




# Chapter 3: Abstract Data Type (ADT) and Object-Oriented Programming (OOP)

## 3.1 Data Type and Type Checking


Ming Liu

March 7, 2018

# Outline

- 
1. Data type in programming languages
  2. Static vs. dynamic data types
  3. Type checking
  4. Mutability & Immutability
  5. Snapshot diagram
  6. Complex data types: Arrays and Collections
  7. Useful immutable types
  8. Null references
  9. Summary

# Objective of this lecture

- 
1. Get to know basic knowledge about data type, and static and dynamic type checking in programming language, especially Java.
  2. Understand mutability and mutable objects
  3. Identify aliasing and understand the dangers of mutability
  4. Use immutability to improve correctness, clarity and changeability
  5. Use snapshot diagram to demonstrate the state of specific time during a program's execution.
  6. Use Arrays and Collections to deal with complex data types
  7. Know the harm of Null references and avoid it



# 1 Data type in programming languages



# Types and Variables

- A **type** is a set of values, along with operations that can be performed on those values.
- **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;`

# Types in Java

- Java has several **primitive types** , among them:
  - 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**, 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

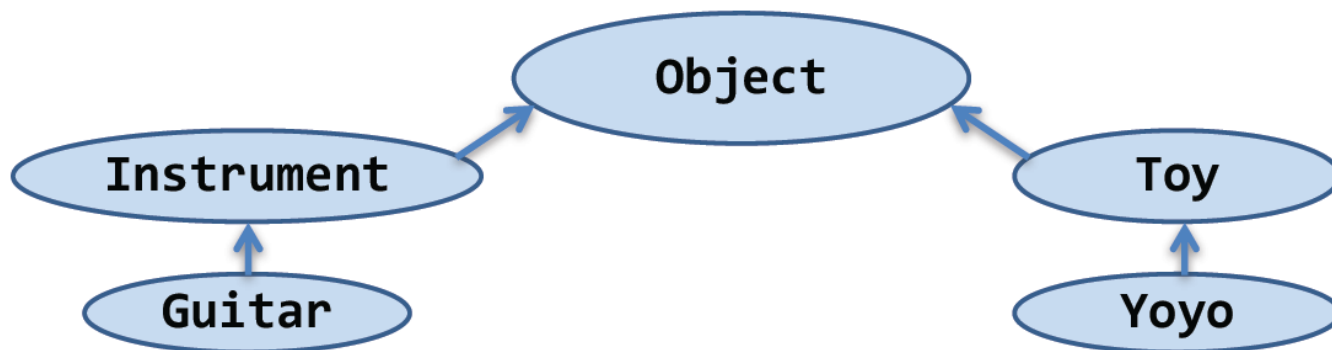
Primitives	Object Reference Types
int, long, byte, short, char, float, double, boolean	Classes, interfaces, arrays, enums, annotations
No identity except their value	Have identity distinct from value
<b>Immutable</b>	Some <b>mutable</b> , some not
On stack, exist only when in use	On heap, garbage collected
Can't achieve unity of expression	Unity of expression with generics
Dirt cheap	More costly

# Hierarchy of object types

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

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

- If **extends** clause omitted, defaults to Object
- A class is an instance of all its superclasses
  - Inherits visible fields and methods from its superclasses
  - Can override methods to change their behavior





# Boxed primitives

- **Immutable containers for primitive types**
  - Boolean, Integer, Short, Long, Character, Float, Double
- **Don't use boxed primitives unless you have to!**
- **Language does autoboxing and auto-unboxing**

# Operators

- **Operators: symbols that perform simple computations**

- Assignment: =
- Addition: +
- Subtraction: -
- Multiplication: \*
- Division: /

- **Order of Operations: follows standard math rules:**

- 1. Parentheses
- 2. Multiplication and division
- 3. Addition and subtraction

- **String concatenation (+)**

- `String text = "hello" + " world";`
- `text = text + " number " + 5;     // text = "hello world number 5"`

# Operations

- **Operations** are functions that take inputs and produce outputs (and sometimes change the values themselves).
  - As an infix, prefix, or postfix **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

- Some operations are overloaded in the sense that the same operation name is used for different types.
- 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 types



