

Lecture 1:

Hailstone sequence starts with a number n

the next number in the sequence is $n/2$ if n is even, or $3n+1$ if n is odd

it ends when it reaches 1

for example:

- $\text{hailstone}(5) = [5 \ 16 \ 8 \ 4 \ 2 \ 1]$
- $\text{hailstone}(3) = [3 \ 10 \ 5 \ 16 \ 8 \ 4 \ 2 \ 1]$
- $\text{hailstone}(8) =$
- $\text{hailstone}(2^n) =$

we still don't know if every hailstone sequences always reaches 1

在 JAVA 中，必须在使用之前声明变量的 type (Python 不是如此)

一个 type 是值的集合，包括了这些值可以进行的操作；

JAVA 有 8 种 primitive types: boolean, byte, char, short, int, long, float and double；

同样也有 object types: String, BigInteger

Java 惯例中，primitive types 为小写，而 object types 以大写字母开头

operation:

operation 是接收输入，产生输出值的函数

有三种句式：

As an infix, prefix, or postfix operator —— $a + b$, 其中 $+ : \text{int} \times \text{int} \rightarrow \text{int}$

As a method of an object —— `bigint1.add(bigint2)`, 其中 `add : BigInteger × BigInteger → BigInteger`

As a function —— `Math.sin(x)`, 其中 `sin : double → double`

其中一些 operation 是 overloaded 的，同样的 operation 的名字被用于不同的 types：

例如：算术 operator $+$, $-$, $*$, $/$ 对于数字 primitive types, 是 overloaded 的

Java 是 statically-typed 语言；

在编译的时候（程序运行之前），所有的变量的 type 都需要知道，然后编译器可以因此推断所有的表达式的 types

例如，如果 a 和 b 被声明为 int，那么编译器会总结出 $a+b$ 也是 int

而在 dynamically-typed 语言中，例如 Python 或者 Javascript，这种检查推迟到运行时（在程序运行时）

Static typing 是一种特殊的 static checking, 意味着在编译的时候检查 bug;
Static typing 能够阻止很大一部分 bug 影响程序, 实际上, 阻止了将操作应用于错误 type 的参数从而产生的 bug;
· 例如, "5" * "6", 这是将两个字符串类型的变量相乘了, static typing 会在你仍然在写代码时发现这个错误, 而不是等到这行代码被执行时。

String:

```
String hello = "Hello"  
hello.length()  
hello[0]  
"a" + "bc"  
hello.substring(x)  
hello.substring(x, y)  
hello.contains("lo")  
hello.indexOf("lo")  
hello.equals("hello")
```

Arrays:

Array 有固定的长度;
int[] a = new int[100]
一个 array 一旦创建, 长度无法改变
a[2]
a[2] = 0
a.length

List and ArrayList:

List 长度可变
List< Integer > list = new ArrayList< Integer >()
list.get(2)
list.set(2, 5)
list.add(5)
list.size()

通过数组创建 list:

```
List< Integer > list = Arrays.asList(10, 20, 30)  
List 是一个 interface, 也就是一个无法直接用 new 来构建的 type, 而是指定 List 必须提供的操作:  
ArrayList 是一个类, 提供这些操作的实现的具体类型;  
ArrayList 不是 List 类型的唯一实现, 尽管是最常用的, (LinkedList)
```

通常都写 List< Integer > 而不是 List< int >, 因为 List 只知道如何处理 object types, 而不

会处理 primitive types:

在 Java 中，每个 primitive type 都有一个对应的等价 object type;

Java 需要在用<尖括号>参数化类型时，使用 object type

Lecture 2:

一种语言可以提供三种自动检查的方式：

1. Static checking: 在程序运行之前，bug 会被自动发现
2. Dynamic checking: 当程序执行的时候，bug 会被发现
3. No checking: 语言完全不帮助找到错误，不得不自己找到或者以出错告终。

不用说，静态的发现 bug 比动态好，动态比没有好

Static checking:

syntax errors: 例如额外的标点或者 spurious words

错误的名字：例如，Math.sine(2) (The right name is sin)

错误的参数数量： Math.sin(30, 20)

错误的参数类型： Math.sin("30")

错误的返回值类型： return "30"; 函数定义返回的是 int

Dynamic Checking:

非法的参数值，例如 x/y，其中 y=0;

无法表示的返回值，即当特定的返回值不能以类型表示时；

超出范围的索引：在字符串上使用负数或太大的索引

在一个 null 对象的 reference 上调用方法

对比：

静态检查往往关于数据类型，错误往往独立于变量的特定值：

1. 回想一下，类型是一组值。 静态类型保证变量将具有该集合中的某个值，但是直到运行时我们才确切知道它具有哪个值
2. 所以如果错误可能会由某些特定的值导致，例如 0 做除数，或者超出范围的索引，这些问题在编译器中不会产生一个静态的错误

动态检查则相反，通常都是由特定值导致的错误。

No Checking:

Java 中的一个陷阱——和其他许多编程语言类似，它的原始数字类型具有极端的情况，其行为不像我们过去所使用的整数和实数。

所以，一些本该由动态检查出来的错误，完全没有被检测：

陷阱有：

1. Integer division: $5/2$ 并不返回一个分数，而是返回一个整数。因此，这是一个我们可能希望是动态错误的示例（因为分数不能表示为整数）经常会产生错误的答案！
2. Integer overflow: `int` 和 `long` 类型其实是整数的有限集合，有最大值和最小值；当答案绝对值过大的时候，计算会悄悄地溢出（环绕），并从合法范围内的某个地方返回一个整数，但返回的答案不正确。
3. Special values in floating-point types: 浮点数类型，例如 `double`，有几个不是实数的特殊值：`Nan` (which stands for "Not a Number")，`POSITIVE_INFINITY`，和 `NEGATIVE_INFINITY`。这导致某些本应被动态检查出来的错误会得到以上结果，从而最终错误。

Javadoc Comments and Documenting Assumptions:

Assumptions:

在 method signature 中，`int n` 意味着 `n` 始终代表一个 `integer`。永远不会是一个 `string` 或者其他类型——Java 实际上在编译的时候检查了这一个 assumption (static checking)，并且确保程序中都满足这个 assumption。

不幸的是，Java 并不会自动检查别的 assumption。

为什么需要写下 assumption？因为编程充满了 assumption；如果不写下来，会记不住，别人看代码也看不懂，只能猜。

编程需要抱有两个目标：

1. 与计算机交流：首先说服编译器，程序是正确的——语法正确和类型正确——接着确保逻辑正确，最终确保答案正确。
2. 与其他人交流：让所写的程序更有可读性，当其他人不得不修改，改善以及未来的时候，会很有用。

Java 帮助我们区分 specification(what does it do?) 和 implementation(what is the code?)

`List` 和 `Map` 都是 interfaces：它们定义了这种类型的工作方式，但是他们不提供实现的代码；

有很多优点，但是其中一个潜在的优点是，我们可以根据情况不同选择不同的具体的 implementations。

```
List list1 = new ArrayList(); List list2 = new LinkedList();
```

如果左右的通用类型参数相同，则 Java 可以推断发生了什么并为我们节省一些输入： List

```
list1 = new ArrayList<>(); List list2 = new LinkedList<>();
```

ArrayList 和 LinkedList 是 List 的两种 implementations

都提供 List 的所有操作，这些操作必须符合 List 的文档中的描述

Snapshot Diagrams:

我们画图描述在运行时候发生的事情会很有用，为了理解细微的问题；

Snapshot diagrams 表示程序在运行时的内部状态——它的 stack（方法进度和它们的局部变量）及其 heap（当前存在的对象）

我们使用 snapshot diagram：

1. 通过图片互相交流
2. 阐释例如 primitive types 和 object types, immutable values 和 immutable references, pointer aliasing, stack 和 heap, abstractions 和 concrete representations 这种概念。

尽管本课程中的图使用 Java 的示例，但是该符号可以应用于任何现代编程语言，例如 Python, Javascript

Primitive Values:

原始值由 bare constant 表示——这个箭头的来源是一个值的 reference，指向一个变量或者 object field

在最简单的形式中，snapshot diagram 展现了一个变量，有一个箭头指向它的值。

Object Values:

这是一个由其类型作为标签的圆形

当想要展示更多的细节时，可以将 field 的名字写在里面，加上一个箭头指向他们的值。

更多的细节：

1. field 可以包括他们声明的种类： int x
2. 也可以写成 x: int

Reassigning Variables vs Mutating Values:

Snapshot diagrams 给我们一个可视化更改变量和更改值之间的区别的方法：

1. Reassigning Variable：当你分配一个变量，你在改变变量的箭头，使其指向不同的值
2. 当改变一个 mutable value 的内容时，比如一个 array 和 list，你在改变这个值中的 references——这被称为 mutating 这个值

Immutable Values:

string 是一个 immutable 值的例子，可以将其从 "a" reassign 为 "ab"

immutable 类型是一种，创造了之后，值就再也不会改变——immutable 值在 snapshot diagram 中是一个双层圆

相反的，StringBuilder 是一个 mutable 值，也表示了字符串，它具有更改对象值的方法

这两种 snapshot diagram 看起来很不同：这是好事，帮助我们确保代码安全。

Immutable References:

Java 同样会给我们 immutable reference：变量一旦被分配就永不 reassing，想要让一个 reference 变成 immutable，声明的时候需要加上 final：final int n = 5

如果 Java 编译器不相信最终变量在运行时只会分配一次，那么它将产生编译器错误。因此最终给了 immutable reference 静态检查

在 snapshot diagram 中，immutable reference(final) 用双箭头赋值，

Reference vs Value:

注意，可以有一个 immutable reference 却拥有 mutable value(例如: final StringBuilder sb)，可以在指向同一个 object 的时候，改变它的值

我们同样可以有一个 mutable reference 却拥有 immutable value(例如: String s)，该变量的值可以改变，因为它可以重指向另一个 object

Final:

final 可以使用于 parameters 和 local variables

1. 当使用在 parameter 上时，final 意味着 parameter 会在方法被调用的时候就赋值，在此之后，在方法的主体部分中无法被重分配
2. 当使用在 local variable 上时，final 意味着 variable 在第一次赋值之后无法被重分配，直到 variable 的作用域结束

final 可以使用于任何种类的变量，不止是 int 这种 immutable 种类，list 这种 mutable 种类也可以。

如果一个 final 变量指向一个 mutable 的 object，该变量无法被重分配，但是这个 object 所指向的值可以改变，例如 list 中可以调用 add()

Map:

一个 map 存储了 key/value 的对子，每一个 key 都有相对应的 value 给定一个特定的 key，map 可以快速寻找到相关联的 value

map 的操作和实现:

map.put(key, val) ——加入一个映射 key→val

map.get(key) ——获得这个 key 对应的 value

map.containsKey(key) ——测试是否这个映射有一个 key

map.remove(key)——删除一个映射

我们选择的 map 的实现是 HashMap

测试导向开发(TDD)和 corner cases:

在 TDD 中, 先写测试代码, 再写实现代码:

1. 如何得到创建测试用例所需的输入和预期输出?
2. 通过阅读问题描述和 Javadoc 的规格, 通过查看方法签名——他们通常足够创造测试用例

总是要为 corner cases 极端案例创造测试用例: 例如: 空列表, 单例列表 (仅包含一个元素的列表), 答案最小的列表等于零/非零或假/真

Question 9

Incorrect

Mark 0.00 out of 10.00

Rewrite the variable declaration below using Lists instead of arrays:

1. char[][] matrix;

Answer: List<List<String>> matrix=new ArrayList<List<String>>();



The correct answer is: List<List<Character>> matrix ;

Question 13

Incorrect Mark 0.00 out of 10.00

Flag question

Given a code:

```
List<Integer> list1 = new ArrayList<>();
list1.add(100);
list1.add(200);
final List<Integer> list2 = list1;
list1.add(300);
```

If we add a line of code below:

```
list2.set(3, 400);
```

choose the **correct** statement:

Select one:

a. there will be an error, detected by static checking.

b. there will be an error, detected by dynamic checking.

c. there is no error.

截图(Alt + A)

Your answer is incorrect.

The correct answer is: there will be an error, detected by dynamic checking.

