哈尔滨工业大学
HARBIN INSTITUTE OF TECHNOLOGY

4 Data Type and Type Checking
数据类型与类型检验

Wang Zhongjie
rainy@hit.edu.cn

May 20, 2021

Objective of this lecture

1. **Get to know basic knowledge about data type, and static and dynamic type checking in programming language, especially Java.** ——静态/动态类型检查

2. **Understand mutability and mutable objects** ——可变/不变的数据类型

3. **Identify aliasing and understand the dangers of mutability** ——可变数据类型的危险性

4. **Use immutability to improve correctness, clarity and changeability** ——不变数据类型的优越性

5. **Use snapshot diagram to demonstrate the state of specific time during a program's execution.** ——用**Snapshot**图理解数据类型

6. **Use** `Arrays, Collections` **and** `Enum` **to deal with complex data types** ——用集合类表达复杂数据类型

7. **Know the harm of** `Null` **references and avoid it**

Outline

1. **Data type in programming languages**

2. **Static vs. dynamic data type checking**

3. **Mutability & Immutability**

4. **Snapshot diagram**

5. **Complex data types: Arrays and Collections**

6. **Useful immutable types**

7. **Summary**

前两章回答了："什么是"高质量的软件"、如何从不同维度刻画软件、软件构造的基本过程和步骤

本章：软件构造的理论基础——ADT
　　　软件构造的技术基础——OOP

Reading

- **MIT 6.031：01、08**

- **CMU 17-214：Oct. 01**

- **Java编程思想：第1-6章、第11章 for all lectures in Chapter 3**

1 Data type in programming languages

Types and Variables for general computer language

- A **type** is a set of values, along with operations that can be performed on those values. 数据类型（为什么设置数据类型？和计算机的体系结构有关吗？）

- **Examples:**

  - boolean: Truth value (true or false).

  - int: Integer (0, 1, -47).

  - double: Real number (3.14, 1.0, -2.1).

  - String: Text ("hello", "example").

- **Variables: Named location that stores a value of one particular type**

  - Form: TYPE NAME;

  - Example: String foo;

  变量：用特定数据类型定义，可存储满足类型约束的值

Types in Java（如何理解数据类型？）

- **Java has several primitive types 基本数据类型, such as:**

  - int (for integers like 5 and -200, but limited to the range $\pm$ 2^{31}, or roughly $\pm$ 2 billion)

  - long (for larger integers up to $\pm$ 2^{63})

  - boolean (for true or false)

  - double (for floating-point numbers, which represent a subset of the real numbers)

  - char (for single characters like 'A' and '$')

- **Java also has object types 对象数据类型, for example:**

  - String represents a sequence of characters.

  - BigInteger represents an integer of arbitrary size.

- **By Java convention, primitive types are lowercase, while object types start with a capital letter.**

Types in Java

Primitives	Object Reference Types
`int, long, byte, short, char, float, double, boolean`	Classes, interfaces, arrays, enums, annotations
No identity except their value 只有值，没有ID (与其他值无法区分)	Have identity distinct from value 既有ID，也有值
Immutable 不可变的	Some mutable, some not 可变/不可变
On stack, exist only when in use 在栈中分配内存	On heap, garbage collected 在堆中分配内存
Can't achieve unity of expression	Unity of expression with generics
Dirt cheap 代价低	More costly代价昂贵

Hierarchy of object types 对象类型形成层次结构

- **The root is** Object **(all non-primitives are objects)**
  - All classes except Object have one parent class, specified with an extends clause，单亲继承！----Java 类

    ```
    class Guitar extends Instrument { ... }
    ```

- **If** extends **clause omitted, defaults to** Object

- **A class is an instance of all its superclasses** 继承关系
  - Inherits visible fields and methods from its superclasses
  - Can override methods to change their behavior
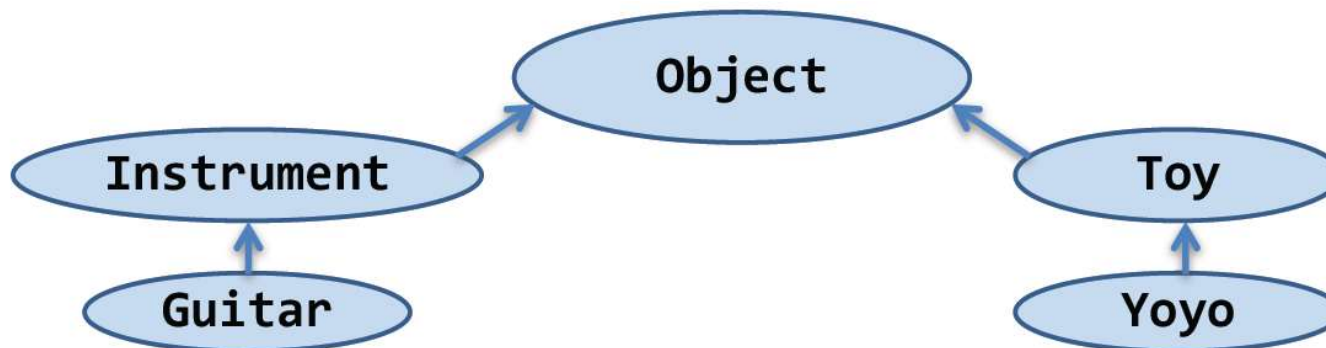
Boxed primitives

- **Immutable containers for primitive types**
  将基本类型包装为对象类型

  – Boolean, Integer, Short, Long, Character, Float, Double

- **Canonical use case is collections**
  通常是在定义集合类型的时候使用它们

- **Don't use boxed primitives unless you have to!**
  一般情况下，尽量避免使用

- **Language does autoboxing and auto-unboxing** 一般可以自动转换

Operators

- **Operators: symbols that perform simple computations** 操作符
  - Assignment: =   *赋值：x=y，将y的值赋给x。对象类型如何理解？*
  - Addition: +
  - Subtraction: -
  - Multiplication: *
  - Division: /

- **Order of Operations: follows standard math rules**
  - 1. Parentheses 圆括号
  - 2. Multiplication and division 乘与除
  - 3. Addition and subtraction 加和减

- **String concatenation (+)**
  - String text = "hello" + " world";
  - text = text + " number " + 5;   // text = "hello world number 5"

Operations

- **Operations 操作 are functions that take inputs and produce outputs (and sometimes change the values themselves).**

  – As an infix, prefix, or postfix （*中缀、前缀或后缀*） **operator**. For example, `a + b` invokes the operation + : `int` × `int` → `int` .

  – As a **method** of an object. For example, `bigint1.add(bigint2)` calls the operation add: `BigInteger` × `BigInteger` → `BigInteger` .

  – As a **function**. For example, `Math.sin(theta)` calls the operation `sin: double` → `double`. Here, `Math` is not an object. It's the class that contains the `sin` function.

Overloading operators/operations 重载

- **Some operations are overloaded in the sense that the same operation name is used for different types 同样的操作名可用于不同的数据类型**

- **The arithmetic operators +，-，*，/ are heavily overloaded for the numeric primitive types in Java.**

- **Methods can also be overloaded. Most programming languages have some degree of overloading.**

- **(to be discussed in Section 3.3 OOP)**

2 Static vs. dynamic data type checking

Conversion by casting 类型转换

```
int a = 2;                    // a = 2

double a = 2;                 // a = 2.0 (Implicit隐性的)

int a = (int) 18.7;           // a = 18

double a = (double)2/3;       // a = 0.6666…


int a = 18.7;                 // ERROR

String a = 1;                 // ERROR

double a = 2/3;               // a = 0.0
```

Static Typing vs. Dynamic Typing

- **Java is a statically-typed language.** 静态类型语言
  - The types of all variables are known at compile time (before the program runs), and the compiler can therefore deduce the types of all expressions as well.
  - If `a` and b are declared as `int`, then the compiler concludes that a+b is also an `int`.
  - The Eclipse environment does this while you're writing the code, in fact, so you find out about many errors while you're still typing.
  - 在编译阶段进行类型检查（检查的细自然有好处！）
- **In dynamically-typed languages like Python, this kind of checking is deferred until runtime (while the program is running).** 动态类型语言
  - 在运行阶段进行类型检查

Static Checking and Dynamic Checking

- **Three kinds of automatic checking that a language can provide:**

  - **Static checking**: the bug is found automatically before the program even runs. 静态类型检查

  - **Dynamic checking**: the bug is found automatically when the code is executed. 动态类型检查

  - **No checking**: the language doesn't help you find the error at all. You have to watch for it yourself, or end up with wrong answers. 无检查（*例如：C语言中不检查数组越界，编程者要十分小心！*）

- **Needless to say, catching a bug statically is better than catching it dynamically, and catching it dynamically is better than not catching it at all.** 静态类型检查 >> 动态 >> 无检查

Mismatched Types

- **Java verifies that types always match:** （*赋值语句的左右必须类型完全一致！*）

- `String five = 5; // ERROR!`

test.java.2: incompatible types

found: `int`

required: `java.lang.String`

`String five = 5;`

Static checking

- **Static checking means checking for bugs at compile time.**

- **Bugs are the bane毒药 of programming.**

- **Static typing prevents a large class of bugs from infecting your program: to be precise, bugs caused by applying an operation to the wrong types of arguments.**

- **If you write a broken line of code like:**

$$"5" * "6"$$

that tries to multiply two strings, then static typing will catch this error while you're still programming, rather than waiting until the line is reached during execution.

静态类型检查：可在编译阶段发现错误，避免了将错误带入到运行阶段，可提高程序正确性/健壮性

Static checking

- **Syntax errors** 语法错误, like extra punctuation or spurious words. Even dynamically-typed languages like Python do this kind of static checking.

- **Wrong names** 类名/函数名错误, like `Math.sine(2).`(The right name is `sin`)

- **Wrong number of arguments** 参数数目错误, like `Math.sin(30, 20).`

- **Wrong argument types** 参数类型错误, like `Math.sin("30").`

- **Wrong return types** 返回值类型错误, like `return "30";` from a function that's declared to return an `int`.

Dynamic checking

- **Illegal argument values** 非法的参数值. For example, the integer expression `x/y` is only erroneous when y is actually `zero`; otherwise it works. So in this expression, `divide-by-zero` is not a static error, but a dynamic error.

- **Unrepresentable return values** 非法的返回值, i.e., when the specific return value can't be represented in the type.

- **Out-of-range indexes** 越界, e.g., using a negative or too-large index on a string.

- **Calling a method on a** `null` **object reference.** 空指针

Static vs. Dynamic Checking

- **Static checking tends to be about types, errors that are independent of the specific value that a variable has.**

  - Static typing guarantees that a variable will have *some* value from that set, but we don't know until runtime exactly which value it has.