# Static Typing vs. Dynamic Typing

- **Java is a statically-typed language.**
  - 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.
- **In dynamically-typed languages like Python, this kind of checking is deferred until runtime (while the program is running).**



# 3 Type checking



# 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.
  - **Dynamic checking:** the bug is found automatically when the code is executed.
  - **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.
- **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.**

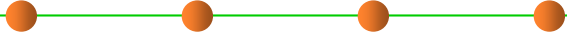


# Mismatched Types

- **Java verifies that types always match:**
- `String five = 5; // ERROR!`

```
test.java.2: incompatible types  
found: int  
required: java.lang.String  
String five = 5;
```

# Conversion by casting



```
int a = 2;                // a = 2
double a = 2;             // a = 2.0 (Implicit)
int a = 18.7;             // ERROR
int a = (int)18.7;        // a = 18
double a = 2/3;           // a = 0.0
double a = (double)2/3;   // a = 0.6666...
```

# Static checking

- Static checking means checking for bugs at compile time.
- Bugs are the bane 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.

# Static checking

- **Syntax errors**, like extra punctuation or spurious words. Even dynamically-typed languages like Python do this kind of static checking.
- **Wrong names**, like `Math.sine(2)` . (The right name is `sin`)
- **Wrong number of arguments**, like `Math.sin(30, 20)` .
- **Wrong argument types**, like `Math.sin("30")` .
- **Wrong return types**, like `return "30";` from a function that's declared to return an `int` .

# Dynamic checking

- **Illegal argument values.** 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**, i.e., when the specific return value can't be represented in the type.
- **Out-of-range indexes**, e.g., using a negative or too-large index on a string.
- **Calling a method on a null object reference.**

# 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*.**

# Primitive Types Are Not True Numbers

- One trap in Java – and many other programming languages – is that its primitive numeric types have corner cases that do not behave like the integers and real numbers we’re used to.
- As a result, some errors that really should be dynamically checked are not checked at all.
  - Integer division:  $5/2$  does not return a fraction, it returns a truncated integer.
  - Integer overflow. If the computation result is too positive or too negative to fit in that finite range, it quietly *overflows* and returns a wrong answer. (no static / dynamic checking!) e.g., `int big = 200000*200000;`
  - Special values in floating-point types. NaN (“Not a Number”), `POSITIVE_INFINITY`, and `NEGATIVE_INFINITY`. e.g., `double a=7/0;`



# 4 Mutability and Immutability





# Assignment

- 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?**
  - When you assign to a variable, you're changing where the variable's arrow points. You can point it to a different value.
  - When you assign to the contents of a mutable value – such as an array or list – you're changing references inside that value.
- **Change is a necessary evil.**
- **Good programmers avoid things that change, because they may change unexpectedly.**

# Immutability

- Immutability is a major design principle.
- Immutable types are types whose values can never change once they have been created.
  - Java also gives us immutable references: 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 you static checking for immutable references.

# Immutability

- It's good practice to use **final** for declaring the parameters of a method and as many local variables as possible.
- Like the type of the variable, these declarations are important documentation, useful to the reader of the code and statically checked by the compiler.
- **Note:**
  - A **final** class declaration means it cannot be **inherited**.
  - A **final** variable means it always contains the same value/reference but cannot be **changed**;
  - A **final** method means it cannot be **overridden** by subclasses;

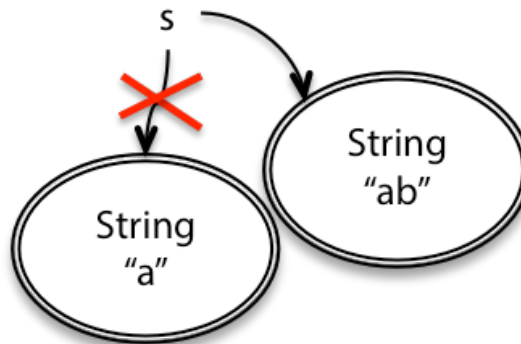
# Mutability and Immutability

- 
- **Objects are immutable:** once created, they always represent the same value.
  - **Objects are mutable:** they have methods that change the value of the object.