Lecture 3:

Coding rules:

Don't Repeat Yourself:

重复的代码会危害安全：如果您在两个地方都有相同或非常相似的代码，则根本的风险是两个副本中都存在错误，并且某些维护人员会在一处修复该错误，而另一处不会。

复制粘贴是一种极具诱惑力的编程工具，每次使用时都应小心！——复制的板块越长，风险越大。

例子：如果短时间内重复使用同一个变量，当要更改变量名时，工程量会很大；解决办法：使用数组或者 list 来存储，使用循环来增加和返回值

Comments Where Needed:

Specification：良好的注释应该使代码更易于理解，更安全（因为已记录了重要的假设），并为更改做好了准备。一种重要的注释是 Specification (spec)，它出现在方法上方或类上方，并记录该方法或类的行为：在 Java 中，它通常以 Javadoc 注释的形式编写，这意味着它以/**开头，并包含@-syntax，例如方法的@param 和@return。

Document Source：另一个至关重要的评论是指定从其他地方复制或改编的一段代码的来源。这对练习软件开发人员非常重要，例如，当您修改在 Web 上找到的代码时。记录来源的原因之一是避免侵犯版权——Stack Overflow 上的一小段代码片段通常是在公共领域中，但是从其他来源复制的代码可能是专有的，也可能被其他种类的开放源代码许可所涵盖，但限制性更强；记录源代码的另一个原因是该代码可能会过时。自从首次被回答以来，这些代码所产生的 Stack Overflow 答案就已经有了很大的发展

Bad Comments: 有些评论是不好的，没有必要 例如，将代码直接音译成英语并不能改善理解，因为您应该假设您的读者至少知道 Java

Fail Fast:

Fail Fast 意味着代码应尽早发现其错误： 越早发现问题（越接近问题的起因），查找和修复问题就越容易 静态检查比动态检查失败快，而动态检查比根本没有检查快，这可能会产生错误的答案，从而可能破坏后续的计算

Avoid Magic Numbers:

实际上，计算机科学家认为只有两个常数是有效的： 0、1，或者 2 也算

其他数字被称为是 magic 的，因为他们看起来像是凭空冒出来，没有任何解释： 一种解释数字的方法是 comment，但是一种更好的方法是将数字声明为一个有名字的常量，其名字需要优秀，清晰

One Purpose for Each Variable:

不要重用参数，也不要重用变量： 编程中，变量并不是稀缺资源 自由的介绍他们，赋予他们好名字，不需要他们的时候就停止使用他们 如果一个某种意思的变量，突然被用于另一种意思，读者会感到困惑。

Don't Use Global Variables:

全局变量是，意义可以改变的变量；

对于程序中的任何地方，都可以使用，都可以改变；

Java 中，全局变量通常是声明为 public static，public 使它在任何地方都能使用，static 意味着该变量只有一个实例

总的来说，需要把全局变量，改成传入参数和返回值。

系统测试：

意味着我们以原则性的方式选择测试用例，目的是设计具有三个理想属性的测试套件：

Correct: 一个正确的测试套件是规范的合法客户，它毫无怨言地接受规范的所有合法实现——这使我们可以自由地更改模块在内部的实现方式，而不必更改测试套件

Thorough: 一个完整的测试套件会发现实现中的实际错误，这些错误是程序员可能犯的错误造成的

Small: 一个小的测试套件，只有很少的测试用例，一开始编写起来更快，而且如果规范不断发展，更新起来也更容易——运行的也快

Test-First Programming:

1. Spec: 为函数写一个 specification
2. Test: 写为这个 specification 准备的测试
3. Implement: 写实际代码

specification 描述了函数的 input 和 output 行为：

给出了参数的种类和其他的额外限制；

给出了返回值的种类，描述了返回值和输入的关联；

包括 method signature 和 comment above it that describes what it does

先写测试是一个理解 specification 的好方法

Partitioning:

我们将输入空间分成子域，每个包含了一组输入

所有的子域放在一起形成了一个 partition：完全覆盖输入空间的不相交集的集合，因此每个输入都恰好位于一个子域中

于是，我们在每一个子域中都选一个测试例子。

Question 6

Correct Mark 10.00 out of 10.00 Flag question

We should not use global variables.

Making a variable into a constant can eliminate the risk of global variables.

What keyword should be added to such global variables to make them constants ?

Answer: final ✓

The correct answer is: final

Question 13

Correct Mark 10.00 out of 10.00 Flag question

In solving a problem recursively, you can define a/an helper method ✓ that uses an arbitrary number of parameters.

Lecture 4:

Testing:

Black-box and White-box Testing:

黑盒测试意味着只从 specification 中选择测试例子，而不是在方法实现中选择； 我们进行了分区，然后寻找边界值，而并非看实际代码

白盒测试(又叫玻璃盒测试)意味着带着该方法会如何实际实现的知识，来选择测试例子； 例如，如果代码实现，对于不同的输入，选择不同的算法，那么你就可以根据这些分区来选择例子； 另一个例子，如果你知道该实现有一个内在的缓存，可以记住先前的输入的答案，那么你应当进行重复输入的测试。

在进行白盒测试时，必须注意测试用例不需要规范未明确要求的特定实现行为。 例如，如果 spec 说“当输入的格式不对时，抛出一个异常”，那么你的测试不应该特地检查 空指针 异常，因为这就是当前的实现所做的事情。 这种情况下的规范允许抛出任何异常，因此您的测试用例同样应该具有通用性，以保持实现者的自由

判断测试套件的一种方法是询问测试套件执行程序的程度 被称作 coverage

有三种常见的 coverage：

Statement coverage：是否每个 statement 都通过一些测试例运行过了？

Branch coverage：对于程序中每一个 if 或者 while，是否都设计了 true 和 false 的测试例？

Path coverage：是否每一个可能的 branch 的连接，都有相应的测试例？

branch coverage 需要更多的测试，比 statement coverage 更加强，而 path coverage 比 branch coverage 更加强。

行业中，100% 的 statement coverage 是很常见的目标，但是也很难实现，因为一些无法到达的防御性代码。

但是，100% 的 Path coverage 是完全不可行的，需要指数大小的测试套件才能实现

一个标准的测试方法是，添加测试，直到测试套件到达足够的 statement coverage，也就是程序中每一个可到达的 statement 都至少被一个测试例执行过。

实际上，statement coverage 经常通过一个 code coverage tool 来计算，它计算每个 statement 被测试套件运行的次数。通过这个工具，白盒测试会很简单；所以只需要计算黑盒测试的覆盖率，并且增加更多测试例，直到所有的重要 statement 都被执行过。

Unit Testing and Integration Testing：

经过严格测试的程序，对每一个单独的 module 都会有测试；

测试单独的 module 的测试，被称作 unit test

单独测试 module 会让 debug 更加的简单

相对的就是 integration test，会测试一系列 modules，甚至是整个程序；

如果只有 integration test，那么当一个 test 失败后，不得不狩猎 bug——可能在程序的任何地方

Integration test 依然很重要，因为程序可能会在 modules 的连接处出现 bug——例如，一个 module 可能会期望不同于从别的 module 那里得到的输入；

但是如果你有一个彻底的 unit test 的集合，给了你对于单独 module 的自信，那么寻找 bug 的过程会简单很多。

Unit Testing and Stub:

可以通过写一个 stub 版本的 module，将一个高级的 module 分离出来。

一个 class 的 stub 又被称为 mock object

Stub 在构建大型系统的时候，是很重要的技术

Automated Testing:

意味着自动运行测试，并且检查结果； 测试驱动程序不应是提示您输入信息并输出结果以供您手动检查的交互式程序。 相反，测试驱动程序应在固定的测试用例上调用模块本身，并自动检查结果是否正确 测试驱动程序的结果应为“所有测试都正常”或“这些测试失败：……” 良好的测试框架（例如 JUnit）可帮助您构建自动化测试套件

Regression Testing:

一旦进行自动化测试，在修改代码后重新运行测试是很重要的；

这避免了 regressing —— 修复新 bug 或者添加新特征的时候引入新的 bug

每次改变后运行测试被称作 regression testing

每当您发现并修复错误时，输入引发该错误的输入并将其作为测试用例添加到您的自动化测试套件中，这被称为 regression testing

这个想法也引出了 test-first debugging： 当出现一个 bug 时，立刻写一个针对它的测试例，加入测试套件； 一旦发现并且解决了 bug，所有的测试例都会 pass，debug 就结束了。

实际上，这两个 idea，automated testing and regression testing，经常被用在连接的 test 上；

Regression testing 只有在测试可以被经常自动运行的时候，才是实际的；

相反的，如果已经有了自动测试，也会选择用 regression testing

所以 automated regression testing 是最好的实践

Question 5

Partially correct Mark 13.33 out of 20.00

Flag question

Which of these techniques are useful for choosing test cases in test-first programming, **before** any code is written?

Select one or more:

- Partitioning ✓
- Boundaries ✓
- Black box
- Regression
- Coverage
- White box
- Integration

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: Partitioning, Boundaries, Black box

Question 7

Correct Mark 10.00 out of 10.00

Flag question

As a temporary substitute for a method that is not yet to be developed, you write a code to simulate the method's functionality.
The method can then be called by another method that you want to test.

Such method is called a/an stub ✓.

Question 8

Incorrect Mark 0.00 out of 10.00

Flag question

Which button to click to get the Java Visualizer run the next line of your code and show the subsequent visualization?

Select one:

- a. Step Into
- b. Step Over
- c. Step Out
- d. Step Off
- e. Step On ✗

Your answer is incorrect.

The correct answer is: Step Into

Question 9

Incorrect Mark 0.00 out of 10.00

Flag question

Write one line of Java code that *declares* a MyList pointer named **p** and *initializes* it to the current MyList object.

Do not forget to end it with a semicolon.

Answer: MyList p = listA; ✗

The correct answer is: MyList p = this;

Lecture 5

规范对于方法的实现者来说是很好的，因为它们给了实现者在不告诉客户端的情况下更改实现的自由

Specification Structure:

Precondition: require

Postcondition: effect

precondition 是客户机（即方法的调用方）上的一个义务——它是调用方法的状态的一个条件（也就是不满足这个条件就无法调用）

postcondition 则是方法实现者的一个义务

如果调用状态的 Precondition 成立，则该方法必须遵守 Postcondition，通过返回适当的值、抛出指定的异常、修改或不修改对象等等

总体结构是一种逻辑含义：**如果在调用方法时 Precondition 成立，那么在方法完成时 Postcondition 必须成立；**

如果在调用方法时 precondition 不成立，则实现不受 postcondition 的约束；它可以自由地做任何事情，包括不终止、抛出异常、返回任意结果、进行任意修改等等

Java 对文档注释有一个约定，其中参数由@param 子句描述，结果由@return 和@throws 子句描述

Testing and Specifications:

即使是玻璃盒测试也必须遵循规范；

您的实现可能提供比规范所要求的更强大的保证，或者在规范未定义的情况下，它可能具有特定的行为，**但是您的测试用例不应该依赖于这种行为，测试用例必须遵守合同，就像其他客户一样。**

- For example, suppose you are testing this specification of `find`, which is slightly different from the one we've used so far:

```
static int find(int[] arr, int val)
  requires: val occurs in arr
  effects:  returns index i such that arr[i] == val
```

Testing and Specifications (2)

- This spec has a strong precondition in the sense that `val` is required to be found; and it has a fairly weak postcondition in the sense that if `val` appears more than once in the array, this specification **says nothing about which particular index of `val` is returned**
- Even if you implemented `find` so that it always returns the lowest index, your test case **can't assume** that specific behavior:

```
int[] array = new int[] { 7, 7, 7 };
assertEquals(0, find(array, 7)); // bad test case: violates the spec
assertEquals(7, array[find(array, 7)]); // correct
```

这个例子就是，规范没写如果出现多个 index，应当返回哪一个，所以专门测试哪一个 index 都是不对的，应当测试 index 对应的 value（也就是哪个 index 都行）

Specifications for Mutating Methods:

我们之前讨论过可变对象和不可变对象，但是我们的 `find` 规范没有给我们机会来说明如何在 postcondition 中描述副作用（对可变数据的更改）

```
static boolean addAll(List<T> list1, List<T> list2)
  requires: list1 != list2
  effects: modifies list1 by adding the elements of list2 to the end of it,
           and returns true if list1 changed as a result of call
```

这里如果 precondition 里不强调 `list1 != list2`，会出现无限给同一个 list 加入元素，无限发现新元素的循环（相当于 `list2` 跟着 `list1` 一起变）

正如我们说过，除非另有说明，否则 `null` 是隐式不允许的，我们也将使用另一个约定——除非另有说明，否则不允许改变可变参数。

Invariants:

Invariants 是在代码执行期间保证为真的条件

就是一些约定成俗的东西，例如 SLList 中，第一个点一定是 sentinel.next

Invariants 让代码更好理解，可以默认都为真

Generics:

为了让 list 支持所有类型的变量，常常用 T 和 E 定义。

Consider the two methods to find the value `val` in an integer array `a` below.

```
1. static int findFirst(int[] a, int val) {  
2.     for (int i = 0; i < a.length; i++) {  
3.         if (a[i] == val) return i;  
4.     }  
5.     return a.length;  
6. }
```

```
1. static int findLast(int[] a, int val) {  
2.     for (int i = a.length - 1; i >= 0; i--) {  
3.         if (a[i] == val) return i;  
4.     }  
5.     return -1;  
6. }
```

If clients only care that the find method should return any index `i` such that `a[i] == val`, if `val` is in `a`; and any integer `j` where `j` is **not** a valid index of array `a`, otherwise; then `findFirst` and `findLast` are behaviorally equivalent.

Select one:

- True
 False ✕

The correct answer is 'True'.