  - So if the error would be caused only by certain values, like divide-by-zero or index-out-of-range then the compiler won't raise a static error about it.

- **Dynamic checking, by contrast, tends to be about errors caused by specific values.**

静态检查：关于"类型"的检查，不考虑值
动态检查：关于"值"的检查

Classroom Exercises

```
int n = 5;
if (n) {
    n = n + 1;
}
```

Type mismatch: cannot convert from int to boolean
if (`boolean`)

Static error

```
int big = 200000;
big = big * big;
```

1345294336
越界

```
double probability = 1/5;
```

0.0类型转换错误

Dynamic error

```
int sum = 0;
int n = 0;
int average = sum/n;
```

Exception in thread "main" java.lang.ArithmeticException: / by zero 除零

No error, but wrong answer

```
double sum = 7;
double n = 0;
double average = sum/n;
```

Infinity 除零?

3 Mutability and Immutability

Assignment 赋值

- **Use "=" to give variables a value**

- **Example:**
  - `String foo;`
  - `foo = "IAP 6.092";`

- **Assignment can be combined with a variable declaration**

- **Example:**
  - `double badPi = 3.14;`
  - `boolean isJanuary = true;`

Changing a variable or its value

- **What's the distinction between changing a variable and changing a value?** 改变一个变量、改变一个变量的值，二者有何区别？

  - When you assign to a variable, you're changing where the variable's arrow points. You can point it to a different value. 改变一个变量：将该变量指向另一个值的存储空间

  - When you assign to the contents of a mutable value – such as an array or list – you're changing references inside that value. 改变一个变量的值：将该变量当前指向的值的存储空间中写入一个新的值。

- **Change is a necessary evil.** 变化是"罪恶"，但程序不能没有变化

- **Good programmers avoid things that change, because they may change unexpectedly.** 尽可能避免变化，以避免副作用

Immutability 不变性

- **Immutability is a major design principle.** 不变性：重要设计原则

- **Immutable types are types whose values can never change once they have been created.** 不变数据类型：一旦被创建，其值不能改变

- **Java also gives us immutable references: variables that are assigned once and never reassigned.** 如果是引用类型（指针），也可以是不变的：一旦确定其指向的对象，不能再被改变

  – To make a reference immutable, declare it with the keyword `final`

  ```
  final int n = 5;

  final Person a = new Person("Ross");
  ```

- **If the Java compiler isn't convinced that your `final` variable will only be assigned once at runtime, then it will produce a compiler error. So `final` gives you static checking for immutable references.**
  如果编译器无法确定**final**变量不会改变，就提示错误，这也是静态类型检查的一部分。

Immutability

- **It's good practice to use `final` for declaring the parameters of a method and as many local variables as possible.** 所以，尽量使用 **`final`变量作为方法的输入参数、作为局部变量。**

- **Like the type of the variable, these declarations are important documentation, useful to the reader of the code and statically checked by the compiler. `final`表明了程序员的一种"设计决策"**

- **Note:**

  – A `final` class declaration means it cannot be inherited. `final`类无法派生子类

  – A `final` variable means it always contains the same value/reference but cannot be changed `final`变量无法改变值/引用（注意区别！）

  – A `final` method means it cannot be overridden by subclasses `final`方法无法被子类重写

Mutability and Immutability

- **Objects are immutable**: once created, they always represent the same value.

- **Objects are mutable**: they have methods that change the value of the object.

不变对象：一旦被创建，始终指向同一个值/引用
可变对象：拥有方法可以修改自己的值/引用

`String` as an immutable type

- **`String` is an example of an immutable type.**

- **A `String` object always represents the same string.**

- **Since `String` is immutable, once created, a `String` object always has the same value.**

- **To add something to the end of a `String`, you have to create a new `String` object:**

```
String s = "a";
s = s.concat("b"); // s+="b"  and  s=s+"b"  also  mean  the  same  thing
```
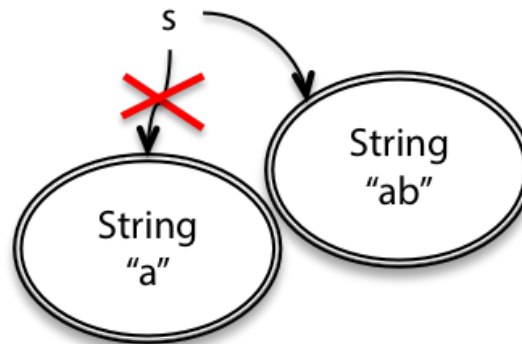

Note: this is a snapshot diagram

StringBuilder as a mutable type

- **StringBuilder is an example of a mutable type.**

- **It has methods to delete parts of the string, insert or replace characters, etc.**

- **This class has methods that change the value of the object, rather than just returning new values:**

```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```

sb

StringBuilder
~~"a"~~ "ab"
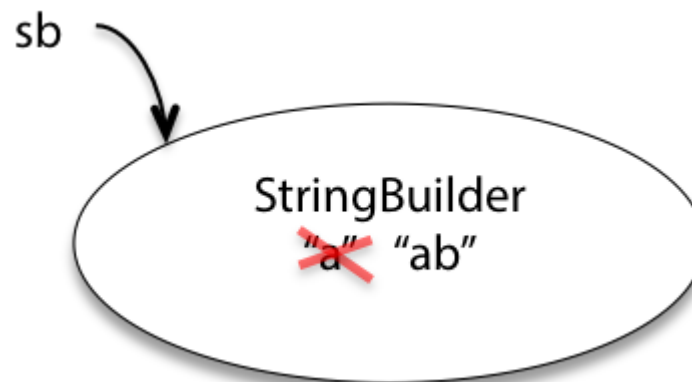
Mutability and Immutability

- **So what? In both cases, you end up with** `s` **and** `sb` **referring to the string of characters** "ab"**.** 有区别吗？最终的值都是一样的

  – The difference between mutability and immutability doesn't matter much when there's only one reference to the object. 当只有一个引用指向该值，没有区别

- **But there are big differences in how they behave when there are other references to the object.** 有多个引用的时候，差异就出现了

  – When another variable `t` points to the same `String` object as `s`, and another variable `tb` points to the same `StringBuilder` as `sb` , then the differences between the immutable and mutable objects become more evident.

```
String t = s;
t = t + "c";

StringBuilder tb = sb;
tb.append("c");
```
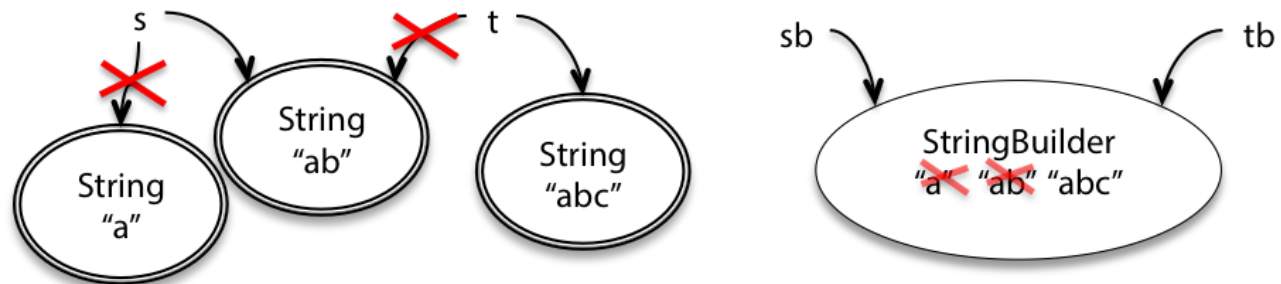
Advantage of mutable types

- **Using immutable strings, this makes a lot of temporary copies** 使用不可变类型，对其频繁修改会产生大量的临时拷贝(需要垃圾回收)

  - The first number ("0") is actually copied n times in the course of building up the final string, the second number is copied n-1 times, and so on.

  - It actually costs O(n^2) time just to do all that copying, even though we only concatenated n elements.

- StringBuilder **is designed to minimize this copying.** 可变类型最少化拷贝以提高效率

  - It uses a simple but clever internal data structure to avoid doing any copying at all until the very end, when you ask for the final String with a toString() call.

```
String s = "";
for (int i = 0; i < n; ++i) {
    s = s + n;
}
```

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; ++i) {
    sb.append(String.valueOf(i));
}
String s = sb.toString();
```

Advantage of mutable types

- **Getting good performance is one reason why we use mutable objects.** 使用可变数据类型，可获得更好的性能

- **Another is convenient sharing:** two parts of your program can communicate more conveniently by sharing a common mutable data structure. 也适合于在多个模块之间共享数据

- **"Global variables"**

- **But you must have known the disadvantages of global variables…**

Exercise

```java
StringBuilder sb = new StringBuilder(" Hello ");
sb.append(" World ");
System.out.println(sb.toString().trim( ));
```

What will be the output of the following code snippet?

```java
String s = " Hello ";
s += " World ";
s.trim( );
System.out.println(s);
```

See the specification of `String.trim()`:
Returns:
A copy of this string with leading and trailing white space removed, or this string if it has no leading or trailing white space.

3 String objects are created, and 2 of them become unreachable as there are no references to them, and gets garbage collected.

S → " Hello "

S → " Hello "
   → " Hello World "

S → " Hello "
  → " Hello World "
   "Hello World"
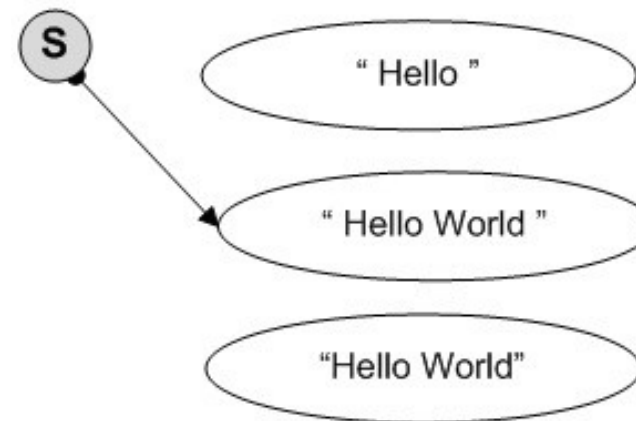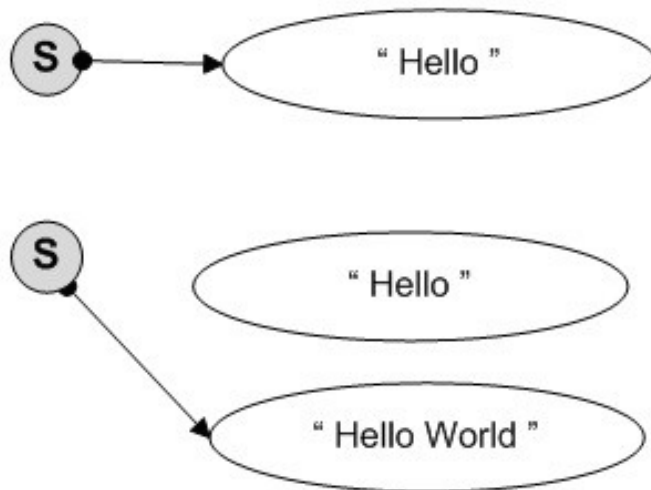
Risks of mutation

- **Since mutable types seem much more powerful than immutable types, why on earth would you choose the immutable one?** 既然如此，为何还要用不可变类型？

  - `StringBuilder` should be able to do everything that `String` can do, plus `set()` and `append()` and everything else.