# 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
```

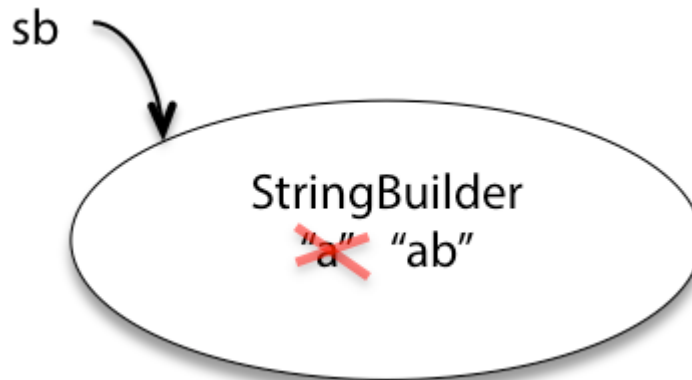


Note: this is a snapshot diagram

# 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:

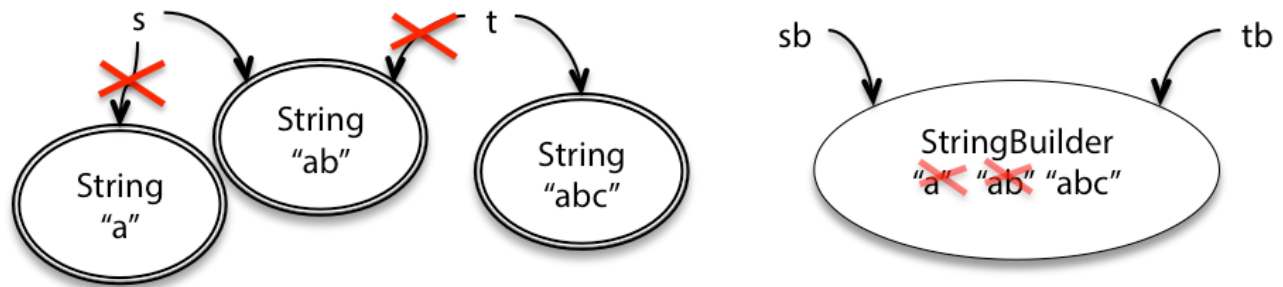
```
StringBuilder sb = new StringBuilder("a");  
sb.append("b");
```



# Mutability and Immutability

- So what? In both cases, you end up with `s` and `sb` referring to the string of characters "ab".
  - The difference between mutability and immutability doesn't matter much when there's only one reference to the object.
- But there are big differences in how they behave when there are other references to the object.
  - For example, 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**
  - The first number of the string ( "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.
- **StringBuilder is designed to minimize this copying.**
  - 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 + i;
}
```

```
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.
- Another is **convenient sharing**: two parts of your program can communicate more conveniently by sharing a common mutable data structure.
- “Global variables”
- **But you must have known the disadvantages of global variables...**

# Risks of mutation

- Since mutable types seem much more powerful than immutable types, why on earth would you choose the immutable one?
  - 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.**
  - Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts.
- Tradeoff between performance and safety?

# Risky example #1: passing mutable values

```
/** @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**

```
/** @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);
}
```

```
// 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 `sumAbsolute()` for going beyond what its spec allowed.
- 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**.

## ■ 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?

# Risky example #2: returning mutable values

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

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

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

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

```
// somewhere else in the code...  
public 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?  
}
```

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 repre 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 k

### 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());
```
- 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.

- **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.**
- **Immutability can be more efficient than mutability, because immutable types never need to be **defensively copied**.**