Lecture 6

Null References:

在 Java 中，对对象和数组的引用也可以采用特殊的 null 值，这意味着引用不指向对象

Primitives 不能为 null，编译器将通过静态错误拒绝此类尝试；

但我们可以将 null 分配给任何 non-primitive 变量，编译器会接受；但是会得到运行时的错误，因为无法调用任何方法。

Null 和空字符串 "" 或者空数组不相同：后者可以调用函数；后者长度为 0 但前者不存在长度，

调用 `.length()` 会抛出 `NullPointerException`

还要注意，非原语数组和集合（如 List）可能为非 null，但可以包含 null 作为值（同样会出错）

Null Values:

空值是麻烦和不安全的，事实上，在大多数优秀的 Java 编程中，空值在参数和返回值中都

是隐式不允许的；

每个方法都隐式地对其对象和数组参数有一个非 null 的前提条件；每个隐式返回对象或数组的方法都有一个后置条件，即其返回值为非 null；如果一个方法允许一个参数为 null 值，它应该显式地声明它，或者如果它可能返回一个 null 值作为结果，它应该显式地声明它

There are extensions to Java that allow you to forbid null directly in the type declaration:

```
static boolean addAll(@NonNull List<T> list1, @NonNull List<T> list2)
```

where it can be checked automatically at compile time or runtime

Exception:

当列表索引 list.get (i) 超出列表的有效范围时引发 IndexOutOfBoundsException
NullPointerException，在尝试调用空对象引用上的方法时引发。
ArithmaticException，为整数除零等算术错误引发；
NumberFormatException，如果传入无法解析为整数的字符串，则由 Integer.parseInt 等方法引发

处理特殊结果的一种常见方法是返回特殊值：

Java 库中的查找操作通常是这样设计的：当需要正整数时，索引为 -1；当需要对象时，索引为空

问题：检查返回值很麻烦；容易忘记

Here's a better approach: the method *throws an exception*:

```
LocalDate lookup(String name) throws NotFoundException {  
    ...  
    if ( ...not found... )  
        throw new NotFoundException();  
    ...
```

The caller *handles* the exception with a *catch clause*, for example:

```
BirthdayBook birthdays = ...  
try {  
    LocalDate birthdate = birthdays.lookup("Alyssa");  
    // we know Alyssa's birthday  
} catch (NotFoundException nfe) {  
    // her birthday was not in the birthday book  
}
```

Checked and Unchecked Exceptions:

使用 checked 的异常来表示特殊结果，使用 unchecked 的异常来表示错误

Checked 异常之所以被称为 Checked 异常，是因为它们是由编译器检查的：

如果一个方法可能抛出一个选中的异常，那么必须在其签名中声明该可能性；

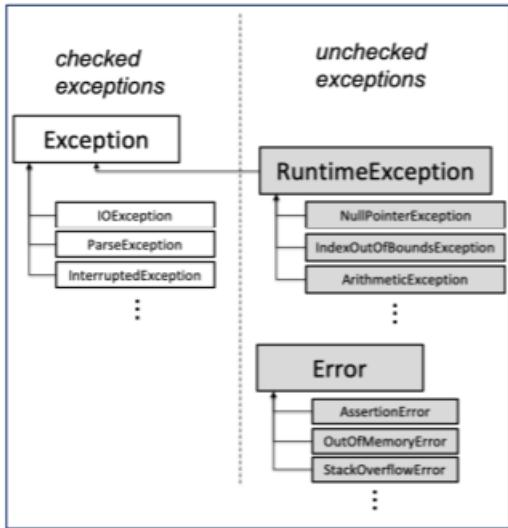
如果一个方法调用另一个可能抛出选中异常的方法，那么它必须要么处理它（使用 try catch），要么声明异常本身，因为如果没有在本地捕获它，它将被传播到调用方；如果没有处理，编译器会拒绝你的代码（static 报错）

相反，Unchecked 异常被用来表示 bug：

这些异常不应该由代码处理，除非可能是在 top level

我们不希望调用链上的每个方法都必须声明它，为了抛出在较低的调用级别上可能发生的所有与 bug 相关的异常：索引越界、空指针、非法参数、断言失败等等

因此，编译器不会检查（但是可以自己写，但是没必要）



Error 及其子类也是 Unchecked 异常

Error 应该被认为是不可恢复的， 并且不应该被代码捕获

自建 Exception 需要继承 Exception 或者 RuntimeException， 不能继承 Error or Throwable

Catch 的时候要尽量缩小范围

抛出异常来表示特殊值的代价很大， 因为客户机使用的时候甚至也要声明 exception；

所以如果阻止客户犯错很容易， 可以用 Unchecked 异常； 如果不容易， 也就是很容易犯错，

还是老老实实用 checked 异常

Question 2

Incorrect

Mark 0.00 out of 10.00

Flag question

Given the following code :

```
1. public static String nope() {
2.     return null;           // (1)
3. }
4.
5. public static void main(String[] args) {
6.     String a = nope();    // (2)
7.     String b = null;     // (3)
8.     if (a.length() > 0) { // (4)
9.         b = a;            // (5)
10.    }
11.    return b;           // (6)
12. }
```

Which line contains a static error ?

Select one:

- (1)
- (2) ✖
- (3)
- (4)
- (5)
- (6)

Your answer is incorrect.

The correct answer is: (6)

Question 7

Incorrect

Mark 0.00 out of 20.00

Flag question

Consider this code below for analyzing some `Thing` objects:

```
1. static List<Thing> allTheThings;
2.
3. static void analyzeEverything() {
4.     analyzeThings();
5. }
6.
7. static void analyzeThings() {
8.     try {
9.         for (Thing t : allTheThings) {
10.             analyzeOneThing(t);
11.         }
12.     } catch (AnalysisException ae) {
13.         return;
14.     }
15. }
16.
17. static void analyzeOneThing(Thing t) throws AnalysisException {
18.     // ...
19.     // ... maybe go past the end of a list
20.     // ...
21. }
```

Note that `IndexOutOfBoundsException`, `NullPointerException`, and `OutOfMemoryError` are unchecked exceptions;

and `AnalysisException` is a checked exception.

Which exception could be thrown by a call to `analyzeEverything`?

Select one or more:

- `AnalysisException` ✗
- `IndexOutOfBoundsException`
- `NullPointerException` ✓
- `OutOfMemoryError`

Your answer is incorrect.

The correct answers are: `IndexOutOfBoundsException`, `NullPointerException`, `OutOfMemoryError`

Question 9

Correct

Mark 10.00 out of 10.00

Flag question

Write one line of Java code that throws an `IllegalArgumentException` object with a message "n must not be even" to complete the if statement below :

```
1. if (n % 2 == 0) {
2.     // your code here
3.
4. }
```

Do not forget to end it with a semicolon.

Answer: `throw new IllegalArgumentException("n must not be even");`



The correct answer is: `throw new IllegalArgumentException("n must not be even");`

Question 10**Correct**

Mark 10.00 out of 10.00

Flag question

When we throw an `IllegalArgumentException` object within a method, that method must advertise it in the method signature.

Select one:

- True
- False ✓

The correct answer is 'False'.

Lecture 8:**Deterministic vs Underdetermined**

Deterministic 意味着当一个 state 满足先置条件的时候，结果已经完全被确定了

只有一个返回值和一个 final 状态是可能的；不存在合法输入使得有超过一个合法输出。

Underdetermined 而非 not deterministic，它对于同一个输入，允许有复数个合法输出。

Declarative vs Operational

Operational 会给出一系列步骤要求该方法执行；

Declarative 不会给出中间步骤的细节（只会给出最终结果的类型，以及它与初始状态的关系）

总体而言，更倾向于使用 declarative

最重要的是，它们不会无意中暴露客户机可能依赖的实现细节（然后在更改实现时发现不再有效）

别用 spec 来解释实现！会陷入 Operational

Stronger vs Weaker

称 S2 比 S1 Stronger 或者 equal 的前提：

1. S2 的前置条件比 S1 weaker 或者 equal
2. S2 的后置条件比满足 S1 的前置条件的后置条件 Stronger 或者 equal

此时，满足 S2 的实现可以满足 S1，于是可以用 S2 替换 S1

如果 1, 2 只能满足其一，那么称这两个 spec 是 incomparable 的

Diagramming Specification

整个空间是所有可能的 java 方法

每个点都是一种方法实现

一个 spec 则定义了一个区域，里面是所有可能的实现；满足 pre/post-condition 的在里面，不满足的在外面；而只要在里面，实现者都可以任意更改代码细节；客户机则不清楚具体的实现

越强的 spec，图中就越小，代表其中可行的实现越少；如果 S1 比 S2 强，那么图中 S2 会把 S1 包裹在内，也就是满足 S1 的实现都满足 S2.

Specification should be Coherent:

不应该有那么多 case，尽量做一件事情

Result of Specification should be Informative:

如果函数可能返回 null，那就不能把特殊返回值设成 null

必须足够 strong 以满足客户机的基本需求

但是不能太强！强的以至于无法保证能满足需求

规范应该尽可能使用抽象类型

另一个设计问题是是否使用一个前提条件，如果是的话，方法代码是否应该在继续之前尝试确保满足这个前提条件

事实上，前提条件最常用的用法就是要求一个属性，因为方法想要检查它很难或很昂贵

如上所述，一个重要的前提条件会给客户机带来不便，因为他们必须确保没有在错误状态下

(违反前提条件) 调用方法；如果他们这样做了，就没有可预测的方法从错误中恢复

这就是为什么 javaapi 类倾向于指定（作为后置条件）在参数不合适时抛出未检查的异常 (**也就是给用户犯错的空间，由程序来抛出异常来解决错误**)

应当 fail fast，避免 fail 大规模扩散

有时，检查一个条件而不使一个方法慢得令人无法接受是不可行的，在这种情况下，一个前提条件通常是必要的

如果我们想使用二分搜索实现 find 方法，我们就必须要求对数组进行排序
强制该方法实际检查数组是否已排序将破坏二分搜索的全部目的：在对数时间而不是线性时
间内获得结果

Access Control

Public 意味着向程序的其他所有部分都公开；但是如果把所有函数都公开了，用户会依赖上
你的内部函数，会导致很难更改

Static vs Instance Methods

Static 函数不和类中任何实际的实例有关联，而 Instance 必须被一个实际的 object 调用；
Static 可能要多传一些参数，也就是 Instance 中的 this

link-based list 的 get () 函数很慢

Question 4

Incorrect

Mark 0.00 out of 10.00

Given this specification:

1. static String join(String delimiter, String[] elements)
2. effects: append together the strings in elements, but at each step,
if there are more elements left, insert delimiter
- 3.

Rewrite the spec so it is declarative, **not** operational.

Select one:

- a. 1. effects: returns elements joined together with copies of delimiter, i.e.
2. elements[0] + delimiter + elements[1] + delimiter +
3. ... + delimiter + elements[elements.length-1]
- b. 1. effects: returns the result of adding all elements to a
2. new StringJoiner(delimiter)
- c. 1. effects: returns the result of looping through elements and
2. alternately appending an element and the delimiter
- d. 1. effects: returns the result of recursive calls on the elements and
2. while concatenating the delimiter

Your answer is incorrect.

The correct answer is:

1. effects: returns elements joined together with copies of delimiter, i.e.
2. elements[0] + delimiter + elements[1] + delimiter +
3. ... + delimiter + elements[elements.length-1]

You want to use a generic array using casting in your implementation of a data structure.
For example, you write the following line in your constructor or your method:

```
1. T[] elements = (T[]) new Object[numOfElements];
```

Write the annotation that you need to write before the constructor or the method:

Answer: `@SuppressWarnings("unchecked")`



The correct answer is: `@SuppressWarnings("unchecked")`

Lecture 9

Avoiding Debugging

设计上让 bug 不可能！

将 bug 缩小成程序的一部分

Make Bugs Impossible:

Static checking/ dynamic checking

by Immutability

immutable 类型是指其值一旦创建就永远不会更改的类型 (final 只让 reference 不可变，实际指向的内容也许可变)

Assertions

检查前提条件是防御性编程的一个例子，防御性编程是减轻 bug 影响的一个方法

通常的做法是为这类防御检查定义一个过程，通常称为 assert：

```
assert (x >= 0);
```

断言有一个好处，那就是记录关于程序当时状态的假设，意味着，在这一行代码的时间点，应当 xxx。

assert 语句还可以包括描述表达式，该表达式通常是字符串，但也可以是基元类型或对对象的引用——当断言失败时，描述将打印在错误消息中，以便可以使用它向程序员提供有关失

败原因的其他详细信息： assert ($x \geq 0$) : "x is " + x;

问题在于，java 默认是不跑 assertion 的，因为会花费大量时间。测试的时候愿意跑，是因为方便 debug，用户使用的时候则不现实了

该 assert 的东西：参数的需求，返回值的需求

```
switch (vowel) {  
    case 'a':  
    case 'e':  
    case 'i':  
    case 'o':  
    case 'u': return "A";  
    default: Assert.fail();  
}
```

这是一个覆盖所有情况的例子

内部的问题可用 Assertion 来解决（是 bug），外部的问题（非 bug）可以用 exception 来处理

因为 assertion 有可能被不可用，因此程序的正确性不能由 assertion 的结果来控制

Incremental Development：很好的定位 bug 到程序的微小部位的方法；

Modularity：将系统分割成组件或者模块（反义词 monolithic）

Encapsulation

意味着给模块造墙，因此模块只对模块内负责，出了 bug 也不会影响外界

access control：用 public 和 private 来控制变量和方法的可见性和可访问性

variable scope：

变量的作用域是程序文本中定义该变量的部分，即表达式和语句可以引用该变量

方法参数的作用域是方法的主体

局部变量的作用域（在 Java 中）从其声明扩展到声明周围块的右大括号

保持变量作用域尽可能小，可以更容易地推断程序中可能存在的 bug

始终在 for 循环初始值设定项中声明循环变量

仅当您第一次需要时才声明变量，并且可以在最里面的花括号块中声明

避免全局变量

Mutability:

Mutability 和 immutability 的区别在这个 object 只有一个 reference 的时候不那么大

但是如果还有一个变量指向它， Mutability (例如 String) A 改变不会改变 B, immutability

(例如 StringBuilder) A 改变会改变 B

StringBuilder 能够节省时间， 提高性能

Mutability 的风险：

传输可变的值——例如函数中改变参数，却改变了本来的东西；

返回可变的值——函数调用处修改了，函数也会变

defensive copying 意味着函数调用处可以任意修改返回值，而不影响函数；

Immutability 效率更高，减少 copying 也就减少了需要的存储空间

Aliasing:

同一个可变对象拥有两个可变的 reference——这就是使得可变类型变得有风险的东西

因此，如果函数里要修改可变变量，需要在 spec 中声明；

Iterator:

逐步遍历元素集合并逐个返回元素的对象

next() 返回下一个值（是个 mutator 方法，不止是返回下一个元素，还推进 iterator 到下一个值）

hasNext() 测试是否到达了集合的尽头

Immutable Types: primitive types, primitive wrappers, BigInteger, BigDecimal

Mutable: Java's collections types (List, Set, Map)

Collections.unmodifiableList , Collections.unmodifiableSet, Collections.unmodifiableMap 是一种另类的 wrapper

Question 3

Incorrect

Mark 0.00 out of 10.00

Flag question

Consider this (incomplete) method:

```
1.  /**
2.   * Solves quadratic equation ax^2 + bx + c = 0.
3.   *
4.   * @param a quadratic coefficient, requires a != 0
5.   * @param b linear coefficient
6.   * @param c constant term
7.   * @return a list of the real roots of the equation
8.   */
9. public static List<Double> quadraticRoots(final int a, final int b, final int c) {
10.    List<Double> roots = new ArrayList<Double>();
11.    // A
12.    ... // compute roots
13.    // B
14.    return roots;
15. }
```

What assertion would be reasonable to write at position **A** (*before* computing the roots) ?