- **The answer is that immutable types are safer from bugs, easier to understand, and more ready for change.** 不可变类型更"安全"，在其他质量指标上表现更好

  - Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts.

- **Tradeoff between <u>performance</u> and <u>safety</u>?** 折中，看你看重哪个质量指标

Risky example #1: passing mutable values

```java
/** @return the sum of the numbers in the list */
public static int sum(List<Integer> list) {
    int sum = 0;
    for (int x : list)
        sum += x;
    return sum;
}
```

Mutating the list directly for better performance

```java
/** @return the sum of the absolute values of the numbers in the list */
public static int sumAbsolute(List<Integer> list) {
    // let's reuse sum(), because DRY, so first we take absolute values
    for (int i = 0; i < list.size(); ++i)
        list.set(i, Math.abs(list.get(i)));
    return sum(list);
}
```

一个计算数组中元素之和的函数
注意这个函数的功能

```java
// meanwhile, somewhere else in the code...
public static void main(String[] args) {
    // ...
    List<Integer> myData = Arrays.asList(-5, -3, -2);
    System.out.println(sumAbsolute(myData));
    System.out.println(sum(myData));
}
```

But, what will happen here?

Risk

- **Safe from bugs?**
  - In this example, it's easy to blame the implementer of `sum-Absolute()` for going beyond what its spec allowed. 该函数超出了spec范畴，因为它改变了输入参数的值！
  - But really, passing mutable objects around is a latent bug. It's just waiting for some programmer to inadvertently mutate that list, often with very good intentions like reuse or performance, but resulting in a bug that may be very hard to track down. 这种错误非常难于跟踪和发现

- **Easy to understand?**
  - When reading `main()`, what would you assume about `sum()` and `sumAbsolute()`?
  - Is it clearly visible to the reader that `myData` gets changed by one of them?
  - 对其他程序员来说，也难以理解
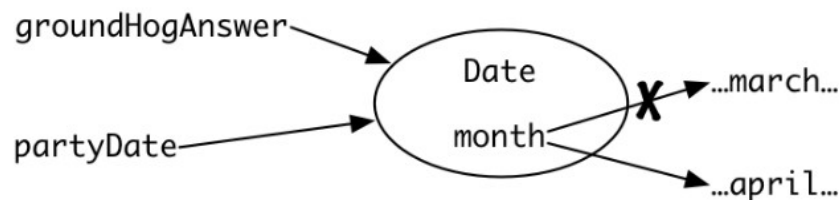
Risky example #2: returning mutable values

- ▪ Date **as a built-in Java class, is a mutable type.**

```java
/** @return the first day of spring this year */
public static Date startOfSpring() {
    return askGroundhog();
}
```

```java
// somewhere else in the code...
public static void partyPlanning() {
    Date partyDate = startOfSpring();
    // ...
}
```

通过全局变量作为
cache

```java
/** @return the first day of spring this year */
public static Date startOfSpring() {
    if (groundhogAnswer == null) groundhogAnswer = askGroundhog();
    return groundhogAnswer;
}

Date groundhogAnswer = null;

where else in the code...
static void partyPlanning() {
    // let's have a party one month after spring starts!
    Date partyDate = startOfSpring();
    partyDate.setMonth(partyDate.getMonth() + 1);
    // ... uh-oh. what just happened?
}
```

groundHogAnswer ──→ (Date / month) X ──→ ...march...
partyDate ──→ ...april...

What will happen here?

Risk

- **In both of these examples — the** `List<Integer>` **and the** `Date` **— the problems would have been completely avoided if the** `list` **and the** `date` **had been immutable types.**

- **The bugs would have been impossible by design.**

- **You should never use** `Date` **!**

  – Use one of the classes from package java.time : `LocalDateTime` , `Instant` , etc.

  – All guarantee in their specifications that they are immutable .

java.time

Class LocalDateTime

java.lang.Object
    java.time.LocalDateTime

All Implemented Interfaces:
Serializable, Comparable<ChronoLocalDateTime<?>>,

---

public final class **LocalDateTime**
extends Object
implements Temporal, TemporalAdjuster, ChronoLocal

A date-time without a time-zone in the ISO-8601 calendar s

LocalDateTime is an **immutable** date-time object that repr
week-of-year, can also be accessed. Time is represented to

This class does not store or represent a time-zone. Instead,
on the time-line without additional information such as an c

The ISO-8601 calendar system is the modern civil calendar
are applied for all time. For most applications written today
will find the ISO-8601 approach unsuitable.

This is a value-based class; use of identity-sensitive operati
results and should be avoided. The equals method should l

Implementation Requirements:
This class is **immutable** and thread-safe.

How to modify the code?

- **In Example 1:**
  - To return a new copy of the object (**defensive copying**), i.e.,

        return new Date(groundhogAnswer.getTime());

    通过防御式拷贝，给客户端返回一个全新的Date对象

  - However, use extra space for *every client* — even if 99% of the clients never mutate the date it returns. We may end up with lots of copies of the first day of spring throughout memory. 大部分时候该拷贝不会被客户端修改，可能造成大量的内存浪费

- **If we used an immutable type instead, then different parts of the program could safely share the same values in memory, so less copying and less memory space is required.** 如果使用不可变类型，则节省了频繁复制的代价

- **Immutability can be more efficient than mutability, because immutable types never need to be defensively copied.**
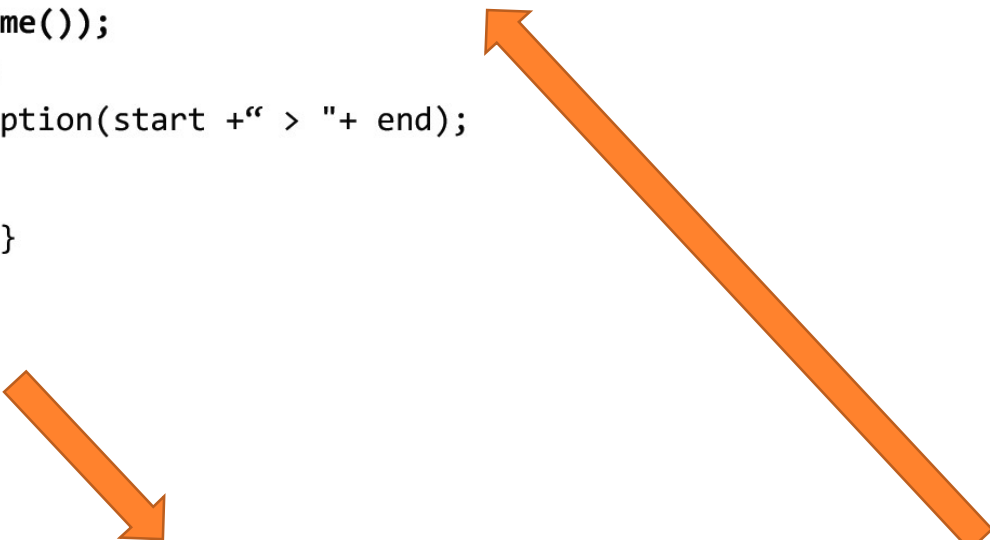
Aliasing is what makes mutable types risky

- Using mutable objects is just fine if you are using them entirely **locally** within a method, and with only one reference to the object. 安全的使用可变类型：局部变量，不会涉及共享；只有一个引用

- What led to the problem in the two examples we just looked at was having multiple references, also called **aliases**, for the same mutable object. 如果有多个引用（别名），使用可变类型就非常不安全

  - In the `List` example, the same list is pointed to by both list (in `sum` and `sumAbsolute`) and `myData` (in `main`). One programmer (`sumAbsolute`'s) thinks it's ok to modify the list; another programmer (`main`'s) wants the list to stay the same. Because of the aliases, `main`'s programmer loses.

  - In the `Date` example, there are two variable names that point to the `Date` object, `groundhogAnswer` and `partyDate` . These aliases are in completely different parts of the code, under the control of different programmers who may have no idea what the other is doing.

More Examples of Defensive Copying

```
public final class Period {
    private final Date start, end; // Invariant: start <= end

    /**
    // Repaired constructor - defensively copies parameters
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end   = new Date(end.getTime());
        if (this.start.after(this.end))
            throw new IllegalArgumentException(start +" > "+ end);
    }

    public Date start() { return start; }
    public Date end()   { return end; }
    ... // Remainder omitted
}
```
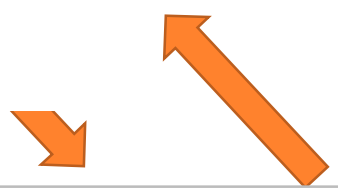
```
// Attack the internals of a Period instance
Date start = new Date();  // (The current time)
Date end   = new Date();  //    "       "       "
Period p = new Period(start, end);
end.setYear(78);    // Modifies internals of p!
```