# Aliasing is what makes mutable types risky

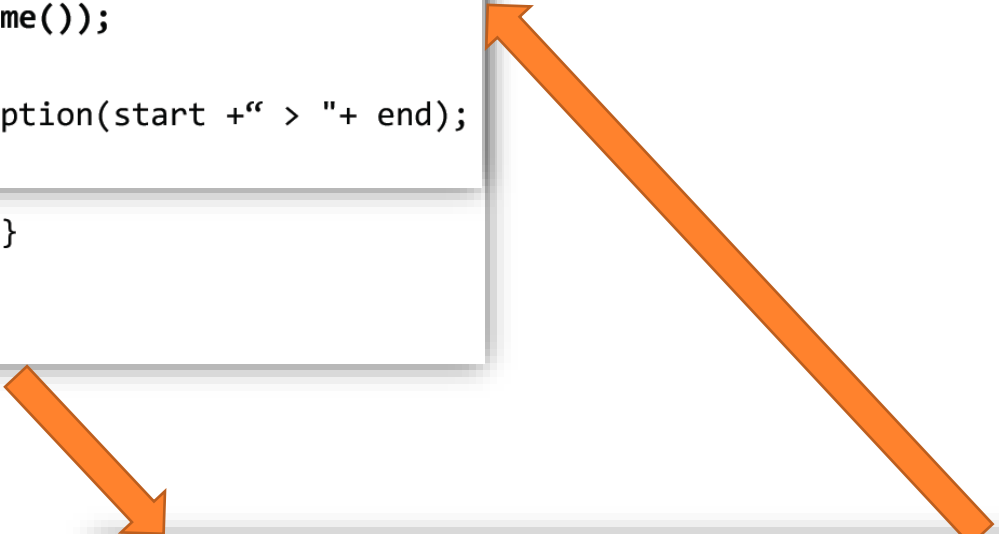
- Using mutable objects is just fine if you are using them entirely **locally** within a method, and with only one reference to the object.
- What led to the problem in the two examples we just looked at was **having multiple references**, also called **aliases**, for the same mutable object.
  - In 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

```
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());
    }
}
```



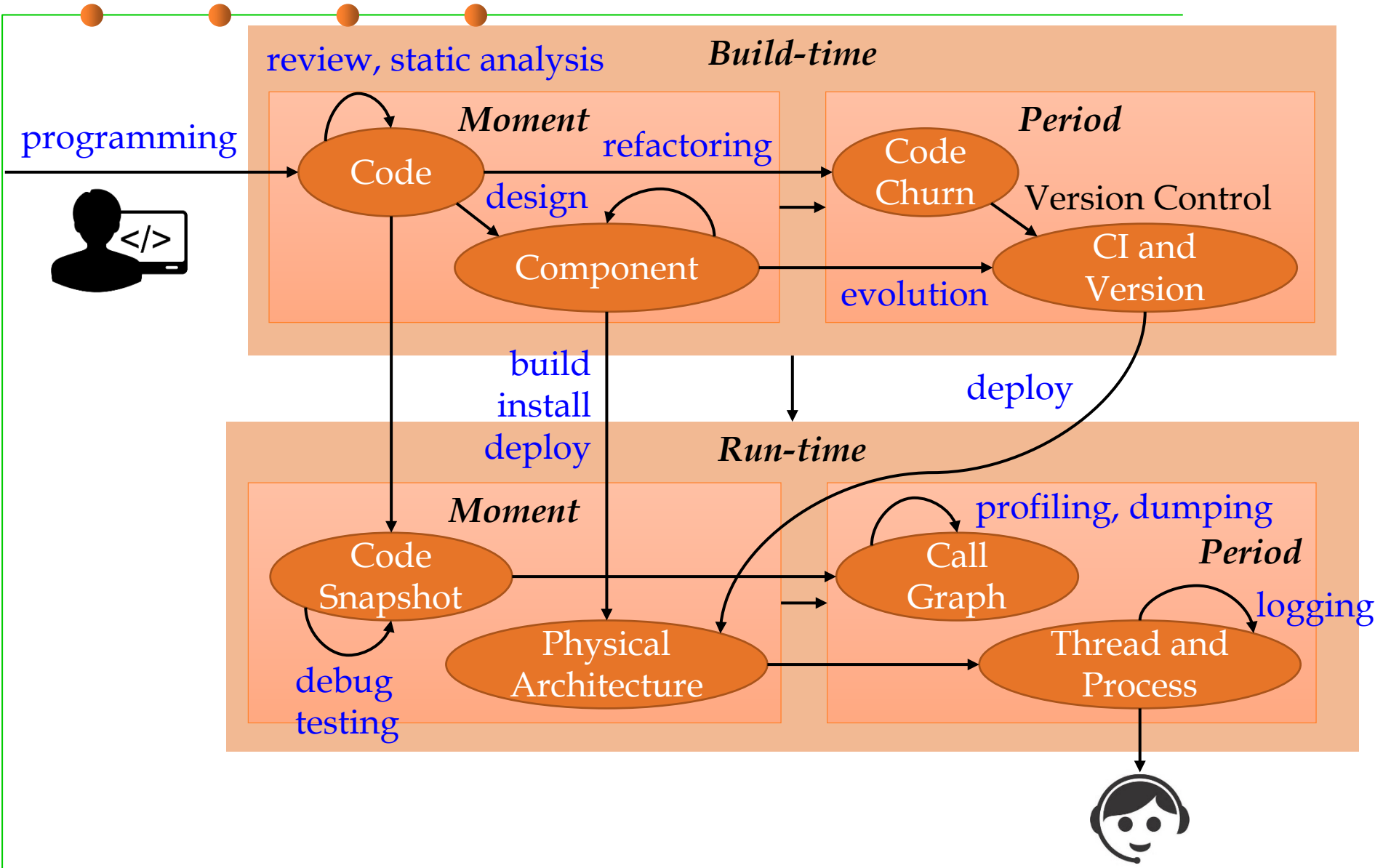
```
// Accessor attack on internals of Period
Period p = new Period(new Date(), new Date());
Date d = p.end();
p.end.setYear(78); // Modifies internals of p!
```