Select one:

- a. assert a != 0;
- b. assert b != 0;
- c. assert c != 0;
- d. assert roots.size() >= 0; ✖
- e. assert roots.size() <= 2;

Your answer is incorrect.

The correct answer is: assert a != 0;

Question 4

Incorrect

Mark 0.00 out of 10.00

Flag question

Consider this (incomplete) method:

```
1.  /**
2.   * Solves quadratic equation ax^2 + bx + c = 0.
3.   *
4.   * @param a quadratic coefficient, requires a != 0
5.   * @param b linear coefficient
6.   * @param c constant term
7.   * @return a list of the real roots of the equation
8.   */
9. public static List<Double> quadraticRoots(final int a, final int b, final int c) {
10.    List<Double> roots = new ArrayList<Double>();
11.    // A
12.    ... // compute roots
13.    // B
14.    return roots;
15. }
```

What assertion would be reasonable to write at position **B** (*after* computing the roots) ?

Select one:

- a. assert a != 0;
- b. assert b != 0;
- c. assert c != 0;
- d. assert roots.size() >= 0;
- e. assert roots.size() <= 2;

Your answer is incorrect.

The correct answer is: assert roots.size() <= 2;

Question 6

Incorrect

Mark 0.00 out of 10.00

Flag question

Consider the following code, which is *missing* some variable declarations :

```
1. class Apartment {
2.
3.     Apartment(String newAddress) {
4.         this.address = newAddress;
5.         this.roommates = new HashSet<Person>();
6.
7.
8.         String getAddress() {
9.             return address;
10.        }
11.
12.        void addRoommate(Person newRoommate) {
13.            roommates.add(newRoommate);
14.            if (roommates.size() > MAXIMUM_OCCUPANCY) {
15.                roommates.remove(newRoommate);
16.                throw new TooManyPeopleException();
17.            }
18.        }
19.
20.        int getMaximumOccupancy() {
21.            return MAXIMUM_OCCUPANCY;
22.        }
23.    }
```

Which one is the best declaration for the MAXIMUM_OCCUPANCY variable?

Select one:

- a. static final int MAXIMUM_OCCUPANCY = 8;
- b. final int MAXIMUM_OCCUPANCY = 8; ×
- c. static int MAXIMUM_OCCUPANCY = 8;
- d. int MAXIMUM_OCCUPANCY = 8;
- e. public int MAXIMUM_OCCUPANCY = 8;
- f. public static int MAXIMUM_OCCUPANCY = 8;

Your answer is incorrect.

The correct answer is: static final int MAXIMUM_OCCUPANCY = 8;

static和final的意义是不同的，

static修饰的时候代表对象是静态的，而final修饰的时候代表对象只能赋值一次，
他们连用的时候是因为定义的那个对象既要它是静态的，也要求它的值不能再被修改。

举例说明：

```
static int a=1;
static final b=1;
```

这里a和b的区别在于，a在程序里可以被重新赋值为2或3或等等的整数，而b在程序里不能被重新赋值，b永远都为1，也就是说b是一个常量。

```
final int c=1;
static final b=1;
```

这里c和b的区别在于，b存放在静态空间，不会在程序运行时被释放，它永远占着内存直到程序终止，而c在程序用完它而不会再用到它的时候就会被自动释放，不再占用内存。

当一个常数或字符串我们需要在程序里反复反复使用的时候，我们就可以把它定义为static final，这样内存就不用重复的申请和释放空间。

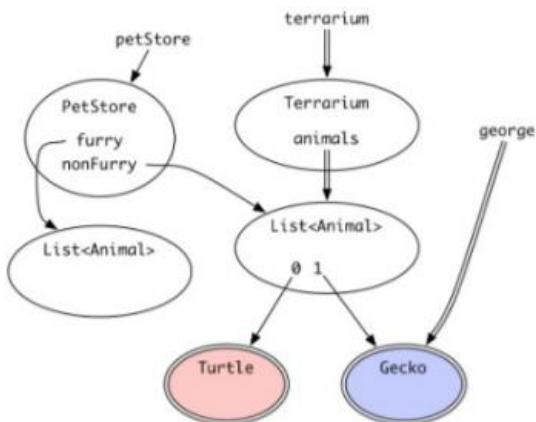
Question 7

Correct

Mark 10.00 out of 10.00

Flag question

Consider the following snapshot diagram:



Is it possible that a client with the variable **terrarium** could modify the **Turtle** in red?

Select one:

- a. No, because the "Turtle" is immutable ✓
- b. Yes, because all the references between "terrarium" and the "Turtle" are mutable
- c. Yes, because of some reference between "terrarium" and the "Turtle" that is mutable
- d. Yes, because the "Turtle" is mutable
- e. No, because of some reference between "terrarium" and the "Turtle" that is immutable
- f. No, because all the references between "terrarium" and the "Turtle" are immutable

Your answer is correct.

The correct answer is: No, because the "Turtle" is immutable

Question 10

Incorrect

Mark 0.00 out of 10.00

Flag question

Consider MyIterator's next method:

```
1. public class MyIterator {
2.
3.     private final ArrayList<String> list;
4.     private int index;
5.
6.     ...
7.
8.     /**
9.      * Get the next element of the list.
10.     * Requires: hasNext() returns true.
11.     * Modifies: this iterator to advance it to the element
12.     *           following the returned element.
13.     *           @return next element of the list
14.     */
15.    public String next() {
16.        final String element = list.get(index);
17.        index++;
18.        return element;
19.    }
20. }
```

What is the type of the input to next?

Select one:

- a. MyIterator
- b. void
- c. ArrayList
- d. String
- e. boolean
- f. int

Your answer is incorrect.

The correct answer is: MyIterator

Question 12

Incorrect

Mark 0.00 out of 10.00

Flag question

Consider MyIterator's next method:

```
1. public class MyIterator {
2.
3.     private final ArrayList<String> list;
4.     private int index;
5.
6.     ...
7.
8.     /**
9.      * Get the next element of the list.
10.     * Requires: hasNext() returns true.
11.     * Modifies: this iterator to advance it to the element
12.     *           following the returned element.
13.     *           @return next element of the list
14.     */
15.    public String next() {
16.        final String element = list.get(index);
17.        ++index;
18.        return element;
19.    }
20.}
```

next has the precondition requires: hasNext() returns true.

Which input to next is constrained by the precondition?

Select one:

- a. this
- b. list
- c. index ✖
- d. element
- e. hasNext

Your answer is incorrect.

The correct answer is: this

Question 15

Incorrect

Mark 0.00 out of 10.00

Flag question

Consider MyIterator's next method:

```
1. public class MyIterator {
2.
3.     private final ArrayList<String> list;
4.     private int index;
5.
6.     ...
7.
8.     /**
9.      * Get the next element of the list.
10.     * Requires: hasNext() returns true.
11.     * Modifies: this iterator to advance it to the element
12.     *           following the returned element.
13.     * @return next element of the list
14.     */
15.    public String next() {
16.        final String element = list.get(index);
17.        ++index;
18.        return element;
19.    }
20. }
```

Another part of the postcondition of next is **modifies: this iterator to advance it to the element following the returned element.**

What is constrained by that postcondition?

Select one:

- a. the return value
- b. this
- c. hasNext ✖
- d. list

Your answer is incorrect.

The correct answer is: this

Lecture 10

我们研究一个强大的概念，称为抽象数据（ADT）类型，它使我们能够将如何在程序中使用数据结构与数据结构本身的特定形式区分开来

Abstract data types:

Abstraction: 用更简单、更高级的想法省略或隐藏低级的细节

Modularity: 将系统划分为组件或模块，每个组件或模块都可以独立于系统的其余部分进行设计、实现、测试、推理和重用

Encapsulation: 在模块（硬壳或胶囊）周围建造墙壁，以便模块对其自身的内部行为负责，而系统其他部分的 bug 不会损害其完整性

Information hiding: 对系统的其余部分隐藏模块实现的详细信息，以便以后可以在不更改系统的其余部分的情况下更改这些详细信息

Separation of concerns: 使一个特性（或“关注点”）由单个模块负责，而不是将其分散到多个模块

Operations of Abstract Type:

Creators: 创建该种类的新对象；创建者可以将对象作为参数，但不能将其作为正在构造的类型的对象

Producers: 从该类型的旧对象创建新对象；例如，String 的 concat 方法是一个 producer：它接受两个字符串，并生成一个表示它们的串联的新字符串

Observers: 获取抽象类型的对象并返回不同类型的对象；例如，List 的 size 方法返回一个 int

Mutators: 改变对象；List 的 add 函数，通过在末尾添加元素来改变列表

▪ Integer.valueOf()	Creator
▪ BigInteger.mod()	Producer
▪ List.addAll()	Mutator
▪ String.toUpperCase()	Producer
▪ Set.contains()	Observer
▪ Map.keySet()	Observer
▪ Collections.unmodifiableList()	Producer
▪ BufferedReader.readLine()	Mutator

creator 操作通常作为构造函数实现，如 new ArrayList ()；但是创建者也可以只是一个静态方法，比如 Arrays.asList ()（被称作 factory method）

Mutators 通常是 void 的返回类型，但也不一定

ADT 例子：

int is Java's primitive integer type

int is immutable, so it has *no* mutators

- creators: the numeric literals 0, 1, 2, ...
- producers: arithmetic operators +, -, ×, ÷
- observers: comparison operators ==, !=, <, >
- mutators: none (it's immutable)

List is Java's list type

List is mutable

List is also an interface, which means that other classes provide the actual implementation of the data type

These classes include `ArrayList` and `LinkedList`

- creators: `ArrayList` and `LinkedList` constructors,
`Collections.singletonList`
- producers: `Collections.unmodifiableList`
- observers: `size`, `get`
- mutators: `add`, `remove`, `addAll`, `Collections.sort`

String is Java's string type

String is immutable

- creators: `String` constructors
- producers: `concat`, `substring`, `toUpperCase`
- observers: `length`, `charAt`
- mutators: none (it's immutable)

ADT 是由其操作决定的

类型的操作构成了它的抽象——这是公共部分，对使用该类型的客户端可见

实现类型的类的字段以及帮助实现复杂数据结构的相关类构成了一个特定的表示——此部分是私有的，仅对类型的实现者可见

设计 ADT:

It's better to have a few, simple operations

Each operation should have a well-defined purpose and a coherent behavior

The set of operations should be adequate

But it should not mix generic and domain-specific features

Next, it should be representation independent

最好有一些简单的操作

每个操作都应该有明确的目的和连贯的行为

这套操作应该足够

但它不应该混合通用和特定领域的特性

其次，它应该是独立于代表性的

Method Overloading

多个同名但参数不同的方法——Java 将确定将调用哪个方法

不好用：代码相同；不适用于将来的列表（如果创造一个新的 list，还要再写一遍）；很难维护，一个改各个要改

Interface

只要写会做什么，不用写怎么做；

```
public interface ListIF<T> {
    public void addLast(T item); ←
    public T getLast();
    public T delLast();
    public T get(int i); ←
    public int size(); ←
}
```

Implements

告知编译器他们是 interface 的实现

Method Overriding

如果子类中有一个方法的签名与超类中的签名完全相同，我们就说该子类重写了该方法

Method Overriding vs Overloading

前者：同一个签名

后者：同一个方法名，不同的签名

@Override Annotation——这个标记的唯一作用是，如果它不是一个重写方法，代码就不会编译

Interface Inheritance

Inheritance：子类从超类继承接口（而非实现），接口规定子类做什么但不规定怎么做（方法都要 override）

Instantiation

```
public static void main(String[] args) {
    ListIF<String> list1 = new SLLList<String>();
    list.addLast("a");
}
```

Implementation Inheritance

子类从超类继承签名和实现

在 interface 中，我们使用 default 关键词来规定一个方法，子类会继承它

Static Type vs Dynamic Type

编译时种类是 static type

运行时种类是 dynamic type

```
public static void main(String[] args) {  
    Animal animal;  
    animal = new Dog();  
}
```

Extend

如果一个类想继承另一个类，则要用 extend

如果 extends，会继承所有的内容（实例和静态变量，方法，嵌套类）

调用父类的函数，用 super

Constructor 里用 super ()；

Question 1

Incorrect

Mark 0.00 out of 10.00

Flag question

Consider an abstract data type Bool.

The type has the following operations:

- 1. true : Bool
- 2. false : Bool
- 3.
- 4. and : Bool × Bool → Bool
- 5. or : Bool × Bool → Bool
- 6. not : Bool → Bool

where the first two operations construct the two values of the type,
and last three operations have the usual meanings of logical *and*, logical *or*, and logical *not* on those
values.

The following are possible ways that Bool might be implemented and still be able to satisfy the specs of
the operations, except one.

Which one is **not** the correct way?

Select one:

- a. As a long value in which all possible values mean true.
- b. As a single bit, where 1 means true and 0 means false.
- c. As an int value where 2 means true and 5 means false. ×
- d. As a reference to a String object where "false" to mean true and "true" to mean false

Your answer is incorrect.

The correct answer is: As a long value in which all possible values mean true.

Question 14

Incorrect

Mark 0.00 out of 10.00

Flag question

Choose the correct statement.