More Examples of Defensive Copying

```java
public final class Period {
    private final Date start, end; // Invariant: start <= end

    /**
     * @throws IllegalArgumentException if start > end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.after(end))
            throw new IllegalArgumentException(start + " > " + end);
        this.start = start;
        this.end   = end;
    }

    // Repaired accessors - defensively copy fields
    public Date start() {
        return new Date(start.getTime());
    }
}   }
    public Date end() {
        return new Date(end.getTime());
    }
}
```
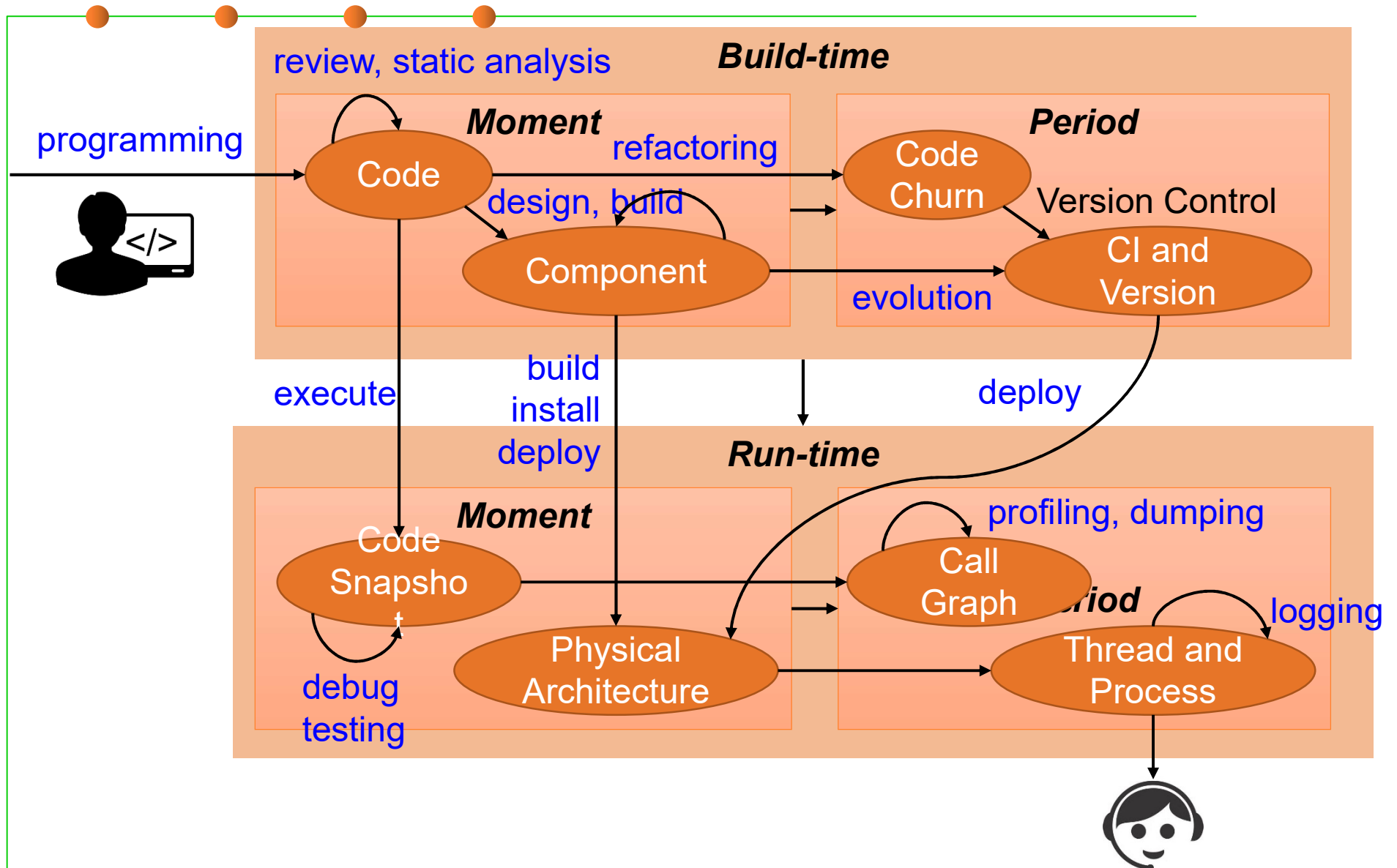
```java
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78);  // Modifies internals of p!
```

4 Snapshot diagram
as a code-level, run-time, and moment view

Software construction: transformation btw views

review, static analysis

Build-time

Moment

programming

Code

refactoring

Period

Code Churn

Version Control

design, build

Component

CI and Version

evolution

execute

build
install
deploy

Run-time

deploy

Moment

profiling, dumping

Code Snapshot

Call Graph

Period

logging

debug testing

Physical Architecture

Thread and Process

Snapshot diagrams

- It will be useful for us to draw pictures of what's happening at runtime, in order to understand subtle questions.

- **Snapshot diagrams** represent the internal state of a program at runtime – its *stack* (methods in progress and their local variables) and its *heap* (objects that currently exist). 用于描述程序运行时的内部状态

- **Why we use snapshot diagrams?**
  - To talk to each other through pictures. 便于程序员之间的交流
  - To illustrate concepts like primitive types vs. object types, immutable values vs. immutable references, pointer aliasing, stack vs. heap, abstractions vs. concrete representations. 便于刻画各类变量随时间变化
  - To help explain your design for your team project (with each other and with your TA). 便于解释设计思路
  - To pave the way for richer design notations in subsequent courses.

Primitive and Object values in Snapshot Diagram

- **Primitive values** 基本类型的值

  - Primitive values are represented by bare constants. The incoming arrow is a reference to the value from a variable or an object field.

- **Object values** 对象类型的值

  3 5.0 'c' null

  - An object value is a circle labeled by its type.

  - When we want to show more detail, we write field names inside it, with arrows pointing out to their values. For still more detail, the fields can include their declared types.
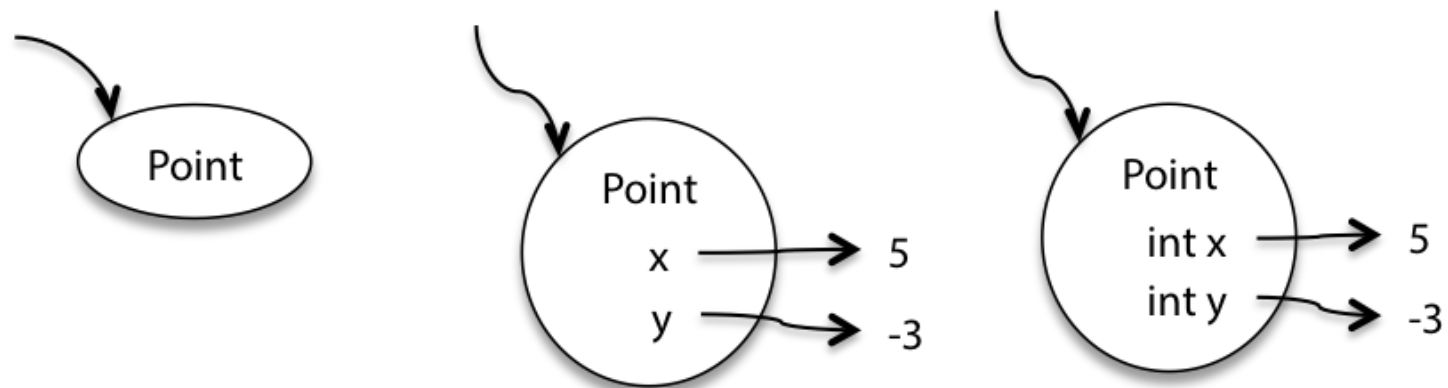
Reassignment and immutable values

- **For example, if we have a** `String` **variable** `s` **, we can reassign it from a value of** `"a"` **to** `"ab"`

  ```
  String s = "a";

  s = s + "b";
  ```

- `String` **is an example of an immutable type, a type whose values can never change once they have been created.**

- **Immutable objects (intended by their designer to always represent the same value) are denoted in a snapshot diagram by a** **double border****, like the** `String` **objects in our diagram.** 不可变对象：用双线椭圆
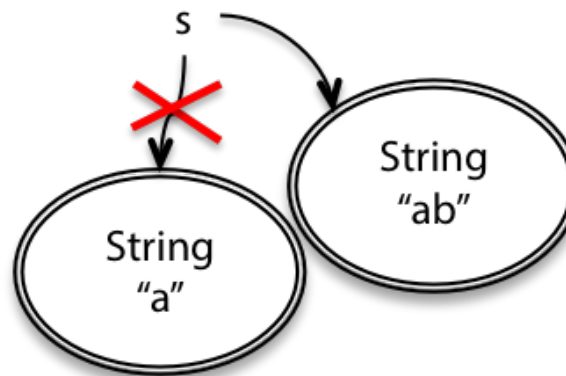
Mutable values

- **By contrast,** `StringBuilder` **(a built-in Java class) is a mutable object that represents a string of characters, and it has methods that change the value of the object:**

```
StringBuilder sb = new StringBuilder("a");

sb.append("b");
```

- **These two snapshot diagrams look very different, which is good: the difference between mutability and immutability will play an important role in making code safe from bugs .**
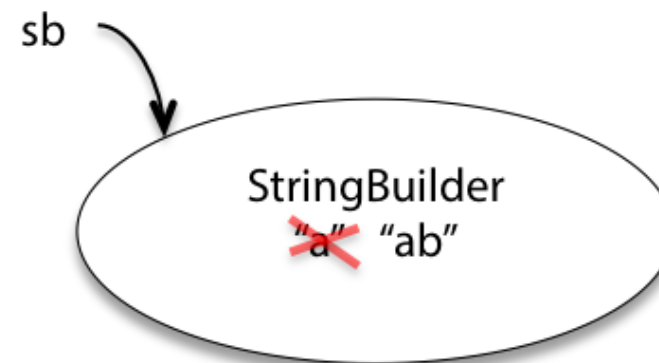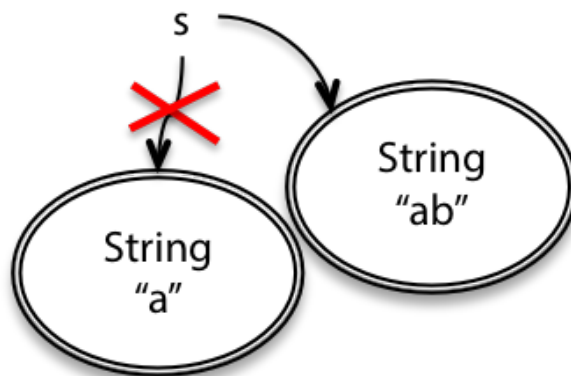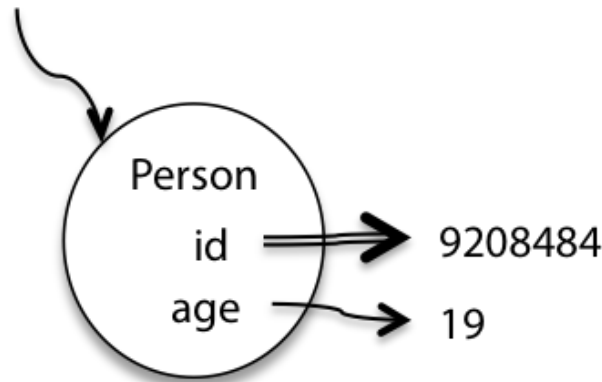
Unreassignable/Immutable references