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



# Software construction: transformation btw views



# 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).
- **Why we use snapshot diagrams?**
  - To talk to each other through pictures.
  - To illustrate concepts like primitive types vs. object types, immutable values vs. immutable references, pointer aliasing, stack vs. heap, abstractions vs. concrete representations.
  - To help explain your design for your team project (with each other and with your TA).
  - To pave the way for richer design notations in subsequent courses.

# Mutating values vs. reassigning variables

- **Snapshot diagrams give us a way to visualize the distinction between changing a variable and changing a value:**
  - When you assign to a variable or a field, you're changing where the variable's arrow points. You can point it to a different value.
  - When you assign to the contents of a mutable value – such as an array or list – you're changing references inside that value.

# Primitive and Object values in Snapshot Diagram

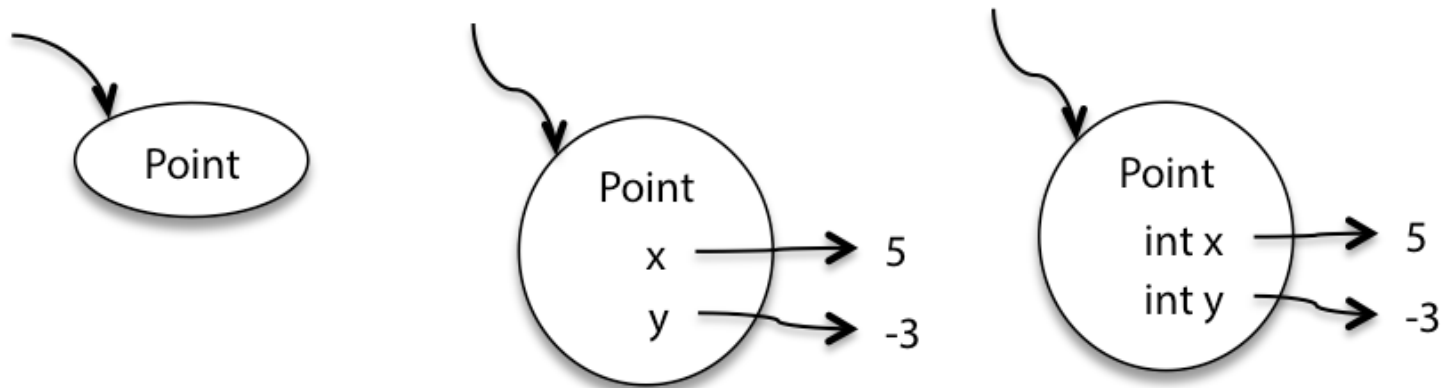
## ■ Primitive values

- 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

- 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. Some people prefer to write `x:int` instead of `int x`, but both are fine.





