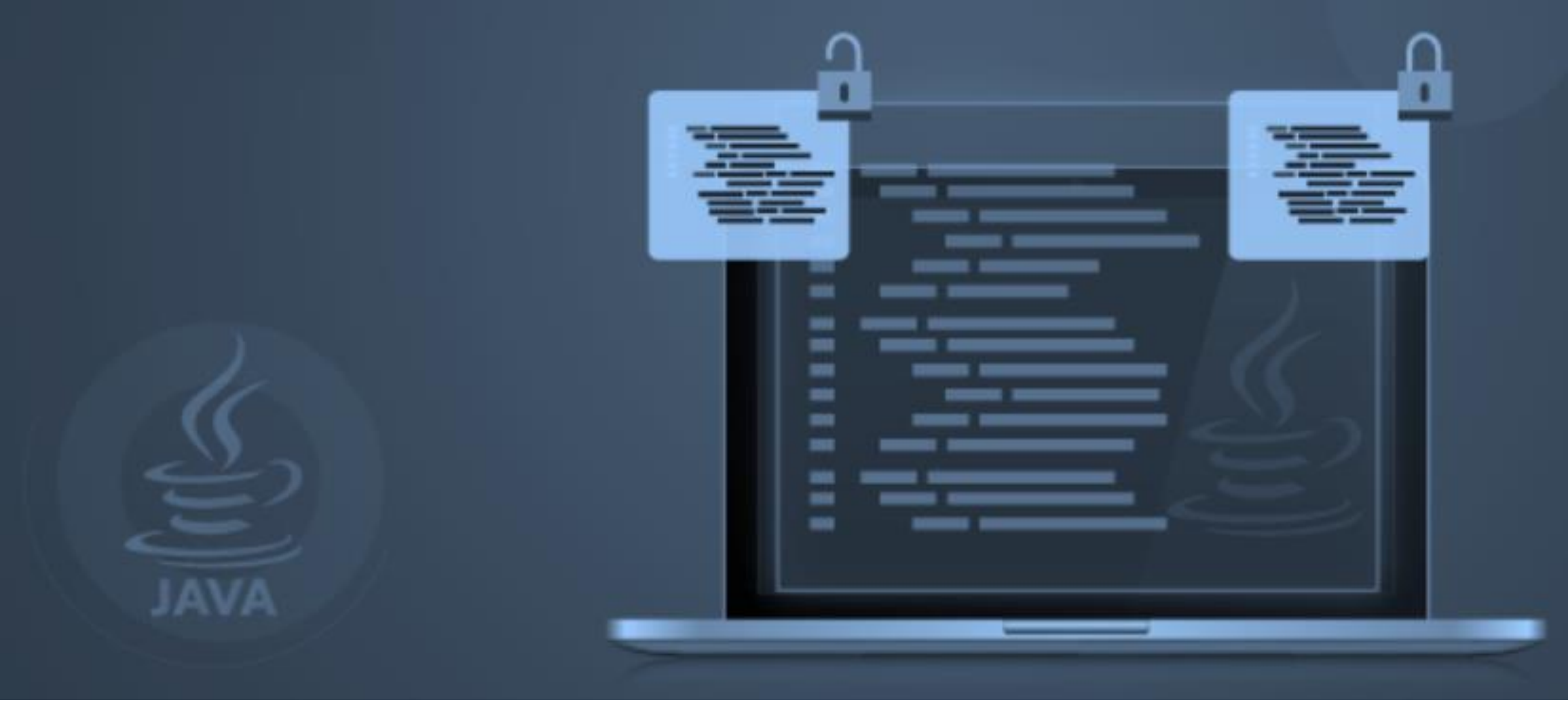


Advanced Object-Oriented Programming

CPT204 – Lecture 2
Erick Purwanto

Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT204 Advanced Object-Oriented Programming

Lecture 2

Checking 2, Testing 2, Immutability, List, Map

Welcome !

- Welcome to Lecture 2 !

- In this lecture we are going to
  - continue to learn about Checking and Testing
  - review Public, Static, Javadoc, List, Map
  - learn how to use Snapshot Diagrams
  - learn about Reassigning Variables vs Mutating Values
  - learn a bit about Immutability
    - immutable values
    - immutable reference by final

- We will continue learning about Mutability and Immutability in future lectures

Recall: Static Typing

- Last week, we mention that Java is a **statically-typed** language
  - The types of all variables have to be known at compile time (before the program runs)
  - In fact, as you experience in the lab, IntelliJ environment does this type checking *while you're still typing* your code !

- Static typing is a particular kind of **static checking**, which means checking for bugs at compile time
  - that is, bugs caused by applying an operation to the wrong types of arguments, for example, `string.size()` should be `string.length()`

Checking

Three kinds of automatic checking that a language can provide:

1. **Static checking**: the bug is found automatically before the program even runs
2. **Dynamic checking**: the bug is found automatically when the code is executed
3. **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 !