- **Java also gives us immutable references 不可变的引用: variables that are assigned once and never reassigned. To make a reference immutable, declare it with the keyword `final`:**

  ```
  final int n = 5;
  ```

- **If the Java compiler isn't convinced that your `final` variable will only be assigned once at runtime, then it will produce a compiler error. So `final` gives a static checking for immutable references.**

- **In a snapshot diagram, an unreassignable reference (`final`) is denoted by a double arrow.**

Unreassignable/Immutable references

- **An object whose** `id` **never changes (it can't be reassigned to a different number), but whose** `age` **can change.** 不可变的引用：用双线箭头

- **An immutable/unreassignable reference to a mutable value** **(for example:** `final StringBuilder sb`**) whose value can change even though we're pointing to the same object.** 引用是不可变的，但指向的值却可以是可变的

- **A mutable/reassignable reference to an immutable value** **(like** `String s`**), where the value of the variable can change because it can be re-pointed to a different object.** 可变的引用，也可指向不可变的值

Exercises

- **We can have an immutable reference to a mutable value (for example:** `final StringBuilder sb`**) whose value can change even though we're pointing to the same object.**
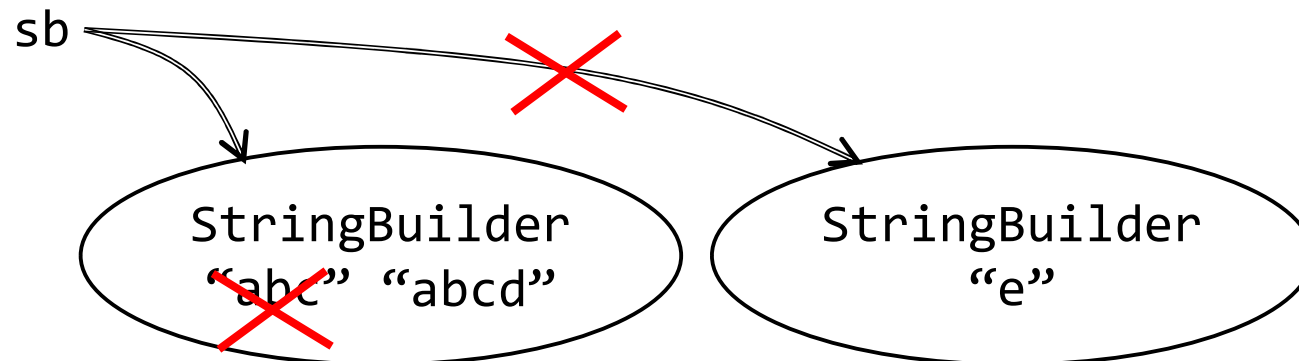
```
final StringBuilder sb = new StringBuilder("abc");
sb.append("d");
sb = new StringBuilder("e");
System.out.println(sb);
```

编译阶段出错：The final variable sb cannot be assigned

输出：abcd

What will happen?
Can you draw the snapshot diagram?

Exercises

- **We can also have a mutable reference to an immutable value (like `String s`), where the value of the variable can change because it can be re-pointed to a different object.**

```
String s1 = new String("abc");
List<String> list = new ArrayList<String>();
list.add(s1);

s1 = s1.concat("d");
System.out.println(list.get(0));

String s2 = s1.concat("e");
list.set(0, s2);
System.out.println(list.get(0));
```

输出：abc

输出：abcde

What will happen?
Can you draw the snapshot diagram?

```
String s1 = new String("abc");
List<String> list = new ArrayList<String>();
list.add(s1);

s1 = s1.concat("d");
System.out.println(list.get(0));

String s2 = s1.concat("e");
list.set(0, s2);
System.out.println(list.get(0));
```
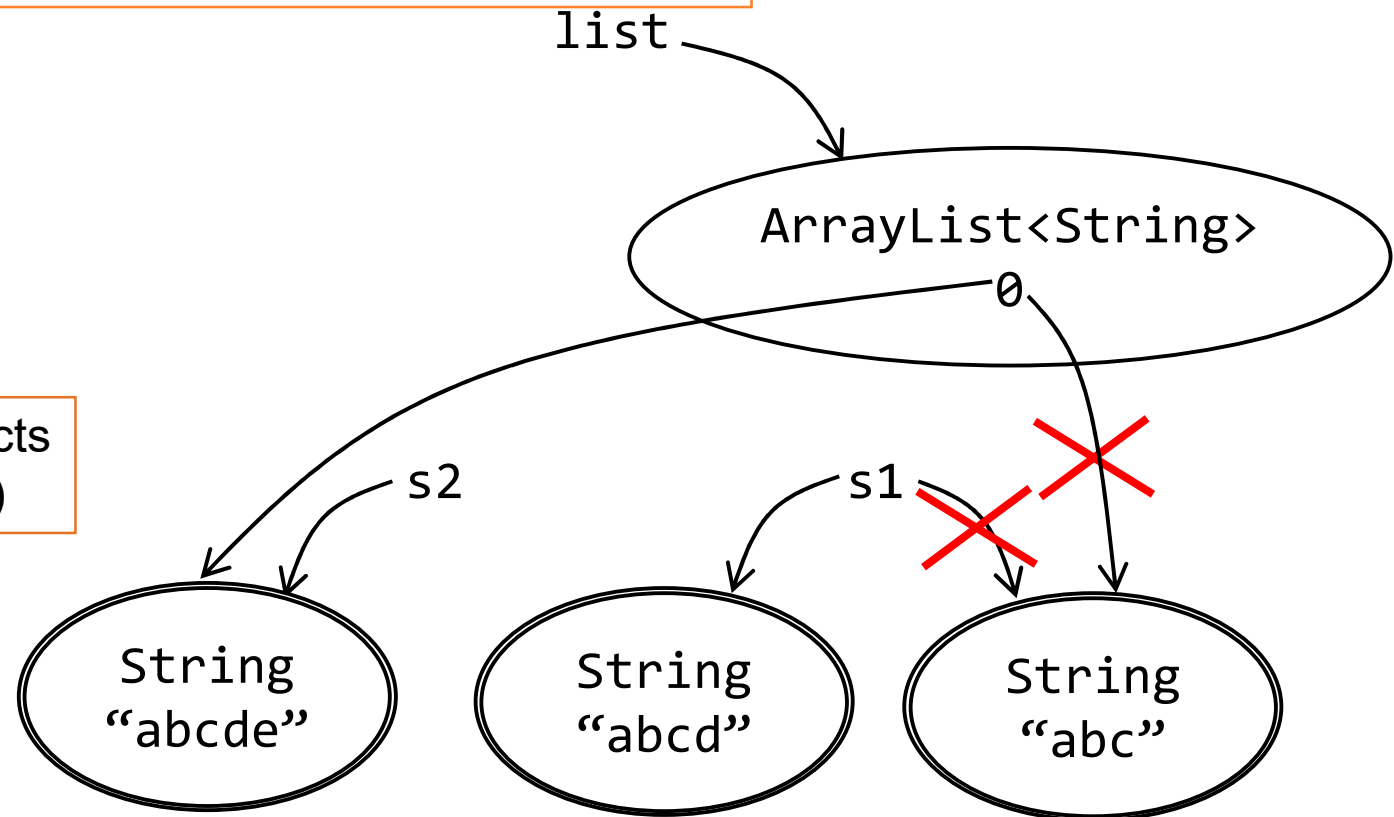
输出：abc

输出：abcde

list

ArrayList<String>
0

Unreachable objects
等待垃圾回收(GC)

s2

s1

String
"abcde"

String
"abcd"

String
"abc"

5 Complex data types: Arrays and Collections

Array

- **Arrays are fixed-length sequences of another type** `T`**. For example, here's how to declare an array variable and construct an array value to assign to it:**

  ```
  int[] a = new int[100];
  ```

- **The** `int[]` **array type includes all possible array values, but a particular array value, once created, can never change its length.**

- **Operations on array types include:**

  - indexing: `a[2]`

  - assignment: `a[2]=0`

  - length: `a.length`

Why 100?

```
int[] a = new int[100];
int i = 0;
int n = 3;
while (n != 1) {
    a[i] = n;
    i++;
    if (n % 2 == 0) {
        n = n / 2;
    }
    else {
        n = 3 * n + 1;
    }
}
a[i] = n;
i++;
```

List

- **Lists are variable-length sequences of another type** `T` **.**

    ```
    List<Integer> list = new ArrayList<Integer>();
    ```
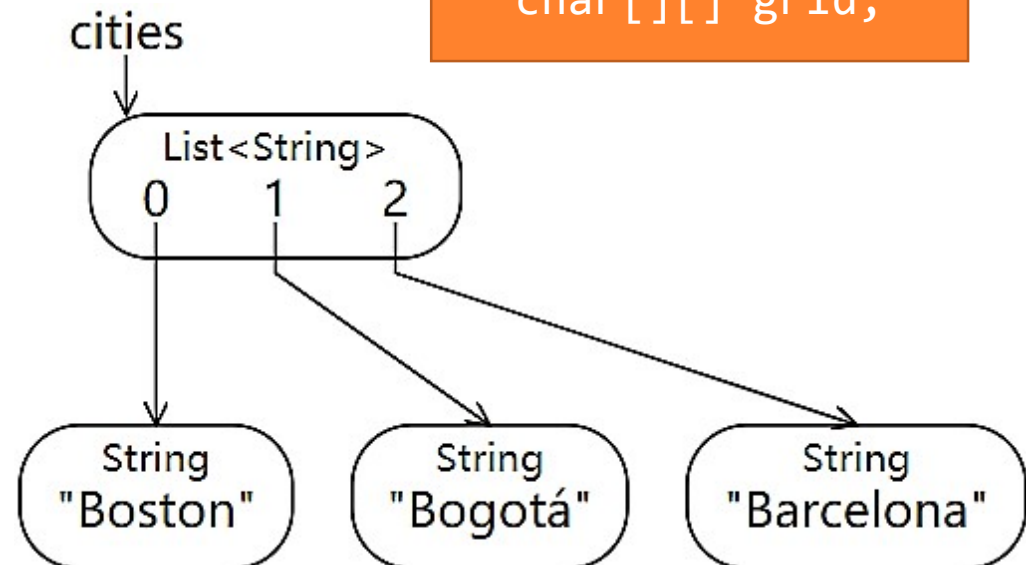
- **Some of its operations:**

    Please use List rather than arrays

    - indexing: `list.get(2)`

    - assignment: `list.set(2, 0)`

        ```
        String[] names;
        int[] numbers;
        char[][] grid;
        ```