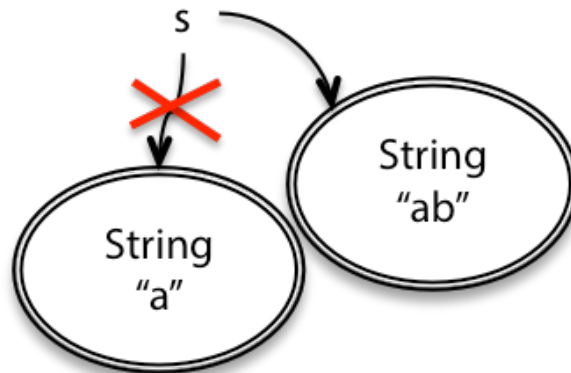
# Reassignment and immutable values

- 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.

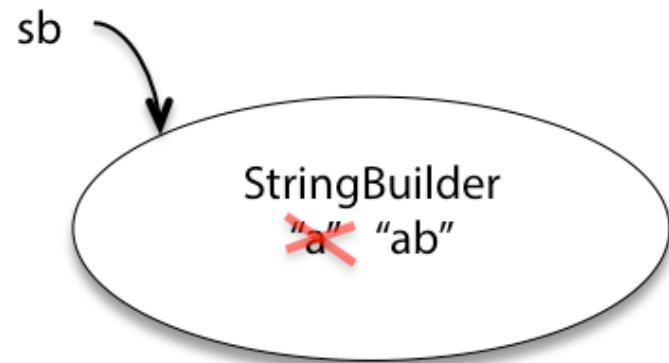
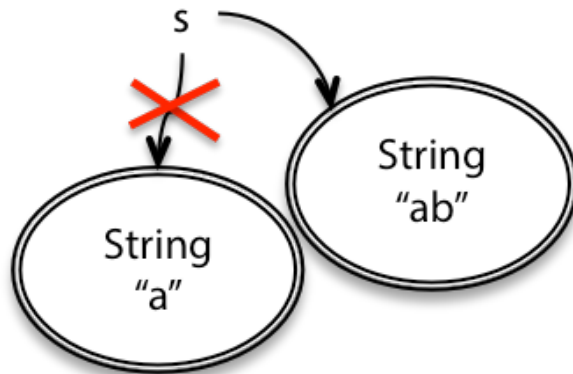


# Mutable values

- 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 .



# Immutable references

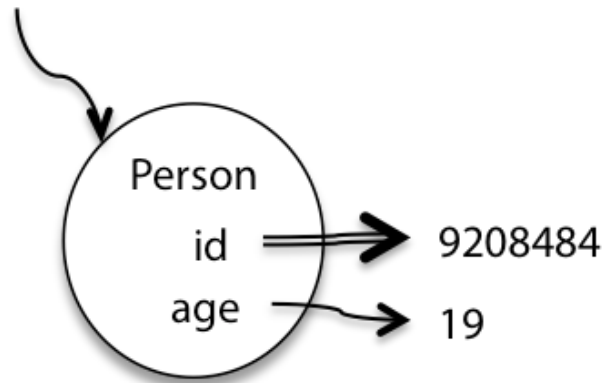
- Java also gives us immutable references: variables that are assigned once and never reassigned. To make a reference immutable, declare it with the keyword **final**:

```
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 immutable reference (**final**) is denoted by a **double arrow**.

# Immutable references

- Here's an object whose id never changes (it can't be reassigned to a different number), but whose age can change.



- 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.
- 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.



# 6 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`

```
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

- Instead of a fixed-length array, let's use the List type.
- Lists are variable-length sequences of another type T.

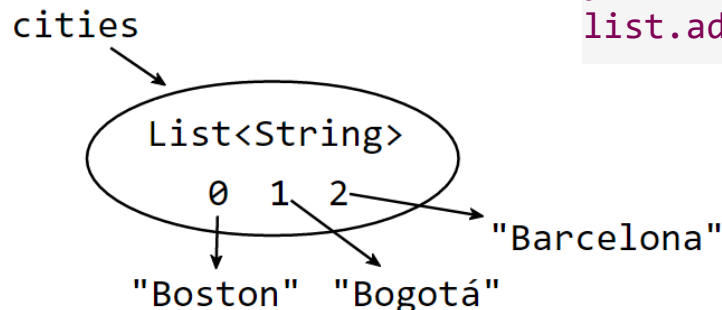
```
List<Integer> list = new ArrayList<Integer>();
```

- Some of its operations:

- indexing: `list.get(2)`
- assignment: `list.set(2, 0)`
- length: `list.size()`

- **Note 1:** List is an interface
- **Note 2:** members in a List must be an object.

```
List<Integer> list =  
    new ArrayList<Integer>();  
int n = 3;  
while (n != 1) {  
    list.add(n);  
    if (n % 2 == 0) {  
        n = n / 2;  
    } else {  
        n = 3 * n + 1;  
    }  
}  
list.add(n);
```



# Iterating

- **Iterating an array**

```
int max = 0;
for (int i=0; i<array.length; i++) {
    max = Math.max(array[i], max);
}
```

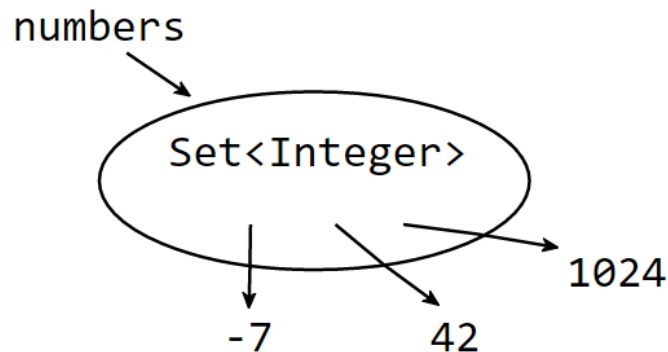
- **Iterating a List**

```
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  $s1 \supseteq s2$
  - `s1.removeAll(s2)` remove `s2` from `s1`
- Set is an abstract interface

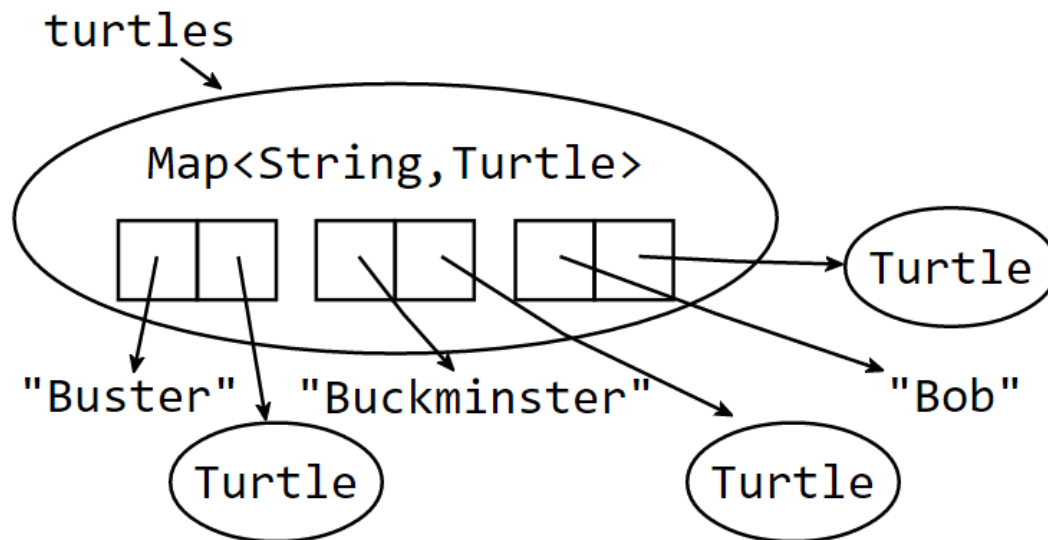


# Map

- **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**



# Building a list from an Array

- `Arrays.asList(new String[] { "a", "b", "c" })`

# 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<>();
```

# Iteration

```
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

- 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>

```
/**
 * A MyIterator is a mutable object that iterates over
 * the elements of an ArrayList<String> from first to last.
 * This is just an example of how an iterator works.
 * In practice, you should use the Iterator interface.
 * object, returned by its next() method.
 */
public class MyIterator {
```

Class  
definition

Instance  
variables