Select one:

- a. If you are a subclass of an interface, you have to override all of its method signatures.
- b. If you override a method, you have to annotate the method with @Override ×
- c. Method overloading is when you have multiple methods with the same signature, but different names.
- d. An object o is instantiated with static type S and dynamic type D.
D is a subclass of S, and D overloads method m() of S.
At runtime, o.m() will call method m() that belongs to D.

Your answer is incorrect.

The correct answer is: If you are a subclass of an interface, you have to override all of its method signatures.

@Override是伪代码,表示重写。(当然不写@Override也可以), 不过写上有如下好处:

- 1、可以当注释用,方便阅读;
- 2、编译器可以给你验证@Override下面的方法名是否是你父类中所有的,如果没有则报错。例如,你如果没写@Override,而你下面的方法名又写错了,这时你的编译器是可以编译通过的,因为编译器以为这个方法是你的子类中自己增加的方法。

Question 11

Incorrect

Mark 0.00 out of 10.00

Flag question

Consider the following abstract data type.

```
1. /**
2. * Represents a family that lives in a household together.
3. * A family always has at least one person in it.
4. * Families are mutable.
5. */
6. class Family {
7.     // the people in the family, sorted from oldest to youngest, with no duplicates.
8.     public List<Person> people;
9.
10.    /**
11.     * @return a list containing all the members of the family, with no duplicates.
12.     */
13.    public List<Person> getMembers() {
14.        return people;
15.    }
16. }
```

Here is a client of this abstract data type:

```
1. void client3(Family f) {
2.     // get any person in the family
3.     Person anybody = f.getMembers().get(0);
4.     ...
5. }
```

Assume all this code works correctly (both `Family` and `client3`) and passes all its tests.

Now `Family`'s representation is changed from a `List` to `Set`, as shown:

```
1. /**
2. * Represents a family that lives in a household together.
3. * A family always has at least one person in it.
4. * Families are mutable.
5. */
6. class Family {
7.     // the people in the family
8.     public Set<Person> people;
9.
10.    /**
11.     * @return a list containing all the members of the family, with no duplicates.
12.     */
13.    public List<Person> getMembers() {
14.        return new ArrayList<>(people);
15.    }
16. }
```

Assume that `Family` compiles correctly after the change.

Which of the following statements are true about `client3` after `Family` is changed?

Select one:

- a. client3 is independent of Family's representation, so it keeps working correctly.
- b. client3 depends on Family's representation, and the dependency would be caught as a static error.
- c. client3 depends on Family's representation, and the dependency would be caught as a dynamic error. ✖
- d. client3 depends on Family's representation, and the dependency would not be caught but would produce a wrong answer at runtime.
- e. client3 depends on Family's representation, and the dependency would not be caught but would (luckily) still produce the same answer.

Your answer is incorrect.

The correct answer is: client3 is independent of Family's representation, so it keeps working correctly.

Lecture 11

Invariant

什么是好的 ADT, 好的 ADT 的最后一个, 也许是最重要的特性是 preserves its own invariants 保留了自己的不变量

invariants 是程序的一个属性, 它总是 true, 对于程序的每个可能的运行时状态,

Immutability 是我们已经遇到的一个关键 invariants: 一旦创建, 一个 immutable 对象在其整个生命周期中应该始终表示相同的值

说 ADT 保留它自己的不变量意味着 ADT 负责确保它自己的不变量保持不变 (与用户操作无关), 这会使编程更加简单

representation exposure

例如客户机直接调用类内的变量

类外的代码可以直接修改表达; 这不仅会威胁 invariant 也会威胁 representation independence (因为我们修改内部实现, 必然会影响直接访问内部的客户机)

解决方法:

private 和 public 关键字表示哪些字段和方法只能在类内访问, 哪些可以从类外访问

final 关键字还有助于确保在构造对象后不会重新分配此不可变类型的字段

然后再用一些 get 函数让客户机访问。

另一个问题是,

类泄露了一个可变对象的 reference, 而类的稳定性依赖于其上

解决方法:

Defensive copying: 做一个可变对象的 copy 防止 reference 外泄

可变类型通常有一个 copy constructor, 允许您创建一个复制现有实例值的新实例

还有一个问题

Constructor 也许会接受多个 instance 的构造请求, 而如果 constructor 中有可变对象, 会导致其存储该变量的 reference, 从而导致多个实例指向同一个 reference

解决方法:

Defensive copying

为啥不能写一个详尽的 sepc 来解决
只有例如 copy 成本过大的时候才可以，否则找问题和 debug 的消耗也是巨大的

Java 也提供了一个方法： **immutable wrappers**

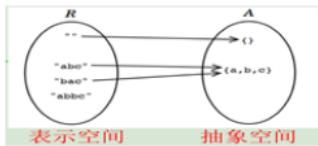
例如： Collections.unmodifiableList()

问题是， compile time 不会提醒 immutability， runtime 才会

Abstract Space vs Rep Space

Abstraction Function 抽象函数

表示空间R	抽象空间A
值的实际实现本质	抽象表示 (client看到和使用的值)
ADT实现者关注表示空间R	用户关注抽象空间A



R到A的映射

一定是满射：A中元素总有R中具体的实现

未必是单射：A中一个元素在R中可能有多重实现方式

未必是双射：R中表示不符合A中需求 (如图中"abbc")

抽象函数AF：R和A之间映射关系的函数

AF：R->A

对于RI : R-> Boolean

RI：某个具体的"表示"是否合法；表示值的一个子集，包含所有合法的表示值；一个条件，描述了什么是"合法"表示值。

从客户机的角度来看，抽象值空间由类型设计为支持的值组成——例如，无界整数的抽象类型（如 Java 的 BigInteger）将数学整数作为其抽象值空间

representation values 空间（简称 rep 值）由实际实现抽象值的 Java 对象组成——例如，BigInteger 的值可能是通过一个数字的数组来实现的，表达了一个原始 int 值，其中 rep space 就是所有这种数组的集合

在简单的情况下，抽象类型将被实现为单个 Java 对象，但更常见的情况是需要一个小型的对象网络——例如，List 的 rep 值可能是一个链表，一组对象通过下一个和上一个指针链接在一起；因此，rep 值不一定是单个对象，而是非常复杂的东西

实现者的工作是使用 rep 值空间实现 Abstract 空间的幻觉

每一个 abstract value 都与一个/一些 rep value 相映射；

实现抽象类型的目的是支持对抽象值的操作；我们需要能够创建和操作所有可能的抽象值，因此它们必须全部是 representable 可表示的

一些 abstract values 甚至和超过一个 rep value 相映射；
发生这种情况是因为 representation 不是严格编码；将无序字符集表示为字符串的方法不止一种

并非所有 rep value 都是映射的；
这将允许我们在命中特定字符的第一个实例时终止 remove 方法，因为我们知道最多只能有一个实例

Rep Invariant and Abstraction Function

abstraction function 映射了 rep value 到表示的 abstract value—— $AF : R \rightarrow A$
图中的弧表示函数术语中的抽象函数，我们上面讨论的属性可以表示为函数是满射的（也称为 on），不一定是内射的（也称为一对一），因此不一定是双射的；

rep invariant 映射了 rep value 到 Boolean—— $RI : R \rightarrow \text{Boolean}$
对于任意 rep value r ，当且仅当 r 是被 AF 映射了， $RI(r)$ 才是 true
换句话说， RI 告诉我们是否一个给定的 rep value 是结构合理的（在 AF 中被定义了）

Abstraction Function 和 **Rep Invariant** 的一个常见混淆是，它们是由 rep 和 abstract value space 的选择决定的，甚至是由 rep value space 单独决定的——如果是这样的话，他们就没什么用了，因为重复了，毫无意义
abstract value space 自己不能决定 AF 和 RI；同一个 abstract type 会有数个表达

关键的一点是，为 rep 定义一个类型，从而为 rep 值的空间选择值，并不能决定哪些 rep 值将被视为合法的，哪些是合法的，如何解释它们

说白了就是，该如何映射，映射谁，都是实现者自己规定的，而不是这两个 value space 规定的。

Checking the Rep Invariant

如果运行时确认了 rep invariant，可以帮助找 bug，帮助解决 rep exposure

No Null Values in the Rep

默认情况下，rep 不变量隐式包含 $x \neq \text{rep}$ 中具有对象类型（包括数组或列表中的引用）的每个引用 x 都为 null

rep exposure safety argument

这是一个注释，它检查 rep 的每个部分，查看处理该部分 rep 的代码（特别是关于来自客户端的参数和返回值，因为这是 rep 暴露的地方），并说明代码不暴露 rep 的原因

How to Establish Invariants

invariant 是一个对整个程序都适用的属性，对于一个对象的不变量，它会减少到该对象的整

个生命周期；

为了维持一个 invariant，需要

在物体的初始状态下使不变量为 true——creators 和 producers 必须为新的对象实例建立 invariant

确保对对象的所有更改都保持不变——mutators 和 observers 必须维持 invariant

- established by creators and producers;
- preserved by mutators, and observers; and
- no representation exposure occurs,

ADT invariants replace preconditions

ADT 可以帮助改善 precondition，因为很多对参数的描述都可以用一个新的 ADT 代替

Dynamic method selection (DMS)

指在运行时，如果具体实现重写了函数，则会运行这个实现的函数

X varName = new Y();

varName.m();

如果 X 定义了 m()，Y 重写了 m()，则发生 DMS

如果 X 没有定义 m()，那么哪怕 Y 定义了，也会出现 compile error!

Casting

- we call the method on two ShihTzus:

```
ShihTzu baobei = new ShihTzu("Baobei", 5);
ShihTzu jiaozi = new ShihTzu("Jiaozi", 7);

Dog largerDog = largerDog(baobei, jiaozi);
ShihTzu largerShihTzu = largerDog(baobei, jiaozi);
```

even though the run-time type will be
ShihTzu, this will cause a **compile
error**, because the static type of
largerDog is Dog, which is not
ShihTzu, nor a subclass of ShihTzu

ShihTzu largerShihTzu = (ShihTzu)largerDog(baobei, jiaozi);

不改变类型，运行时类型不会变，静态会变。

Object Class

每个类型都是 Object 类的子类

对象类是一个具体的类

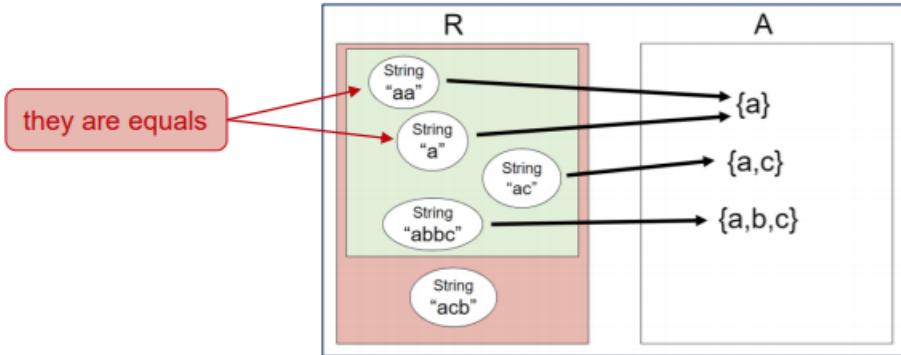
如何比较？

Implement Comparable<dog>

然后重写（实现） compareTo——return this.weight-other.weight

Three Ways to Regard Equality (between abstract values)

Using an abstraction function——回想一下，抽象函数 $AF: R \rightarrow A$ 将数据类型的具体实例映射到相应的抽象值；用 AF 作为等式的定义，我们可以说 a 等于 b 当且仅当 $AF(a) = AF(b)$ ；例如，当我们使用允许重复的字符串，同时要求对该字符串的字符进行排序，以表示字符集时



Using a relation——等价关系是 $E \subseteq T \times T$ 使得， $E(t,t) \forall t \in T$ (reflexive)， $E(t,u) \Rightarrow E(u,t)$ (symmetric)， $E(t,u) \wedge E(u,v) \Rightarrow E(t,v)$ (transitive)

这两个概念是等价的：

等价 relation 产生一个抽象函数（该 relation 将 T 划分，因此 f 将每个元素映射到它的划分类）

抽象函数所产生的关系是等价关系（请自己检查这三个属性是否成立）

Using observation——一个局外人（客户）对他们的观察；

我们可以说，当两个物体不能通过观察来区分时，它们是相等的——我们可以应用的每一个操作都会对这两个物体产生相同的结果；

考虑集合表达式{1,2}和{2,1}——使用集合、基数|…|和成员资格可用的观察者操作 \in ，这些表达式无法区分：

对于 ADT 来说，观察意味着调用函数

所以两个对象相等**当且仅当**它们不能通过调用抽象数据类型的任何操作来区分时

== operator 比较 reference

更精确，测试 referential equality；

指向同一个存储空间时，是==的

equals() 比较对象内容

换句话说，object equality；

对于每个不同的 ADT，都需要定义

equals() 的 default meaning 就是 ==，所以需要重写

instanceof 测试了是否一个对象是另一个实际类型的实例；

最好用于 static type checking

Disallow

Object Contract

equals 必须定义一个等价关系，即一个自反的、对称的和传递的关系

equals 必须一致：如果没有修改对象 equals 比较中使用的信息，则对方法的重复调用必须产生相同的结果

对于非空引用 x, x.equals (null) 应返回 false

hashCode 必须为 equals 方法认为相等的两个对象生成相同的结果

Equality of Mutable Types

Observational equality：意味着在程序的当前状态下，现在无法区分两个引用

Behavioral equality：意味着现在或将来都无法区分两个引用，即使调用一个 mutator 来更改一个对象而不是另一个对象的状态

应当用后者

Consider the following problematic datatype:

```
1.  /* Represents an immutable right triangle. */
2.  class RightTriangle {
3.      /*A*/
4.      private double[] sides;
5.
6.      // sides[0] and sides[1] are the two legs,
7.      // and sides[2] is the hypotenuse, so declare it to avoid having a
8.      // magic number in the code:
9.      /*B*/
10.     public static final int HYPOTENUSE = 2;
11.
12.     /** Make a right triangle.
13.      * @param legA, legB  the two legs of the triangle
14.      * @param hypotenuse  the hypotenuse of the triangle.
15.      * @throws IllegalArgumentException if
16.      *          * Requires hypotenuse^2 = legA^2 + legB^2
17.      *          * (within the error tolerance of double arithmetic)
18.      */
19.
20.     public RightTriangle(double legA, double legB, double hypotenuse) {
21.         /*D*/
22.         this.sides = new double[] { legA, legB, hypotenuse };
23.     }
24.
25.     /**
26.      * Get all the sides of the triangle.
27.      * @return three-element array with the triangle's side lengths
28.      */
29.     public double[] getAllSides() {
30.         return sides;
31.     }
32.
33.     /**
34.      * @return length of the triangle's hypotenuse
35.      */
36.     public double getHypotenuse() {
37.         return sides[HYPOTENUSE];
38.     }
39.
40.     /**
41.      * @param factor to multiply the sides by
42.      * @return a triangle made from this triangle by
43.      *         multiplies all side lengths by factor.
44.      */
45.     public RightTriangle scale(double factor) {
46.         return new RightTriangle (sides[0]*factor, sides[1]*factor, sides[2]*factor);
47.     }
48.
49. }
```

Which of the following statements are true?