    - length: `list.size()`

- **Note 1:** `List` **is an interface.**

- **Note 2: members in a** `List` **must be an object.**

cities

List<String>
0   1   2

String
"Boston"

String
"Bogotá"

String
"Barcelona"

Iterating

- **Iterating an array**

```
int max = 0;

for (int i=0; i<array.length; i++) {

    max = Math.max(array[i], max);

}
```

- **Iterating a** `List`

```
int max = 0;

for (int x : list) {

    max = Math.max(x, max);

}
```

Set

- **A `Set` is an unordered collection of zero or more unique objects.**

- An object cannot appear in a set multiple times. Either it's in or it's out.
  - `s1.contains(e)` test if the set contains an element
  - `s1.containsAll(s2)` test whether s1 ⊆ s2
  - `s1.removeAll(s2)` remove s2 from s1

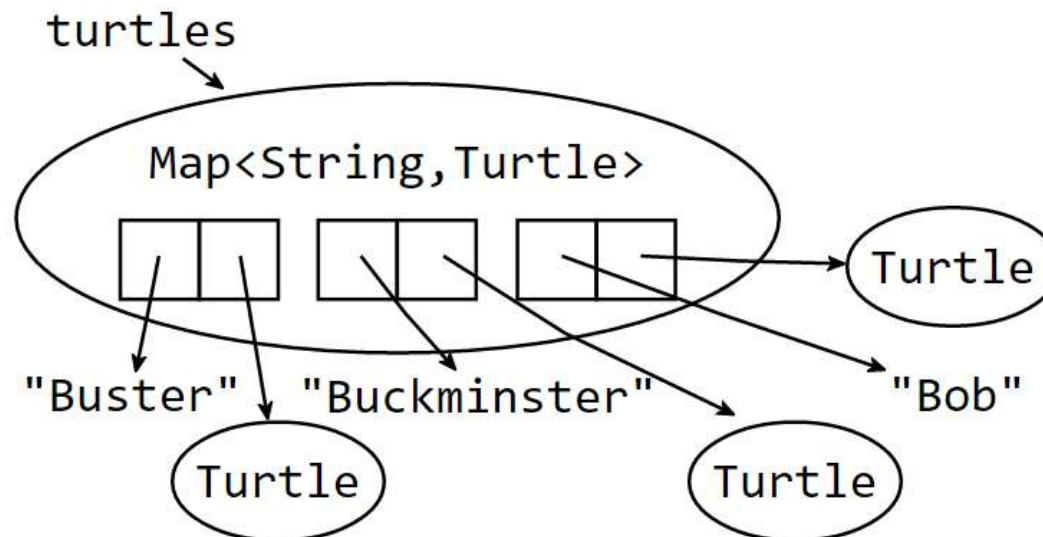- **`Set` is an abstract interface**
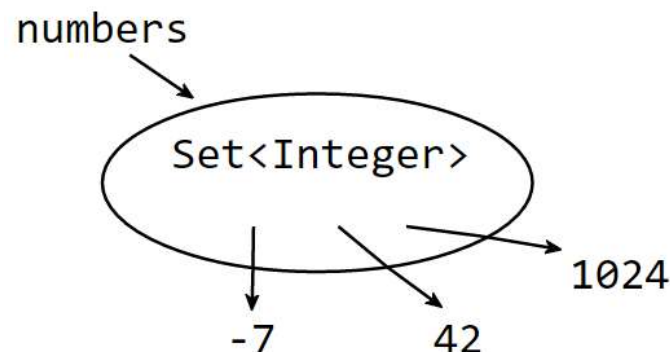
Map

- **A `Map` is similar to a dictionary (key-value)**
    - `map.put(key, val)`   add the mapping key → val
    - `map.get(key)`   get the value for a key
    - `map.containsKey(key)`   test whether the map has a key
    - `map.remove(key)`   delete a mapping
- **`Map` is an abstract interface**

Declaring `List, Set, and Map` variables

- **With Java collections we can restrict the type of objects contained in the collection.**

- **When we add an item, the compiler can perform *static checking* to ensure we only add items of the appropriate type.**

- **Then, when we pull out an item, we are guaranteed that its type will be what we expect.**

Declaring `List,` `Set,` and `Map` variables

- **Declaration:**

  ```
  List<String> cities;                // a List of Strings

  Set<Integer> numbers;               // a Set of Integers

  Map<String, Turtle> turtles;        // a Map with String keys and Turtle values
  ```

- **We cannot create a collection of primitive types.**

  - For example, `Set<int>` does not work.

  - However, `int` have an Integer wrapper we can use (e.g. `Set<Integer> numbers`).

  - When using:

    ```
    sequence.add(5);                   // add 5 to the sequence
    int second = sequence.get(1);      // get the second element
    ```

Creating `List`, `Set`, and `Map` variables

- **Java helps distinguish between**
  - the specification of a type – what does it do?   Abstract Interface
  - The implementation – what is the code?   Concrete Class

- `List`, `Set`, **and** `Map` **are all interfaces:**
  - They define how these respective types work, but they don't provide implementation code.
  - Advantage: users have the right to choose different implementations in different situations.

- **Implementations of** `List`, `Set`, **and** `Map` **:**
  - List: `ArrayList` and `LinkedList`
  - Set: `HashSet`
  - Map: `HashMap`

```
List<String> firstNames = new ArrayList<String>();
List<String> lastNames = new LinkedList<String>();

List<String> firstNames = new ArrayList<>();
List<String> lastNames = new LinkedList<>();

Set<Integer> numbers = new HashSet<>();

Map<String,Turtle> turtles = new HashMap<>();
```

Using List, Set, and Map variables

```
Map<String, Double> treasures = new HashMap<>();
String x = "palm";

treasures.put("beach", 25.);
treasures.put("palm", 50.);
treasures.put("cove", 75.);
treasures.put("x", 100.);
treasures.put("palm", treasures.get("palm") + treasures.size());

treasures.remove("beach");

double found = 0;
for (double treasure : treasures.values()) {
    found += treasure;
}
```
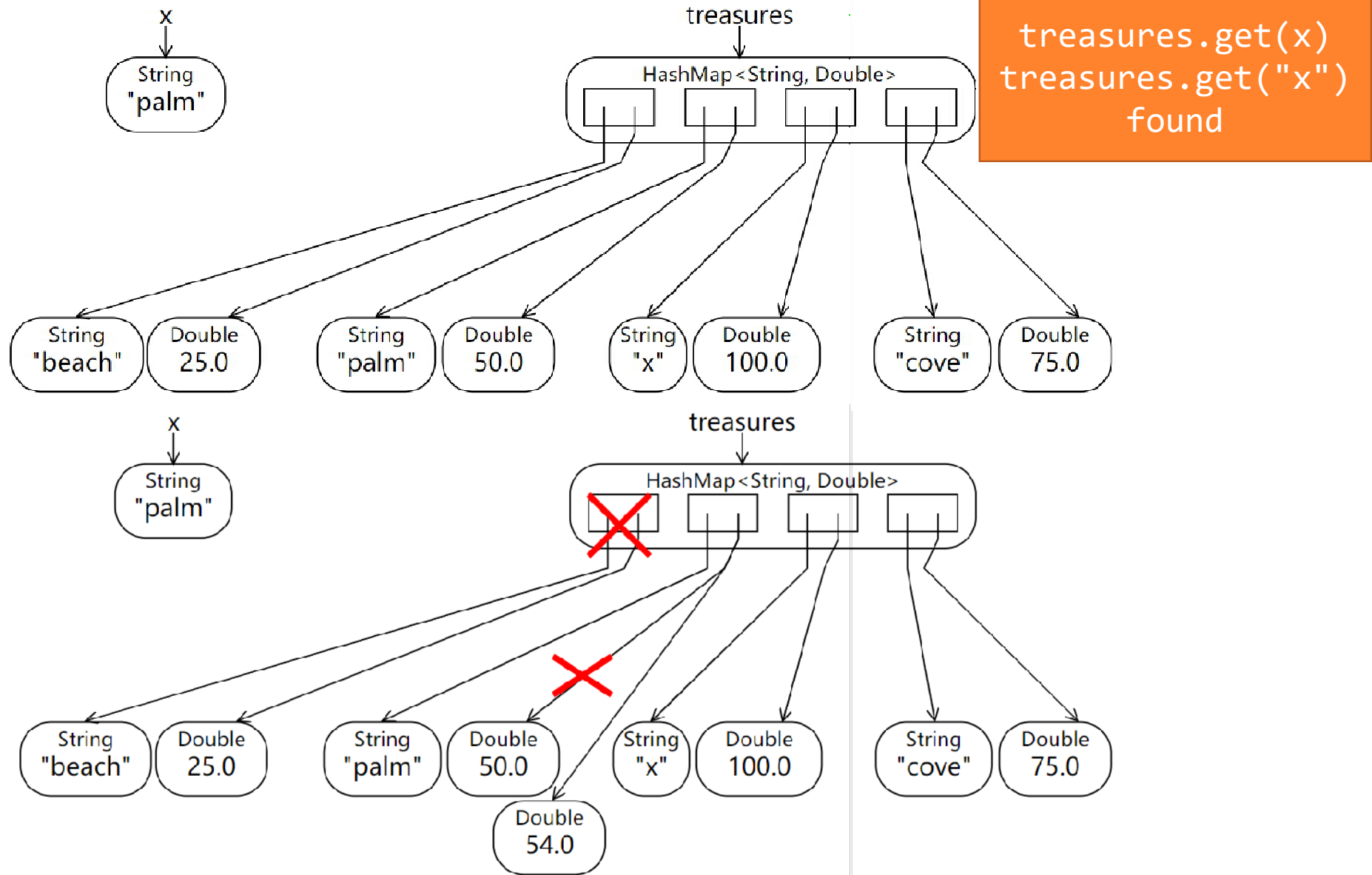
```
treasures.get(x)
treasures.get("x")
found
```

Using `List`, `Set`, and `Map` variables

x

String
"palm"

treasures

HashMap<String, Double>

```
treasures.get(x)
treasures.get("x")
found
```

String
"beach"

Double
25.0

String
"palm"

Double
50.0

String
"x"

Double
100.0

String
"cove"

Double
75.0

x

String
"palm"

treasures

HashMap<String, Double>

String
"beach"

Double
25.0

String
"palm"

Double
50.0

String
"x"

Double
100.0

String
"cove"

Double
75.0

Double
54.0

Iteration

```java
List<String> cities           = new ArrayList<>();
Set<Integer> numbers          = new HashSet<>();
Map<String,Turtle> turtles    = new HashMap<>();

for (String city : cities) {
    System.out.println(city);
}

for (int num : numbers) {
    System.out.println(num);
}

for (int ii = 0; ii < cities.size(); ii++) {
    System.out.println(cities.get(ii));
}

for (String key : turtles.keySet()) {
    System.out.println(key + ": " + turtles.get(key));
}
```

Iterator as a mutable type 迭代器

- `Iterator` **is an object that steps through a collection of elements and returns the elements one by one.**

- **Iterators are used under the covers in Java when you're using a** `for (... : ...)` **loop to step through a** `List` **or array.**

- **An** `iterator` **has two methods:**

  – `next()` returns the next element in the collection --- this is a **mutator** method!

