认参数 `start(Stage primaryStage)`
- 使用 `Application.launch(args);` 启动 JavaFX 应用程序。

Design & Concepts

1. **Stage (窗体)** 是 JavaFX 应用程序的外框，通常对应于一个窗口，一个 JavaFX 应用程序可以包含一个或多个 Stage (即可以同时打开多个窗口)。

```
1 Stage stage = new Stage();
2 stage.setTitle("New Window");
3 stage.show();
```

2. **Scene (场景)**



- 定义: Scene 包含显示在窗口中的所有 GUI 组件，用于在 Stage 上显示内容。

- 特点：

i. Scene 必须通过 `stage.setScene(scene)` 方法设置到一个 Stage（窗体）上。一个 Stage 一次只能显示一个 Scene，但可以在运行时更换不同的 Scene。

ii. 一个 JavaFX Scene 是通过指定一个 **根节点（Root Node）** 创建的。根节点是场景图（Scene Graph）中的顶层节点，所有 GUI 组件都作为根节点的子节点组织成一棵树。

1. **Root Node（根节点）**：场景的顶层节点（例如 `VBox`、`StackPane` 等布局容器）。

2. **Branch Node（分支节点）**：可以有子节点的节点，通常是容器。

a. `Group`：

- 最简单的根节点类型，不会对子节点的布局进行任何约束。
- 对 Group 进行的任何变换（例如旋转）、效果或状态都会应用到该 Group 的所有子节点上。

b. `Region`（如 `Pane`）：Java 控件（Control）的基础