```
    private final ArrayList<String> list;
    private int index;
    // list[index] is the next element that will be returned
    // list.size() - index is the number of elements remaining
    // no more elements if index == list.size()
```

Specification

Invariants

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

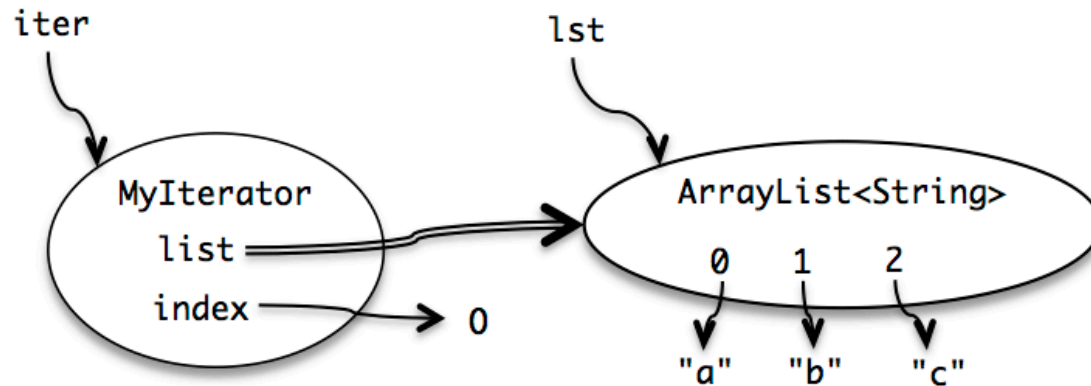
Constructor

```
/**
 * 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();
}
```

Instance  
method

```
/**
 * 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;
}
```

# Snapshot diagram of MyIterator



# Mutation undermines 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 (to be studied in Chapter 7)

```
// 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:

```
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

```
// dropCourse6(["6.045", "6.005", "6.813"])  
// expected [], actual ["6.005"]
```

Why?

- How to fix?

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



# 7 Useful immutable types



# Useful immutable types

- The primitive types and primitive wrappers are all immutable. 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

# 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);`





# 8 Null references



# Null references

- In Java, **references to objects and arrays can also take on the special value Null**, which means that the reference doesn't point to an object. Null values are an unfortunate hole in Java's type system.
- **Primitives cannot be null and the compiler will reject such attempts with static errors:**

```
int size = null; //illegal
```

- **Can assign null to any non-primitive variable**, and the compiler happily accepts this code at compile time. **But you'll get errors at runtime** because you can't call any methods or use any fields with one of these references (throws NullPointerExceptions):

```
String name = null;          name.length();
```

```
int[] points = null;         points.length;
```

- **null is not the same as an empty string "" or an empty array.**

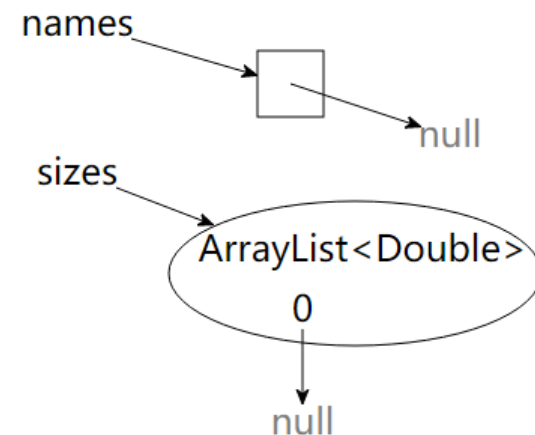
# Null references

- Arrays of non-primitives and collections like `List` might be non-null but contain `null` as a value
- These nulls are likely to cause errors as soon as someone tries to use the contents of the collection.

```
String[] names = new String[] { null };  
List<Double> sizes = new ArrayList<>();  
sizes.add(null);
```

- Null values are troublesome and unsafe**, so much so that you're well advised to remove them from your design vocabulary.
- Null values are implicitly disallowed in parameters and return values.**

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



## From Guava (by Google)

- “Careless use of null can cause a staggering variety of bugs. Studying the Google code base, we found that something like 95% of collections weren’t supposed to have any null values in them, and having those fail fast rather than silently accept null would have been helpful to developers.”
- “Additionally, null is unpleasantly ambiguous. It’s rarely obvious what a null return value is supposed to mean — for example, `Map.get(key)` can return null either because the value in the map is null, or the value is not in the map. Null can mean failure, can mean success, can mean almost anything. Using something other than null makes your meaning clear.”



# 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

March 7, 2018