  – `hasNext()` tests whether the iterator has reached the end of the collection.

```
List<String> lst = ...;
for (String str : lst) {
    System.out.println(str);
}
```

```
List<String> lst = ...;
Iterator iter = lst.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    System.out.println(str);
}
```

An example iterator for `ArrayList<String>`

```java
/**
 * A MyIterator is a mutable object that iterates over
 * the elements of an ArrayList<String>  from first to last.
 * This is just an example          tor works.
 * In practice, you shoul                 n iterator
 * object, returned by it
 */
public class MyIterator {

    private final ArrayList<String> list;
    private int index;
    // list[index] is the next element that will be returned
    //
    //                        more e

    /**
     * Make an iterator.
     * @param list list to iterate over
     */
    public MyIterator(ArrayList<String> list) {
        this.list = list;
        this.index = 0;
    }
}
```

Class definition

Instance variables

Specification

Invariants

Constructor

```java
/**
 * Test whether the iterator has more elements to return.
 * @return true if next() will return another element,
 *         false if all elements have been returned
 */
public boolean hasNext() {
    return index < list.size();
}

/**
 * Get the next element of the list.
 * Requires: hasNext() returns true.
 * Modifies: this iterator to advance it to the element
 *           following the returned element.
 * @return next element of the list
 */
public String next() {
    final String element = list.get(index);
    ++index;
    return element;
}
```
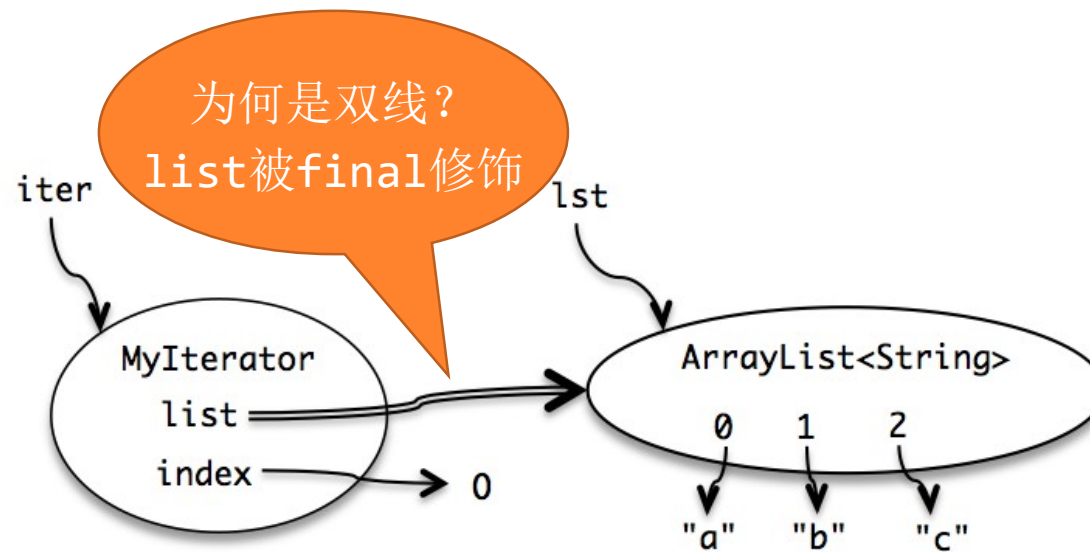
Instance method

改变了index的取值
故该类是mutable的

Snapshot diagram of `MyIterator`

Mutation undermines (暗中破坏) an iterator

- **Suppose we have a list of subjects represented as strings. We want a method `dropCourse6` that will delete the Course 6 subjects from the list, leaving the other subjects behind.**

- **First write the specification (to be introduced in next lecture)**

```
/**
 * Drop all subjects that are from Course 6.
 * Modifies subjects list by removing subjects that start with "6."
 *
 * @param subjects list of MIT subject numbers
 */
public static void dropCourse6(ArrayList<String> subjects)
```

- **Next, write the test cases**

```
// Testing strategy:
//    subjects.size: 0, 1, n
//    contents: no 6.xx, one 6.xx, all 6.xx
//    position: 6.xx at start, 6.xx in middle, 6.xx at end

// Test cases:
//    [] => []
//    ["8.03"] => ["8.03"]
//    ["14.03", "9.00", "21L.005"] => ["14.03", "9.00", "21L.005"]
//    ["2.001", "6.01", "18.03"] => ["2.001", "18.03"]
//    ["6.045", "6.005", "6.813"] => []
```

Mutation undermines an iterator

- **The implementation:**

```java
public static void dropCourse6(ArrayList<String> subjects) {
    MyIterator iter = new MyIterator(subjects);
    while (iter.hasNext()) {
        String subject = iter.next();
        if (subject.startsWith("6.")) {
            subjects.remove(subject);
        }
    }
}
```
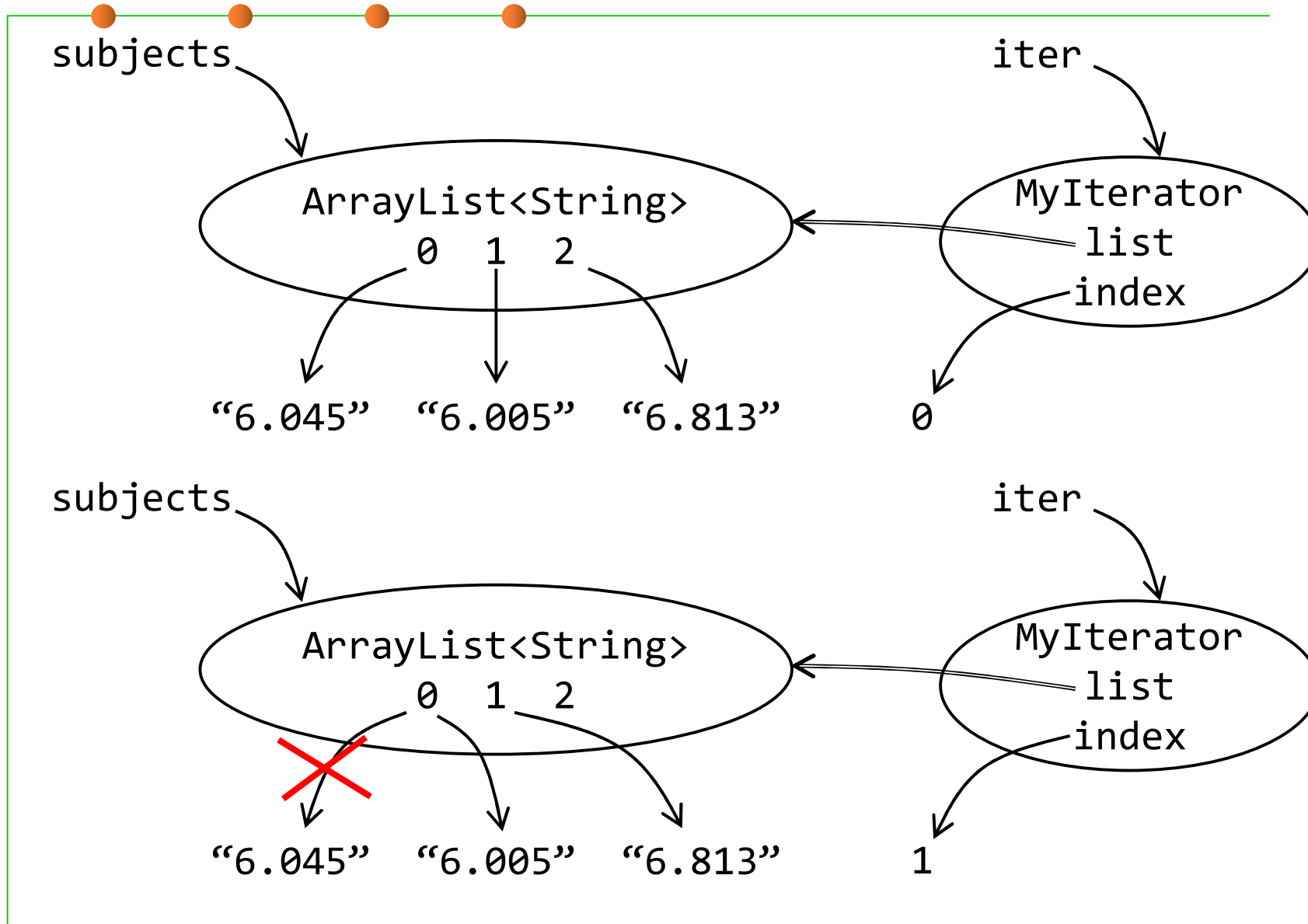
- **Run it**

```java
// dropCourse6(["6.045", "6.005", "6.813"])
//    expected [], actual ["6.005"]
```
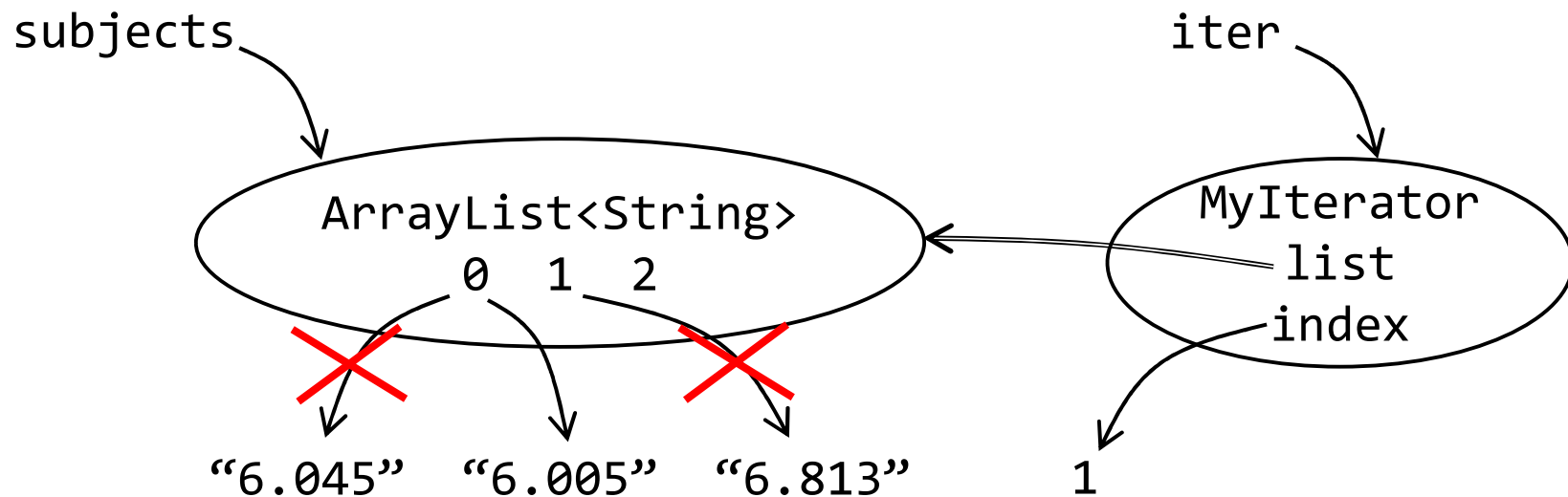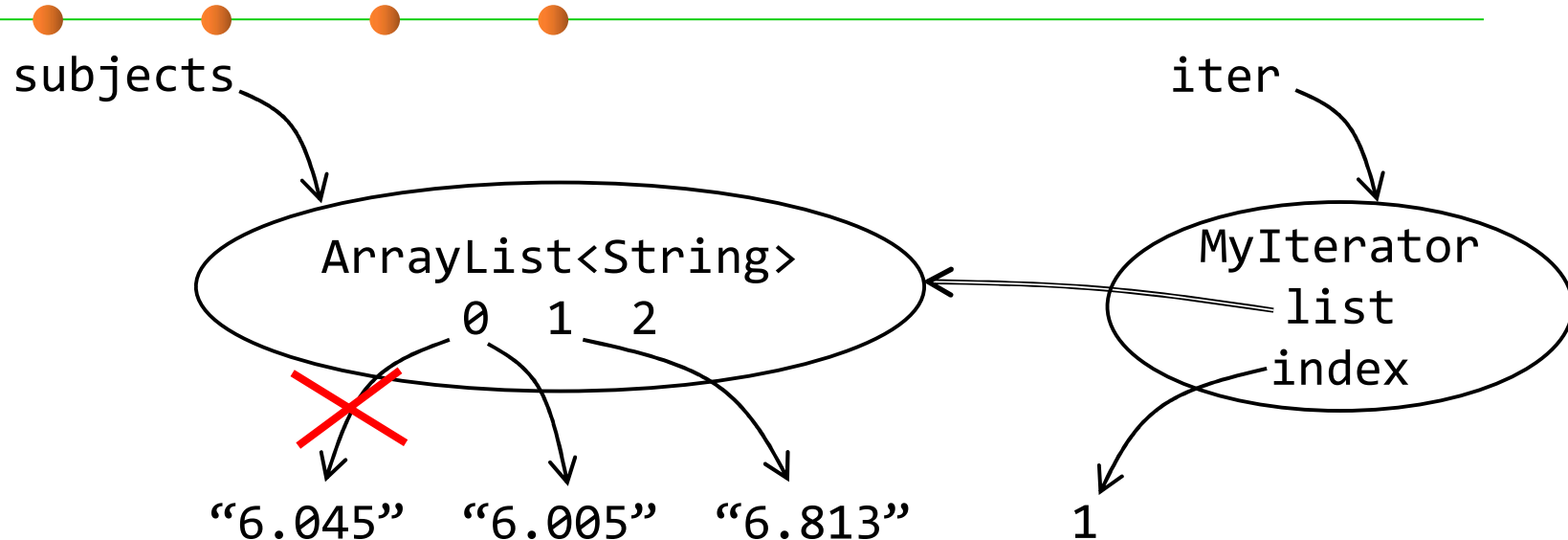
Why?
Draw the snapshot
and analysis…