Static Checking

Static checking can catch:

- syntax errors, like extra punctuation or spurious words
- wrong names, e.g. `Math.sine(2)` (The right name is `sin`)
- wrong number of arguments, e.g. `Math.sin(30, 20)`
- wrong argument types, e.g. `Math.sin("30")`
- wrong return types, e.g. `return "30";` from a function that's declared to return an `int`

Dynamic Checking

Dynamic checking can catch:

- 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 Checking vs Dynamic Checking

- Static checking tends to be about types, errors that are independent of the specific value that a variable has
    - Recall that a type is a set of values. 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

No Checking (1)

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!

Here are the traps:

1. **Integer division**.
   5/2 does not return a fraction, it returns a truncated integer.
   So this is an example of where what we might have hoped would be a dynamic error (because a fraction isn't representable as an integer) frequently produces the *wrong* answer instead!

No Checking (2)

2. **Integer overflow.**
   The int and long types are actually finite sets of integers, with maximum and minimum values.
   What happens when you do a computation whose answer is too positive or too negative to fit in that finite range?
   The computation **quietly overflows** (wraps around), and returns an integer from somewhere in the legal range but *not* the right answer.

No Checking (3)

3. **Special values in floating-point types.**
   Floating-point types like double types have several special values that aren't real numbers: `NaN` (which stands for "Not a Number"), `POSITIVE_INFINITY`, and `NEGATIVE_INFINITY`.
   When you apply certain operations to a double that you'd expect to produce dynamic errors, like *dividing by zero* or taking the *square root of a negative number*, you will get one of these special values instead.
   If you keep computing with it, you'll end up with a bad final answer.

Quiz

- Recall our implementation of hailstone using array

```java
int[] a = new int[100];
int i = 0;
int n = 5;
while (n != 1) {
    a[i] = n;
    i++;
    if (n % 2 == 0) {
        n = n / 2;
    }
    else {
        n = 3 * n + 1;
    }
}
a[i] = n;
```

What would happen if we tried an n that turned out to have a very long hailstone sequence?
It wouldn't fit in a length-100 array
We have a bug !
Would Java catch the bug :

a. statically
b. dynamically
c. not at all ?

Review: Public and Static

- **`public`** means that any code, anywhere in your program, can refer to that method
- **`static`** means the method is associated with the class, not with an object
  - to call our static hailstone method below: `Hailstone.hailstone(100)`

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

the right way to call **static method** uses the class name

contrast that with **instance method**, which is called on an object, in this case, `list`

Javadoc Comments and Documenting Assumptions

- You can learn a lot from the method signature, such as

```
public static List<Integer> hailstone(int n) {
```

  e.g. the parameters' types and return type, but it is not enough

- We use the Javadoc comments, such as

```
/**
 * Compute a hailstone sequence.
 * For example, hailstone(5) = [5 16 8 4 2 1].
 * @param n starting number for sequence. Assumes n > 0.
 * @return hailstone sequence starting at n and ending with 1.
 */
```

  to document the additional assumption, in this case, in addition to assuming that parameter n is an integer, n is *also* greater than zero

Review: Javadoc Comments /** *... *... */

```
/**
 * Compute a hailstone sequence.
 * For example, hailstone(5) = [5 16 8 4 2 1].
 * @param n starting number for sequence. Assumes n > 0.
 * @return hailstone sequence starting at n and ending with 1.
 */
```

- Starts with **summary** of the method/class documented, optionally include one or two **examples**
- Describe *each* parameter with **`@param`**, state the assumptions
- Describe the return value with **`@return`**, if users follow the assumptions

Assumptions

- In method signature, writing `int n` means that `n` will always refer to an integer, never to a string or list or any other type
    - Java actually checks this assumption at compile time (static checking) and guarantees that there's no place in your program where you violated this assumption
- Unfortunately, Java doesn't check our other assumption that `n > 0` automatically
- Why do we need to write down our assumptions?
    - Because programming is full of assumptions!
    - If we don't write them down, we won't remember them, and other people who need to read or change our programs later won't know them. They'll have to guess...

Two goals

Programs have to be written with two goals in mind:

1. Communicating with **the computer**
   First persuading the compiler that your program is sensible — syntactically correct and type-correct — and then getting the logic right so that it gives the right results at runtime

2. Communicating with **other people**
   Making the program you write easy to understand, so that when somebody has to fix it, improve it, or adapt it in the future, they can do so! This somebody includes *you in the future*!