Select one:

- a. The line marked /*A*/ is a problem for rep exposure because arrays are mutable.
- b. The line marked /*B*/ is a problem for representation independence because it reveals how the sides array is organized.
- c. The line marked *C* is a problem because creator operations should not have preconditions.
- d. The line marked /*D*/ is a problem because it puts legA, legB, and hypotenuse into the rep without doing a defensive copy first. x

Your answer is incorrect.

The correct answer is: The line marked /*B*/ is a problem for representation independence because it reveals how the sides array is organized.

Suppose Bag<E> is a mutable ADT representing what is often called a *multiset*, an unordered collection of objects where an object can occur more than once. It has the following operations:

```
1.  /** make an empty bag */
2.  public Bag<E>()
3.
4.  /** modify this bag by adding an occurrence of e, and return this bag */
5.  public Bag<E> add(E e)
6.
7.  /** modify this bag by removing an occurrence of e (if any), and return this bag */
8.  public Bag<E> remove(E e)
9.
10. /** return number of times e occurs in this bag */
11. public int count(E e)
```

Suppose we run this code:

```
1.  Bag<String> b1 = new Bag<>().add("a").add("b");
2.  Bag<String> b2 = new Bag<>().add("a").add("b");
3.  Bag<String> b3 = b1.remove("b");
4.  Bag<String> b4 = new Bag<>().add("b").add("a"); // swap!
```

If Bag is implemented with **behavioral** equality, which of the following expression is **true**?

Select one or more:

- i. b1.equals(b2)
- ii. b1.equals(b3) ✓
- iii. b1.equals(b4)
- iv. b2.equals(b3)
- v. b2.equals(b4) x
- vi. b3.equals(b1) ✓

Your answer is partially correct.

You have selected too many options.

The correct answers are: b1.equals(b3),
b3.equals(b1)

Lecture 12

Hash table

两个非常常见的集合实现 HashSet 和 HashMap 使用 **Hash table** 数据结构，并且依赖于 hashCode 方法来正确实现存储在集合中并用作映射中的键的对象

Hash table 是映射的一种表示形式：一种抽象数据类型，将键映射到值

Hash table 提供了固定时间的 lookup，因此它们的性能往往优于树或列表

除了提供 equals 和 hashCode 之外，key 不必被排序，也不必有任何特定的属性

它包含一个数组，该数组初始化大小为与，希望插入的元素数

当一个 key 和一个 value 被提供以供插入时，我们计算该键的 Hash code，并将其转换为数组范围内的 index（例如，通过模除）

然后在该索引处插入该值

Hash table 的 rep invariant 包含一个基本约束，即可以从哈希代码确定的槽开始查找密钥

Hash code 的设计使得 key 将均匀地分布在 index 上

但有时会发生冲突，两个 key 放在同一 index 中

因此，Hash table 不是在 index 中保存单个 value，而是实际保存一个 key/value 的 pair 的列表，通常称为 hash bucket

key/value pair 可以简单的通过有两个 field 的 object 实现

插入时，将一个 pair 添加到由 hash code 确定的数组槽中的列表中

对于查找，您 hash the key，找到正确的槽，然后检查每个 pair，直到找到一个 key 等于查询

Breaking Hash Tables

必须要 override

如果同一个对象的 hash code 不同，就会放在不同地方，查询就查不到。

default hashCode() 是返回存储的内存地址（但这只对 mutable 比较合适）

simple and drastic way——返回常数，灾难性表现，但是满足 object contract

standard way——使用 equality（为每一个部件调用 hashCode，再把他们都连接）（例如可以用 getLength()）

只要满足，相等的对象有同一个 hash code 值，都可以用（但会影响性能）

always override hashCode when you override equals

Equality and hashCode Mutable Types Bug

Java 关注内容，所以如果两个 list 包含相同的项，java 就认为他们相同，如果 list 新 add 了一个项目，java 会认为这是个新 list

当 mutable 对象被改变，hash code 也会改变

像 List、Set 和 Map 这样的集合使用 observational equality，但是像 StringBuilder 和 arrays 这样的可变类使用 behavioral equality

总结，对于 mutable object，equals() 应该使用 behavioral equality，这意味着，对于两个 reference，当且仅当他们 aliases 于同一个对象，才 equals()，所以 mutable 可以直接继承

Object 中的 equals() 和 hashCode() (也就是比较地址)

但是对于 immutable type, 必须重写!

```
Integer x = new Integer(3);
Integer y = new Integer(3);
x.equals(y)      // returns true
```

- But there's a subtle problem here; == is overloaded :

For reference types like Integer, it implements **referential equality**:

```
x == y          // returns false
```

but for primitive types like int, == implements **behavioral equality**:

```
(int)x == (int)y // returns true
```

所以 Integer 和 int 不能乱换, 会出 bug

INTSet

Index 才是存的数组, value 只有 True 和 False, True 表示 set 中有, False 表示没有

问题:

很浪费内存 (因为数组多大就能存多少数字, 所以会要一个很大的数组)
只能存 int

ENGSet

首字母对应数字, 也就是 index; 26 个字母过后, 开始代表首 2 字母, 以此类推。于是可以把字母组合变成数字

如果还要存别的英文字符, 用 ASCII 码 (126 个) (**ASCII-based SET**)

还要存别的字符 (中文), 用 Unicode (**Unicode-based SET**)

但是太大会导致 integer Overflow

Integer 的上限在那里, 所以必然会有冲突

Handling Collision

如果有 N 个项目都有相同的 hash code h, 那么在对应的位置 h, 不要存 True 或者 False, 存他们的 bucket (可以用 SLList, ARList, ARSet, 任何可以搜索和迭代的数据结构都可以) (这被称作 **Separate Chaining**)

如果 bucket h 是空的, 创造一个新 list 包含 x, 并存在 h

如果 bucket h 非空, 将 x 加入 list, 如果 list 里已经有 x 了, 则啥都不干

Hash Table

我们刚刚创建了所谓的 Hash Table

该项由 hash 函数转换为称为 hash code 的整数表示形式

然后，通常使用 **模除 modulus operator** 将 hash code 缩减为有效索引

和 INTSet or ENGSet 相比存储空间需要的更少，但是运行时间从 $O(1)$ 变成了 $O(Q)$ ， Q 是最长的 list 的长度（最坏情况会变成 $O(N)$ ）

总量是 N ，bucket 数量为 M ，则 $Q=N/M \Rightarrow O(N)$

解决方法：将 M 以与 N 相同的速率增大

当 N/M 大于 1.5，将 M 翻倍（如果太满了，就翻倍！ N/M 被称作 load factor，这表示了 hash table 有多满）

此时 $M=O(N)$ ，那么 $O(N/M)=O(1)$ ，但是多了 resizing 的过程！

在 resizing 中，大部分 add 操作都是 $O(1)$ ，个别是 $O(N)$ ；类似于 ARList，只要我们按乘法因子调整大小，平均运行时间仍然是 $O(1)$

Negative Hash Code in Java

因为 $-1 \% 4 \Rightarrow -1$ ，会出现 index error！

解决方法：使用 Math.floorMod

```
public class ModTest {  
    public static void main(String[] args) {  
        System.out.println(-1 % 4); → -1  
        System.out.println(Math.floorMod(-1, 4)); → 3  
    }  
}
```

Typical Base

典型的 hash code 的 base 是一个小素数

素数的原因：

Never even 不是偶数，避免溢出导致变成 0；

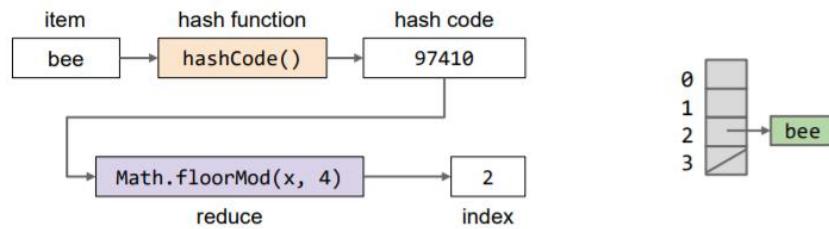
产生的 hash code 与 buckets 数的关系不良的可能性较低，例如 hash code 是 buckets 的倍数（会导致全是 0）

Small 的原因：节省计算成本

总结

Summary for Hash Table

- Hash table (HASet):
 - Item is converted into a hash code using good hashCode method
 - The hash code is then reduced to a valid index
 - Item is then stored in a bucket corresponding to that index
- Resizing hash table (for your HAMap):
 - Double when load factor N/M exceeds some constant, say 1.5 or 0.75
 - If items are spread out nicely, you get O(1) average runtime



Higher order function:

用另一个函数作为输入的函数

例如: function applyTwice(func, x) : func(func(x))

so, given function `multByTen(y) : 10 * y`

we have that, `applyTwice(multByTen, 5) → 500`

Java 7 之前要这样做

- 1 创造一个 interface 来声明要当输入的函数
- 2 创造一个 class 做这个 interface 的子类, 然后实现这个函数
- 3 实现 applyTwice 函数, 将这个 interface 作为输入, 并且调用 interface 里的那个函数
- 4 调用 applyTwice 函数, 将一个 2 中 class 的实现作为输入

Java 8 之后只要

```

import java.util.function.Function;

public class Java8HigherOrderFunc {
    public static int multByTen(int x) {
        return 10 * x;
    }

    public static int applyTwice(Function<Integer, Integer> f, int x) {
        return f.apply(f.apply(x));
    }

    public static void main(String[] args) {
        int result = applyTwice(Java8HigherOrderFunc::multByTen, 5);
        System.out.println(result);
    }
}

```

Natural Order

Natural Order 是指 Comparable 的 compareTo 方法所隐含的顺序，例如 Dog 对象的大小但是有时需要别的顺序

Additional Orders

```

private static class NameComparator implements Comparator<Dog> {
    public int compare(Dog d1, Dog d2) {
        return d1.name.compareTo(d2.name);
    }
}

public static Comparator<Dog> getNameComparator() {
    return new NameComparator();
}

```

Concurrency

意味着同时进行多个计算

Shared memory——在 Concurrency shared memory 模型中，Concurrency 模块通过读写内存中的共享对象进行交互，

例如：A 和 B 可能是同一台计算机中的两个处理器（或处理器内核），共享相同的物理内存；
A 和 B 可能是同一个 Java 程序中的两个线程（将解释线程是什么），共享相同的 Java 对象
Message passing——在消息传递模型中，Concurrency 模块通过通信信道相互发送消息进行交互；模块发送消息，传入每个模块的消息排队等待处理

例如：A 和 B 可能是网络中的两台计算机，通过网络连接进行通信；A 和 B 可能在同一台计算机上运行的两个程序，其输入和输出通过管道连接，就像在命令提示符中键入 ls | grep 一样

Processes

进程是与同一台机器上的其他进程隔离的正在运行的程序的实例——它有自己的机器内存的私有部分；

进程抽象是一个虚拟机——它让程序好像拥有了整个机器——就像一台新的计算机被创造出来，有了新的内存，只是为了运行那个程序

就像通过网络连接的计算机一样，进程之间通常不共享内存——不能访问别的进程

Thread

线程是运行程序中的一个控制点——把它想象成正在运行的程序中的一个地方；加上一堆方法调用（因此线程可以在到达 return 语句时返回堆栈）

正如进程表示虚拟计算机一样，线程抽象表示虚拟处理器，生成一个新线程模拟在由进程表示的虚拟计算机中生成一个新的处理器，这个新的虚拟处理器与进程中的其他线程运行相同的程序并共享相同的内存

线程自动准备好共享内存，因为线程共享进程中的所有内存（要获得线程本地内存很难，还需要通过创建和使用队列数据结构来显式地设置消息传递）

一旦开始跑 Java 程序，程序会启动一个线程，称为 main()，该线程也被称为 main thread

Time-slicing

当线程数多于处理器数时，并发性是通过 time slicing 来模拟的，这意味着处理器在线程之间切换；

在大多数系统上，time slicing 的发生是不可预测和不确定的，这意味着线程可能随时暂停或恢复

Anonymous Class

匿名类是没有自己类名的接口的实现；

通常，当我们实现一个接口时，我们是通过声明一个类来实现的（例如，public class StringLengthComparator implements Comparator<String>）