Mutation undermines an iterator

Mutation undermines an iterator

Mutation undermines an iterator

- **How about this code?**

```java
for (String subject : subjects) {
    if (subject.startsWith("6.")) {
        subjects.remove(subject);
    }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
        at java.util.ArrayList$Itr.next(Unknown Source)
        at Immutable.main(Immutable.java:23)
```

- **Try this:**

```java
Iterator iter = subjects.iterator();
while (iter.hasNext()) {
    String subject = iter.next();
    if (subject.startsWith("6.")) {
        iter.remove();
    }
}
```

The iterator adjusts its index appropriately.

6 Useful immutable types

Useful immutable types

- **The primitive types and primitive wrappers are all immutable.** 基本类型及其封装对象类型都是不可变的

    - If you need to compute with large numbers, `BigInteger` and `BigDecimal` are immutable.

- **Don't use mutable** `Date`**, use the appropriate immutable type from** `java.time` **based on the granularity of timekeeping you need.**

- **The usual implementations of Java's collections types —** `List`**,** `Set`**,** `Map` **— are all mutable:** `ArrayList`**,** `HashMap`**, etc.**

- **The** `Collections` **utility class has methods for obtaining unmodifiable views of these mutable collections:**

    - `Collections.unmodifiableList`
    - `Collections.unmodifiableSet`
    - `Collections.unmodifiableMap`

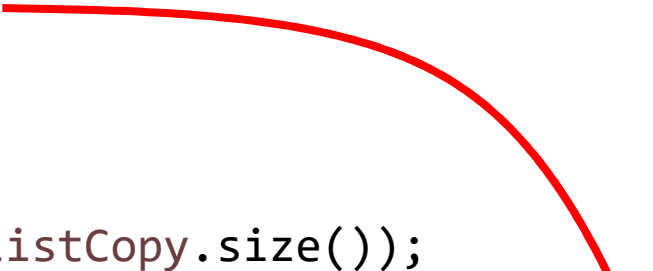   a wrapper around the underlying list/set/map

Immutable Wrappers around Mutable Data Types

- **The Java collections classes offer an interesting compromise: immutable wrappers.** 这种包装器得到的结果是不可变的：只能看

  - Collections.unmodifiableList() takes a (mutable) List and wraps it with an object that looks like a List , but whose mutators are disabled – set() , add() , remove() , etc. throw exceptions. So you can construct a list using mutators, then seal it up in an unmodifiable wrapper (and throw away your reference to the original mutable list, and get an immutable list.

- **The downside is that you get immutability at runtime, but not at compile time.** 但是这种"不可变"是在运行阶段获得的，编译阶段无法据此进行静态检查

  - Java won't warn you at compile time if you try to sort() this unmodifiable list.

  - You'll just get an exception at runtime.

  - But that's still better than nothing, so using unmodifiable lists, maps, and sets can be a very good way to reduce the risk of bugs.

Unmodifiable Wrappers

What will happen?

```java
List<String> list = new ArrayList<>();

list.add("ab");

List<String> listCopy = Collections.unmodifiableList(list);

listCopy.add("c");

list.add("c");

System.out.println(listCopy.size());
```

2

```
Exception in thread "main" java.lang.UnsupportedOperationException
at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
at Immutable.main(Immutable.java:33)
```

What will happen?

```java
public class Zoo {
    private List<String> animals;
    public Zoo(List<String> animals) {
        this.animals = animals;
    }
    public List<String> getAnimals() {
        return this.animals;
    }
}
```

```java
List<String> b = zoo.getAnimals();
b.add("flamingo");
System.out.println(a);
```

What is the output from this code?

```java
List<String> a = new ArrayList<>();
a.addAll(Arrays.asList("lion", "tiger", "bear"));
Zoo zoo = new Zoo(a);
a.add("zebra");
System.out.println(a);
System.out.println(zoo.getAnimals());
```

Unmodifiable Wrappers

- **The unmodifiable wrappers take away the ability to modify the collection by intercepting all the operations that would modify the collection and throwing an** `UnsupportedOperationException`.

- **Unmodifiable wrappers have two main uses, as follows:**

  - To make a collection immutable once it has been built. In this case, it's good practice not to maintain a reference to the backing collection. This absolutely guarantees immutability.

  - To allow certain clients read-only access to your data structures. You keep a reference to the backing collection but hand out a reference to the wrapper. In this way, clients can look but not modify, while you maintain full access.

Unmodifiable Wrappers

- `public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);`

- `public static <T> Set<T> unmodifiableSet(Set<? extends T> s);`

- `public static <T> List<T> unmodifiableList(List<? extends T> list);`

- `public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);`

- `public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<? extends T> s);`

- `public static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> m);`

Static methods for unmodifiable collections

- **To create immutable collections from some known values, use** `List.of`, `Set.of`, **and** `Map.of`.

  ```
  List<String> a = List.of("lion", "tiger", "bear");
  ```

- **Alternatively, use** `List.copyOf` **(Java 10) to create unmodifiable shallow copies of mutable collections.**

Summary

Summary of this lecture

- **Static type checking:**

  – Safe from bugs. Static checking helps with safety by catching type errors and other bugs before runtime.

  – Easy to understand. It helps with understanding, because types are explicitly stated in the code.

  – Ready for change. Static checking makes it easier to change your code by identifying other places that need to change in tandem. For example, when you change the name or type of a variable, the compiler immediately displays errors at all the places where that variable is used, reminding you to update them as well.

Summary

- **Mutability** is useful for **performance** and **convenience**, but it also creates **risks of bugs** by requiring the code that uses the objects to be well-behaved on a global level, greatly complicating the reasoning and testing we have to do to be confident in its **correctness**.

- **Make sure you understand the difference between an immutable object (like a** `String` **) and an immutable reference (like a** `final` **variable).**

- **Snapshot diagrams can help with this understanding.**

  - Objects are values, represented by circles in a snapshot diagram, and an immutable one has a double border indicating that it never changes its value.

  - A reference is a pointer to an object, represented by an arrow in the snapshot diagram, and an immutable reference is an arrow with a double line, indicating that the arrow can't be moved to point to a different object.

Summary

- The key design principle is **immutability**: **using immutable objects and immutable references as much as possible**.

  - Safe from bugs. Immutable objects aren't susceptible to bugs caused by aliasing. Immutable references always point to the same object.

  - Easy to understand. Because an immutable object or reference always means the same thing, it's simpler for a reader of the code to reason about — they don't have to trace through all the code to find all the places where the object or reference might be changed, because it can't be changed.

  - Ready for change. If an object or reference can't be changed at runtime, then code that depends on that object or reference won't have to be revised when the program changes.

The end

May 20, 2021