Review: Java Collections and List

- Java provides a number of more powerful and flexible tools for managing collections of objects: the **Java Collections Framework**
- It includes List (last week's and this week's lab) and Map (next week)

- A List contains an ***ordered*** collection of zero or more objects, where the same object might appear multiple times
  - we can add and remove items to and from the List, which will grow and shrink to accommodate its contents

- Please recall List's operations from last week!

Creating Lists (1)

- Java helps us distinguish between the *specification* of a type – *what does it do?* – and the *implementation* – *what is the code?*
- List and Map are all *interfaces*: they define **how** these respective types work, but they **don't** provide implementation code
- There are several advantages, but one potential advantage is that we, the users of these types, get to choose different implementations in different situations

- Here's how to create some actual Lists:
  - `List<String> list1 = new ArrayList<String>();`
  - `List<String> list2 = new LinkedList<String>();`

Creating Lists (2)

- If the generic type parameters are the **same** on the left and right, Java can infer what's going on and save us some typing:
  - `List<String> list1 = new ArrayList<>();`
  - `List<String> list2 = new LinkedList<>();`

empty here. let's use this!

- `ArrayList` and `LinkedList` are two implementations of List
  Both provide all the operations of List, and those operations must work as described in the documentation for List
  In this example, `list1` and `list2` will behave the same way, i.e. if we swap which one used `ArrayList` vs `LinkedList`, our code will **not** break

- Here in our course, let us just **always** use `ArrayList`

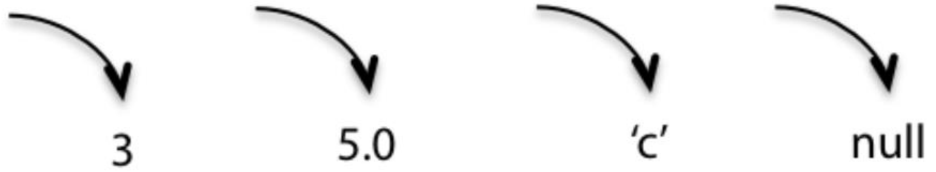
Snapshot Diagrams

- 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)
- We use snapshot diagrams:
  - To talk to each other through pictures (in class and in labs)
  - To illustrate concepts like primitive types vs. object types, immutable values vs. immutable references, pointer aliasing, stack vs. heap, abstractions vs. concrete representations

- Although the diagrams in this course use examples from Java, the notation can be applied to any modern programming language, e.g., Python, Javascript

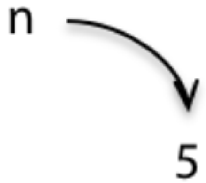
Primitive Values

- Primitive values are represented by bare constants



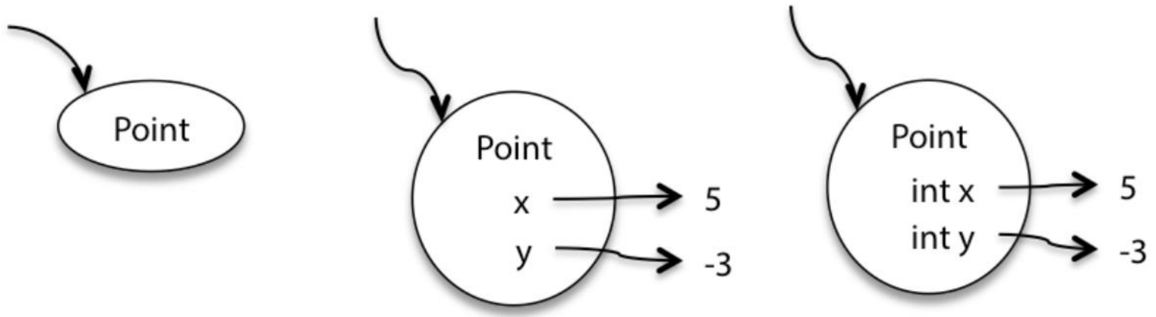
  - the incoming arrow is a reference to the value from a variable or an object field

- In its simplest form, a snapshot diagram shows a variable with an arrow pointing to its value



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

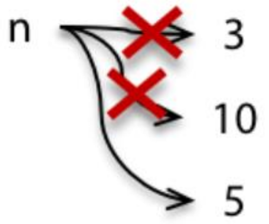


Reassigning Variables vs Mutating Values (1)

Snapshot diagrams give us a way to visualize the distinction between *changing a variable* and *changing a value*:

1. When you assign to a variable, you're changing where the variable's arrow points — you can point it to a different value