i. `BorderPane`

ii. `Hbox & Vbox Pane`

c. `WebView`：用于显示 HTML 内容的节点。

3. **Leaf Node（叶子节点）**：没有子节点的节点，通常是控件（如按钮、文本框等）。

`root.getChildren().add(button);` 每个子节点的位置都是相对于父节点来说的

`setOnTargetType(EventHandler<TargetEvent> v)`

`setOnMouseClicked`

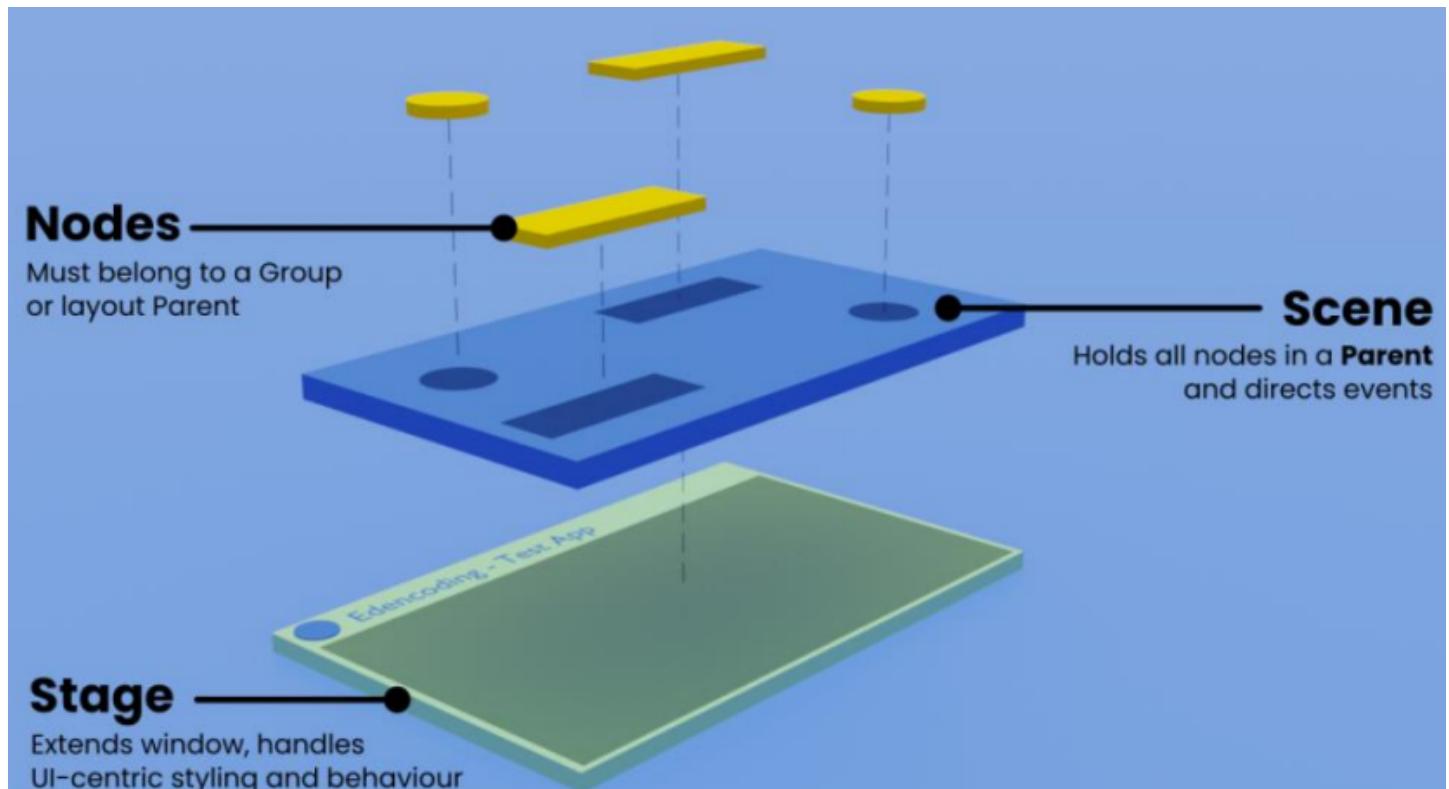
`MouseEvent`

Functional Interface

3. Scene Graph（场景图）

○ **定义：**所有的可视化组件（如控件、布局等）都附加在一个 Scene 中，这些组件的层级结构称为 Scene Graph（场景图）。

○ **作用：**场景图是 GUI 元素的层次结构，用于组织和管理界面中的各个元素。



- **JavaFX Stage Modality**

模态类型	阻止对象	场景
NONE	不阻止任何窗口	独立窗口，如帮助窗口
WINDOW_MODAL	只阻止拥有者窗口的交互	设置窗口、文件打开对话框
APPLICATION_MODAL	阻止应用程序中所有窗口的交互	登录窗口、确认窗口（必须先处理完成）

- **Skinnable** 接口

- 分离视觉表现与行为逻辑
 - **Skinnable** 接口允许为控件定义一个 **Skin** 对象。
 - **Skin** 对象控制控件的视觉表现（外观），而控件本身（比如 **Button** 或 **TextField**）负责处理交互逻辑（如点击、输入等）。
- 支持 CSS 样式化