用 Comparator 进行比较：

With a Comparator:

```
// uses StringLengthComparator declared in previous slide
Comparator<String> compareByLength = new StringLengthComparator();
SortedSet<String> strings = new TreeSet<>(compareByLength);
strings.addAll(Arrays.asList("yolanda", "zach", "alice", "bob"));
// strings is { "bob", "zach", "alice", "yolanda" } ←
```

匿名类声明实现接口的未命名类，并立即创建该类的唯一实例，例子：

```
// no StringLengthComparator class!
SortedSet<String> strings = new TreeSet<>(new Comparator<String>() {
    @Override public int compare(String s1, String s2) {
        if (s1.length() == s2.length()) {
            return s1.compareTo(s2);
        }
        return s1.length() - s2.length();
    }
});
```

匿名类的**优点**:

如果我们**只在这一段代码中使用** comparator, 那么我们已经通过使用匿名类缩小了它的范围; 对于命名类, 任何其他代码都可以开始使用 StringLengthComparator, 具体取决于 StringLengthComparator

读者**不再需要在别处搜索比较器的细节**; 一切都在这里

匿名类的**缺点**:

如果我们不止一次地需要同一个比较器, 我们可能会受到**复制和粘贴的诱惑**; 命名类比较 DRY

如果比较器的实现很长, 它会**中断周围的代码, 使其更难理解**; 一个命名类可以分离为一个模块化的部分

所以匿名类适合于方法的短期一次性实现

利用匿名类启动线程

```
public static void main(String[] args) {

    new Thread(new Runnable() {
        public void run() {
            System.out.println("Hello! It's me, a thread!");
        }
    }).start();
}
```

Interleaving

线程交替运行

Race Condition

意味着程序的正确性（后置条件和不变量的满足）取决于并发计算 A 和 B 中事件的 relative timing 相对计时（A 在与 B 赛跑）

Java 编译器，实际上是处理器本身，对于它将从您的代码中生成什么样的低级操作不做任何承诺

关键的教训是，你不能通过看一个表达式来判断它是否在 **Race Condition** 下是安全的

Reordering

更糟

变量的更改顺序都会不同

```
answer = 42;
```

```
ready = true;
```

可能会先存储 ready! 这会导致另一个线程的循环中，检测到 ready 改变但是 answer 没变！

使用测试很难发现 **Race Condition**，也很难定位
并发错误的再现性很差，并不是每次都会发生

Heisenbugs

与 bohrbug 相反，bohrbug 在你看它的时候会反复出现，具有复现性；顺序编程中几乎所有的 bug 都是 bohrbug；

而观察的时候，heisenbugs 可能会消失

Question 1

Incorrect

Mark 0.00 out of 10.00

Here is the code again from the slide Autoboxing and Equality:

```
1. Map<String, Integer> a = new HashMap<>(), b = new HashMap<>();
2. a.put("c", 130); // put ints into the map
3. b.put("c", 130);
```

What is the compile-time type of the expression 130?

After executing `a.put("c", 130)`, what is the runtime type that is used to represent the value 130 in the map?

What is the compile-time type of `a.get("c")`?

Select one:

- a. int, Integer, Integer
- b. int, Integer, int ✗
- c. Integer, int, int
- d. int, int, Integer
- e. Integer, int, Integer
- f. Integer, Integer, int

Your answer is incorrect.

30 is an integer literal, so its compile-time type is int.

In the `Map<String, Integer>`, the keys are Strings and the values are Integers. So when 130 is placed in the map, it is automatically boxed up into a fresh Integer object.

The `get()` operation for a `Map<K, V>` returns values of type `V`, so for a `Map<String, Integer>`, the type would be `Integer`.

The correct answer is: int, Integer, Integer

The expression `new Runnable() { ... }` creates a new object that implements `Runnable`, which will be passed as a parameter to `new Thread()`. Note especially that the code inside the anonymous class is not executed yet. It won't be executed until its `run()` method is called.

Once we have the `Runnable` object, the next thing that happens is the call to `new Thread()`, which creates a new `Thread` object.

Then `start()` is called on that new `Thread` object.

The thread then starts, and `Thread.start()` calls `run()` on the `Runnable` object.

Inside the body of `run()`, the `println` statement executes.

Finally, the `run()` method returns, and the thread finishes.

The correct answer is:

For this code that starts a thread:

```
1. new Thread(new Runnable() {
2.     public void run() {
3.         System.out.println("Hello! It's me, a thread!");
4.     }
5. }).start();
```

Put the following events in the order that they occur.

1. [a `Runnable` object is created]
2. [a `Thread` object is created]
3. [`start()` is called]
4. [`run()` is called]
5. ["Hello! It's me, a thread!" is printed]
6. [`run()` returns]

先匿名类创建，然后线程创建，然后运行 `start()`，然后运行 `run()` 里的东西。因为 `new Thread` 里面只是一个匿名类的声明，还没有运行它呢。

```

3.     Thread clotho = new Thread(new Runnable() {
4.         public void run() { System.out.println("spinning"); };
5.     });
6.     clotho.start();
7.     new Thread(new Runnable() {
8.         public void run() { System.out.println("measuring"); };
9.     }).start();
10.    new Thread(new Runnable() {
11.        public void run() { System.out.println("cutting"); };
12.    });
13. }
14.
}

```

How many new Thread objects are created?

Select one:

- a. 3
- b. 2
- c. 1
- d. 0
- e. 4
- f. 5
- g. 6

只要 new 了一个线程，就是创建了线程; .start()之后才是运行

Suppose we run main in this program, which demonstrates two common bugs:

```

1.  public class Farce {
2.      public static void main(String[] args) {
3.          Thread nona = new Thread(new Runnable() {
4.              public void run() { System.out.println("spinning"); };
5.          });
6.          nona.run();
7.          Runnable decima = new Runnable() {
8.              public void run() { System.out.println("measuring"); };
9.          };
10.         decima.run();
11.         // ...
12.     }
13.
}

```

How many new threads are run?

Select one:

- a. 3
- b. 2
- c. 1
- d. 0
- e. 4
- f. 5
- g. 6



不能.run(), .run()只是单纯的跑了一遍

Lecture 13

Priority Queue

优先级队列是支持以下操作的抽象数据结构:

add an item: add(T item)

delete smallest item: T delMin()

Applications of Priority Queues

- Applications of Priority Queues:
 - Customers in a line, Colliding particles simulations
 - Reducing roundoff error in numerical computation
 - Huffman codes
 - Dijkstra's algorithm, Prim's algorithm
 - Sum of powers
 - A* search
 - Maintain largest M values in a sequence
 - Load balancing, Interrupt handling
 - Bin packing, Scheduling
 - Bayesian spam filter

Interface MinPQ with comparable items:

```
public interface MinPQ<T extends Comparable<T>> {  
    void add(T item);  
    T getMin();  
    T delMin();  
    boolean isEmpty();  
    int size();  
}
```

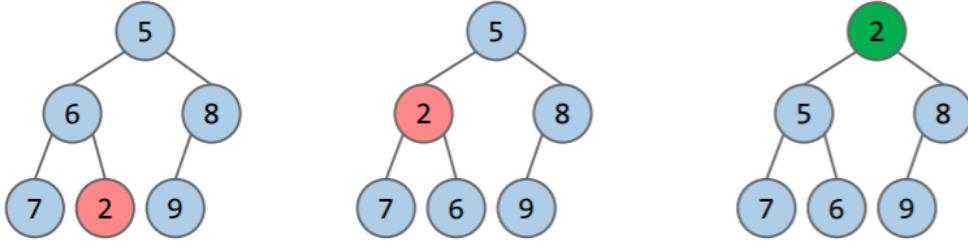
用 Binary Heap 实现

完成: 平衡的二叉树, 可以在底层左对齐

堆属性: 父级的项/优先级小于或等于子级的项/优先级

Restoring Heap Property : Swim

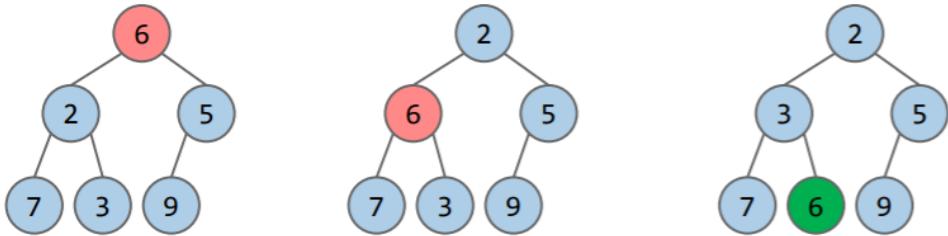
- What should we do if a child becomes smaller than its parent ?



```
private void swim(int k) {
    while (k > 1 && greater(k/2, k)) {
        swap(k, k/2);
        k = k/2;
    }
}
```

Restoring Heap Property : Sink

- What should we do if a parent becomes larger than one child / both of its children ?



```
private void sink(int k) {
    while (2*k <= size) {
        int j = 2*k;
        if (j < size && greater(j, j+1)) j++;
        if (!greater(k, j)) break;
        swap(k, j);
        k = j;
    }
}
```

Add: O(log N)

Remove min: O(log N)

Thread Safety

在共享内存并发中，基本上有四种方法可以使变量访问安全

Confinement: 线程间不共享变量和数据

Immutability: 使共享变量不可访问或共享数据不可更改

Threadsafe data type: 将共享数据封装在现有的 threadsafe 数据类型中，该数据类型为您进行协调

Synchronization : 使用 synchronization 来防止线程同时访问共享变量或数据 (synchronization 是构建自己的线程安全数据类型所需要的)

如果 **数据类型或静态方法在从多个线程使用时行为正确**，则它是 Threadsaf 的，**而不管这些线程是如何执行的，并且不要求调用代码进行额外的协调**

"behaves correctly"意味着满足 Spec 并且保留它的 rep invariant

"regardless of how threads are executed"意味着线程可能位于多个处理器上，或者同一处理器上的 timesliced

"without additional coordination"这意味着数据类型不能在与计时相关的调用者上设置前提条件，比如“set () 正在进行时不能调用 get () ”

Iterator 不满足，它规定不能在 iterating 的同时修改它

Confinement

线程限制是一个简单的想法：通过将数据限制在单个线程中，可以避免在可重新分配的引用和可变数据上的 race（使用本地变量，别用全局/ static）

Immutability

我们实现线程安全的第二种方法是使用 un assignable references 不可访问的引用和 immutable data types 不可变的数据类型——不变性解决了共享可变数据引起的竞争条件，并通过使共享状态不可变来解决它

Final 变量是不可传递/不可变的引用，因此声明为 Final 的变量可以从多个线程安全访问只能读取变量，不能写入；要小心，因为这种安全性只适用于变量本身，而且我们仍然必须证明变量所指向的对象是不可变的

Immutable 对象仍有可能偷偷修改 rep value，这不安全！

于是

threadsafe immutable

没有修改方法；

所有 field 都是 private 且 final

没有 representation exposure

在 rep 中没有任何可变对象的改变

Using Threadsafe Data Types

在 existing threadsafe data types 现有的线程安全数据类型中存储共享的可变数据

Collection 接口是不 threadsafe 的

一个 atomic action 实际上是一次发生的——它不会将其内部操作与其他操作的内部操作交织在一起，并且在整个操作完成之前，其他线程都看不到该操作的任何效果，因此看起来它

永远不会部分完成

不要绕过 wrapper

请确保丢弃对底层非线程安全集合的引用，并且仅通过同步包装器访问它
这会在上面的代码行中自动发生，因为新的 HashMap 只传递给 synchronizedMap ()，而不会存储在其他任何地方
我们在不可修改的包装器中看到了同样的警告：底层集合仍然是可变的，引用它的代码可以规避不可变性

Synchronization

阻止线程同时访问共享数据（并发程序的正确性不应依赖于时间的偶然性）

Locks

一项 Synchronization 技术

锁是一种抽象，一次最多允许一个线程拥有它
持有一个锁是一个线程告诉其他线程的方式：“我正在处理这个东西，现在不要碰它。”

Acquire: 允许线程获得锁的所有权

如果一个线程试图获取另一个线程当前拥有的锁，它将 block，直到另一个线程释放该锁
此时，它将与试图获取锁的任何其他线程竞争

一次最多只能有一个线程拥有锁

Release: 放弃锁的所有权，允许另一个线程获得它的所有权

使用锁还告诉编译器和处理器您正在同时使用共享内存，这样寄存器和缓存将被刷新到共享存储中——避免了 reordering

Blocking

意味着线程等待（不做进一步的工作）直到事件发生

如果另一个线程（比如线程 2）持有锁 l，线程 1 上的 Acquire (l) 将被 Blocking
它等待的事件是线程 2 执行 release (l)

此时，如果线程 1 可以 Acquire l，它将继续运行其代码，并拥有锁的所有权

有可能另一个线程（比如线程 3）也在 Acquire (l) 上被阻塞

这样的话，线程 1 或线程 3（胜利者是不确定的）中的任何一个都将获取锁 l 并继续运行
另一个将继续 Blocking，等待再次释放 (l)

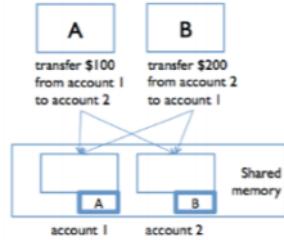
Deadlock

当使用得当和小心，锁可以防止 race conditions，但是

因为使用锁需要线程等待（acquire 会阻塞，当另一个线程持有锁时），有可能陷入两个线程互相等待的情况，两个线程都无法取得进展

Deadlock (2)

- In the figure, suppose A and B are making simultaneous transfers between two accounts in our bank
- A transfer between accounts **needs to lock both accounts**, so that money can't disappear from the system
- A and B each acquire the lock on their respective "from" account: A acquires the lock on account 1, and B acquires the lock on account 2
- Now, each must acquire the lock on their "to" account: so A is waiting for B to release the account 2 lock, and B is waiting for A to release the account 1 lock
- Stalemate! A and B are frozen in a "deadly embrace", and accounts are locked up!