Region 和 Control 的区别

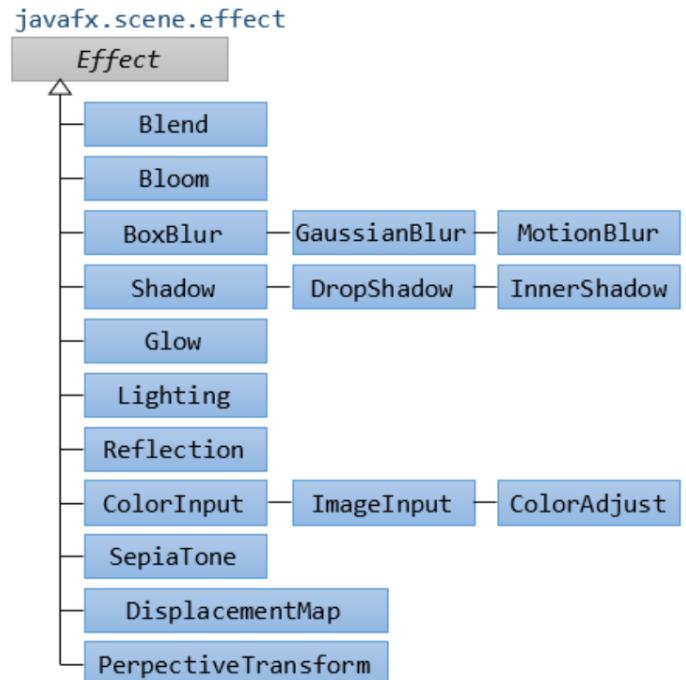
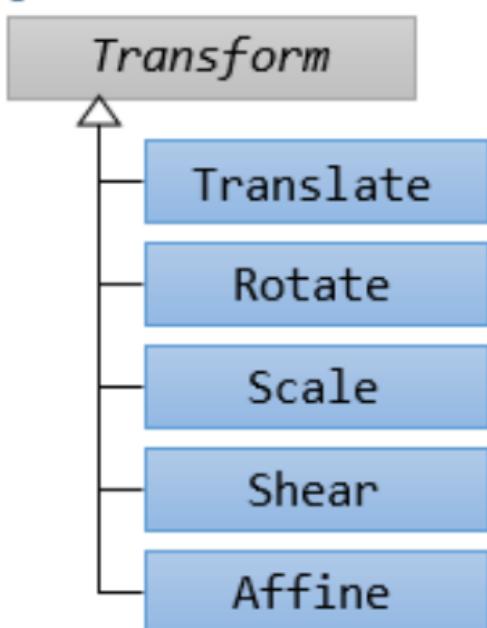
特性	Region	Control
功能	仅提供布局能力和样式支持	提供布局、样式支持，并且具备交互能力
使用场景	作为布局容器（如 <code>Pane</code> 、 <code>VBox</code> ）	作为交互控件（如 <code>Button</code> 、 <code>TextField</code> ）
事件支持	不直接支持交互事件	支持鼠标、键盘等用户输入事件
样式支持	支持 CSS 定义样式	支持 CSS，并可通过皮肤（Skin）自定义外观
继承结构	直接继承自 <code>Parent</code>	继承自 <code>Region</code> ，扩展了交互和逻辑功能

- **Chart**

- `Title` `Legend` (图例) `ChartContent`
- `PieChart` `XYChart`

Transformation, Animation, Effects

`javafx.scene.transform`



FXML

- `motivation`：
 - 分离设计与逻辑
 - 提高代码可维护性
- **FXML** 是一种基于 XML 的标记语言，用于描述 JavaFX 用户界面。
 - `.fxml` 设计用户界面
 - `Controller class` 负责应用逻辑

Multithreading in JavaFX

- 在 JavaFX 中，界面（UI）更新是一个线程敏感的操作，JavaFX 强制要求 **所有的 UI 更新和场景图的修改必须在 JavaFX 应用线程中进行**。为了确保 UI 更新的线程安全性，以避免出现多线程引发的竞态条件和不一致状态。
- 不应该在任务（逻辑）中直接操控UI，会导致tight coupling（高耦合），如果任务执行时间很久，UI界面就会卡住
- 正确的方式：

耗时的任务：

- 放到 **后台线程** 中处理（如通过 `Thread` 或更高级的工具如 `Task`）。
- 这可以避免阻塞 JavaFX 主线程，从而保证用户界面（UI）始终响应。

更新 UI：

- 在后台线程中，通过 `Platform.runLater()` 把更新 UI 的操作交给 JavaFX 主线程处理。
- 这样可以确保对 UI 的修改是线程安全的，符合 JavaFX 的线程规则。

- Task 类：**

- 用于实现异步任务，专门为 JavaFX 应用程序设计。
- 特点：
 - 它是 `Runnable` 和 `Future` 的子类，可以与线程或执行器一起使用。
 - 提供了直接更新 UI 的方法，例如 `updateProgress()` 和 `updateMessage()`，这些方法会自动在主线程中执行。