当并发模块被困在一起等待彼此执行某项操作时，就会发生死锁

死锁可能涉及两个以上的模块，例如，A 可能正在等待 B, B 正在等待 C, C 正在等待 A (a cycle of dependencies)

gap buffer (多用于文字修改)

它是一个字符数组，其中有额外的空间，但是，除了在结尾有全部额外的空间，额外的空间是一个可以出现在 **buffer** 中的任何地方的 **gap**

每当需要执行插入或删除操作时，数据类型首先将 **gap** 移动到操作的位置，然后执行插入或删除操作；

如果间隙已经存在，则不需要复制任何内容—插入只会消耗间隙的一部分，删除只会扩大间隙！

间隙缓冲区特别适合于表示用户用光标编辑的字符串，因为插入和删除往往集中在光标周围，所以间隙很少移动

```
/** GapBuffer is a non-threadsafe EditBuffer that is optimized
 * for editing with a cursor, which tends to make a sequence of
 * inserts and deletes at the same place in the buffer. */
public class GapBuffer implements EditBuffer {
    private char[] a;
    private int gapStart;
    private int gapLength;
    // Rep invariant:
    // 0 <= gapStart <= a.length
    // 0 <= gapLength <= a.length - gapStart
    // Abstraction function:
    // represents the sequence a[0],...,a[gapStart-1],
    // a[gapStart+gapLength],...,a[a.length-1]
```

Steps to Develop The Data Type

Specify: define the operations (method signatures and specs)

Test: develop test cases for the operations

Rep: choose a rep

a) Implement a simple, brute-force rep first

b) Write down the rep invariant and abstraction function, and implement checkRep()

到此为止都是单线程

Synchronize: make an argument that your rep is threadsafe

Iterate: 返回优化

Lock

锁的使用非常普遍，以至于 Java 将其作为一种内置的语言特性提供

在 Java 中，每个对象都有一个与之隐式关联的锁——一个字符串、一个数组、一个 ArrayList，

而您创建的每个类，它们的所有对象实例都有一个锁

可以使用 synchronized 语句获取语句块期间的锁：

```
synchronized (lock) { // thread blocks here until lock is free  
    // now this thread has the lock  
    balance = balance + 1;  
    // exiting the block releases the lock  
}
```

像这样的同步区域提供 **mutual exclusion**: 一次只有一个线程可以在由给定对象的锁保护的同步区域中，换句话说，回到了顺序编程世界，一次只运行一个线程，至少对于引用同一对象的其他同步区域是这样

锁用于保护共享数据变量，如果对一个数据变量的所有访问都由同一个锁对象保护（由一个同步块包围），那么这些访问将被保证是原子的—不受其他线程的干扰

锁只提供与获取相同锁的其他线程的互斥——对数据变量的所有访问必须由同一个锁保护；可以在一个锁后面保护整个变量集合，但是所有模块都必须就获取和释放哪个锁达成一致但是：synchronized (obj) 仅能保证别的线程不进入 obj 的同步空间，但是他们仍然可以修改，所以每一步都需要 synchronized lock

Monitor Pattern 监视模式

例如在一个类中，每个函数套一个 synchronized (this)

```

/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    ...
    public SimpleBuffer() {
        synchronized (this) {
            text = "";
            checkRep();
        }
    }
    public void insert(int pos, String ins) {
        synchronized (this) {
            text = text.substring(0, pos) + ins + text.substring(pos);
            checkRep();
        }
    }
}

```

每个方法都有锁保护——即使是很小很琐碎的方法，比如 length () 和 toString () ——因为读操作和写操作都必须受到保护——如果读操作不受保护，那么它们就可以看到处于部分修改状态的 rep

监视器是一个方法 mutually exclusive 的类，因此一次只能有一个线程在类的实例中

简便方法是直接将 synchronized 放在签名里

```

public synchronized void insert(int pos, String ins) {
    text = text.substring(0, pos) + ins + text.substring(pos);
    checkRep();
}

```

Constructor 不要加

Locking Discipline

1. 每个共享可变变量都必须由某个锁保护——除非在获取锁的同步块内，否则不能读取或写入数据

2. 如果一个不变量包含多个共享的可变变量（甚至可能在不同的对象中），那么所有涉及的变量都必须由同一个锁保护——一旦线程获得锁，在释放锁之前必须重新建立不变量

Monitor Pattern 监视模式都满足

哪怕都是原子调用，也可能出问题，因为操作之间，内容可能会改变

有时，将数据类型的锁提供给客户机是很有用的，这样客户机就可以使用它来使用您的数据类型实现更高级别的原子操作

可以给所有的调用套一个大的 synchronized，让其变成一个原子操作

但是全用 synchronized 也不行，开销大，时间长，java 默认一些变量不允许

而且 synchronized 也并不是获取一个锁，例如声明的那个函数是 static，那么回导致其他的实例想要调用这个函数，也会被 block（本来是 safe 的）

Designing a Data Type for Concurrency

Deadlock again

Synchronization 用于保护线程安全很有效，但是会有 deadlock 的风险
线程有时必须等待其他线程退出同步区域，然后才能继续
阻塞增加了 deadlock 的可能性
当线程同时获取多个锁时，就会发生死锁，两个线程在各自等待另一个线程释放的锁时会被阻塞
两个对象可能会要求调用彼此，导致谁都无法调用（因为函数都先把自己锁了）

Deadlock Solution

Lock Ordering

对需要同时获取的锁进行排序，并确保所有代码都按此顺序获取锁
因为线程 A 和线程 B 都需要哈利和斯内普的锁，所以它们都会按照这样的顺序获得锁：首先
先是哈利的锁，然后是斯内普的锁
如果线程 A 在 B 之前获得了 Harry 的锁，那么它也会在 B 之前获得 Snape 的锁，**因为在 A 再次释放 Harry 的锁之前，B 无法继续**
对锁的排序强制对获取锁的线程进行排序，因此无法在等待图中生成循环
缺点——它不是模块化的，代码必须知道系统中的所有锁，或者至少在其子系统中；其次，代码可能很难或不可能在获得第一个锁之前准确地知道它需要哪些锁

Coarse-grained Locking

比锁排序更常见的方法，特别是对于应用程序编程（与操作系统或设备驱动程序编程相反），是使用 **grained Locking** 更粗糙的锁—使用单个锁来保护多个对象实例，甚至保护程序的整个子系统
例如，我们可以为整个社交网络设置一个锁，并使其任何组成部分上的所有操作在该锁上同步
例子，创一个很大的变量，让所有变量都属于它，然后只要锁了这个变量就行
缺点——可能会有很大的性能损失；如果您用一个锁来保护一大堆可变数据，那么您就放弃了并发访问这些数据的能力；在最坏的情况下，使用一个锁来保护所有内容，您的程序可能基本上是连续的—一次只允许一个线程进行进程

Goals of Concurrent Program Design

当我们询问并发程序是否安全无 bug 时，我们关心两个属性：
Safe——并发程序是否满足其不变量和规范——访问可变数据的 race 威胁到安全问题：你能证明一些坏事情永远不会发生吗？
Liveness——程序是否一直在运行并最终做你想做的事情，或者它是否被困在某个永远等待永远不会发生的事件的地方？能证明好事终究会发生吗？

Deadlock 威胁了 liveness

liveness 可能还需要 fairness，这意味着并发模块被赋予处理能力，以便在计算上取得进展
fairness 主要是操作系统的线程调度程序的问题，但是您可以通过设置线程优先级来影响它（无论是好是坏）

什么样的变量可以用于 multiple threads

Not only does the variable's type have to be **threadsafe**, but the variable itself should be **unreassignable**

Consider this class's rep:

```
1. public class Building {  
2.     private final String buildingName;  
3.     private int numberofFloors;  
4.     private final int[] occupancyPerFloor;  
5.     private final List<String> companyNames = Collections.synchronizedList(new ArrayList<>());  
6.     private final Set<String> roomNumbers = new HashSet<>();  
7.     private final Set<String> floorplan = Collections.synchronizedSet(roomNumbers);  
8.     ...  
9. }
```

Which of these variables are safe for use by multiple threads?

Select one or more:

- i. buildingName ✓
- ii. numberofFloors
- iii. occupancyPerFloor
- iv. companyNames ✓
- v. roomNumbers
- vi. floorplan ✗

Your answer is partially correct.

Not only does the variable's type have to be threadsafe, but the variable itself should be unreassignable. buildingName and companyNames satisfy that, but reads and writes of numberofFloors may have race conditions.

When using a synchronized collection wrapper, you have to be sure not to keep any aliases to the underlying collection. So companyNames is safe because no other variables hold a reference to the underlying ArrayList, but floorplan is not safe because roomNumbers points to the same HashSet.

You have selected too many options.

The correct answers are: buildingName, companyNames

Question 6

Incorrect

Mark 0.00 out of 10.00

Flag question

Consider this class's rep:

```
1. public class Building {
2.     private final String buildingName;
3.     private int numberOfFloors;
4.     private final int[] occupancyPerFloor;
5.     private final List<String> companyNames = Collections.synchronizedList(new ArrayList<>());
6.     private final Set<String> roomNumbers = new HashSet<>();
7.     private final Set<String> floorplan = Collections.synchronizedSet(roomNumbers);
8.     ...
9. }
```

Which of these variables **cannot** be involved in any race condition?

Select one or more:

- i. buildingName
- ii. numberOfFloors ×
- iii. occupancyPerFloor ×
- iv. companyNames
- v. roomNumbers ×
- vi. floorplan

Your answer is incorrect.

`buildingName` is unassignable and immutable, so it can't be involved in any race condition.

`companyNames` might still be involved in a race condition caused by (otherwise safe) mutations to the list, e.g.:

```
1. if (companyNames.size() > 0) { String firstCompany = companyNames.get(0); }
```

If another thread could empty the `companyNames` list between the size check and the `get` call, then this code will fail.

The correct answer is buildingName.

Question 9

Partially correct

Mark 2.50 out of 10.00

 Flag question

Suppose `list` is an instance of `ArrayList<String>`.

What is true while thread A is in a `synchronized (list) { ... }` block?

Select one or more:

- i. it owns the lock on `list`
- ii. it does not own the lock on `list`
- iii. no other thread can use observers of `list`
- iv. no other thread can use mutators of `list`
- v. no other thread can acquire the lock on `list`
- vi. no other thread can acquire locks on elements in `list`

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: it owns the lock on `list`, no other thread can acquire the lock on `list`

Question 10

Correct

Mark 10.00 out of 10.00

Flag question

Suppose we run this code:

```
1. synchronized (obj) {  
2.     // ...  
3.     synchronized (obj) { // 1  
4.         // ...  
5.     }  
6.     // 2  
7. }
```

On the line // 1 do we experience deadlock?

If we don't deadlock, on the line // 2, does the thread own the lock on obj?

Select one:

- a. No, we do not experience deadlock; Yes, the thread owns the lock on obj
- b. Yes, we experience deadlock; No, the thread does not own the lock on obj
- c. Yes, we experience deadlock; Yes, the thread owns the lock on obj
- d. No, we do not experience deadlock; No, the thread does not own the lock on obj

Your answer is correct.

In Java, a thread is allowed to re-acquire a lock it already owns. The technical term for this is *reentrant locks*.

Acquire and release come in pairs, and synchronized blocks on the same object can be safely nested inside each other. This means that a lock actually stores a *counter* of the number of times that its owner has acquired it without yet releasing it. The thread continues to own it until each acquire has had its corresponding release, and the counter has fallen to zero. So on the "do we own the lock" line, the thread does still have a lock on obj.

Nested synchronization on the same lock happens frequently, e.g. if a synchronized method is recursive, or if one synchronized method calls another synchronized method on this.

The correct answer is: No, we do not experience deadlock; Yes, the thread owns the lock on obj

在 Java 中，线程可以重新获取它已经拥有的锁。这个术语是 reentrant lock。

同一锁上的嵌套同步经常发生，例如，如果同步方法是递归的，或者如果一个同步方法调用此锁上的另一个同步方法。

完全 block 才是 deadlock，将要 block (哪怕最终总会 block) 都不叫 deadlock

Question 12

Incorrect

Mark 0.00 out of 10.00

[Flag question](#)

In the code below three threads 1, 2, and 3 are trying to acquire locks on objects alpha, beta, and gamma.

Thread 1

```
synchronized (alpha) {  
    // using alpha  
    // ...  
}  
  
synchronized (gamma) {  
    synchronized (beta) {  
        // using beta & gamma  
        // ...  
    }  
}  
// finished
```

Thread 2

```
synchronized (gamma) {  
    synchronized (alpha) {  
        synchronized (beta) {  
            // using alpha, beta, & gamma  
            // ...  
        }  
    }  
}  
// finished
```

Thread 3

```
synchronized (gamma) {  
    synchronized (alpha) {  
        // using alpha & gamma  
        // ...  
    }  
  
    synchronized (beta) {  
        synchronized (gamma) {  
            // using beta & gamma  
            // ...  
        }  
    }  
}  
// finished
```

This system is susceptible to deadlock.

For each of the scenarios below, determine whether the system is in deadlock if the threads are currently on the indicated lines of code.

Scenario C

Thread 1 running `synchronized (beta)`
Thread 2 blocked on `synchronized (gamma)`
Thread 3 blocked on 1st `synchronized (gamma)`

Scenario D

Thread 1 blocked on `synchronized (beta)`
Thread 2 finished
Thread 3 blocked on 2nd `synchronized (gamma)`

Select one:

a. C not deadlock, D deadlock

b. C deadlock, D deadlock ✗

c. C deadlock, D not deadlock

d. C not deadlock, D not deadlock

Your answer is incorrect.