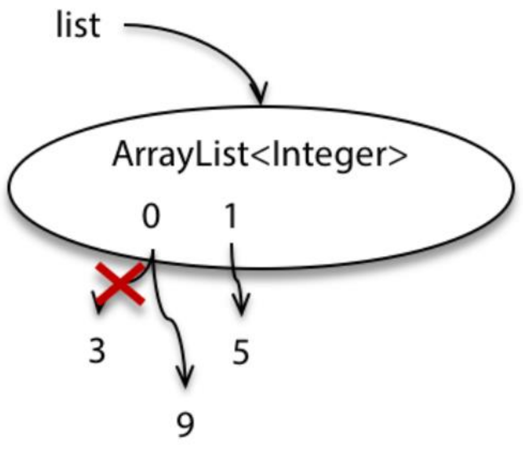
```
int n = 3;
n = 3*n + 1;
n = n/2;
```



Reassigning Variables vs Mutating Values (2)

2. When you change the contents of ***a mutable value*** — such as an array or list — you're changing references inside that value
    - this is called **mutating** the value

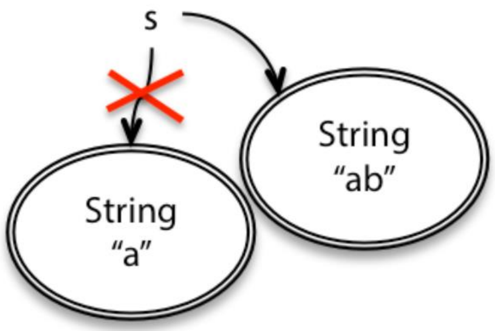
```
List<Integer> list = new ArrayList<>();
list.add(3);
list.add(5);
list.set(0, 9);
```



Reassignment and Immutable Values

- In the previous slide, a list is an example of a mutable value
- Now, a string is an example of ***an immutable value***
    - for example, if we have a `String` variable `s`, we can **reassign** it from a value of `"a"` to `"ab"`

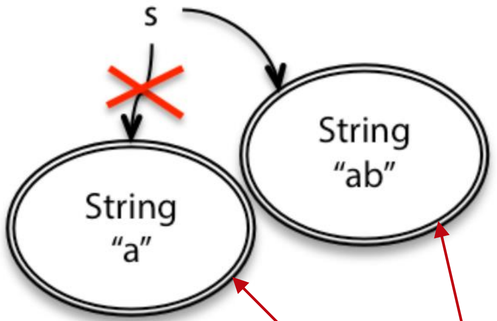
```
String s = "a";
s = s + "b";
```



Reassignment and Immutable Values

- In the previous slide, a list is an example of a mutable value
- Now, a string is an example of ***an immutable value***
  - 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";
```

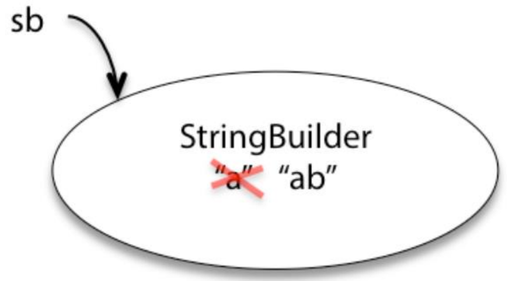


- String is an example of **an immutable type**, a type whose values can **never** change once they have been created
  - immutable values are denoted in a snapshot diagram by **a double border**, like the String objects in our diagram

Mutable Values

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

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



- Those two snapshot diagrams look ***very different***, which is good: the difference between mutability and immutability will play an important role in making our code *safe from bugs*

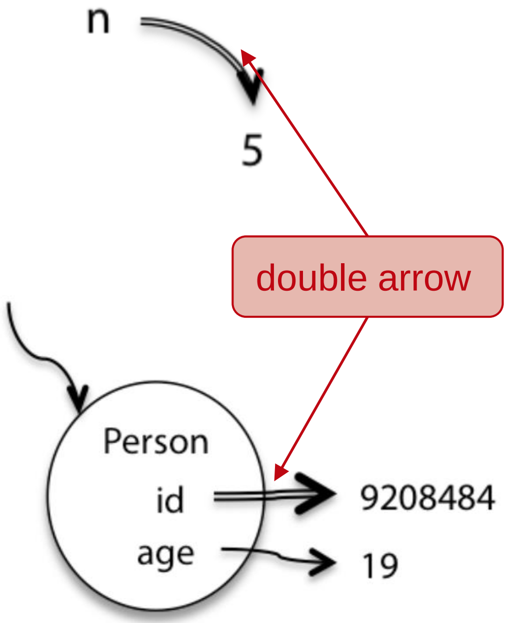
Immutable References

- 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 you *static checking* for immutable references!

- In a snapshot diagram, an immutable reference (final) is denoted by **a double arrow**. Here's an object whose `id` *never changes* (it can't be reassigned to a different number), but whose age *can change*



Reference vs Value

- Pay attention not to confuse *reference versus value*, when we talk about *mutability versus immutability*

- Notice that we can have ***an immutable reference*** to ***a mutable value*** (for example: `final StringBuilder sb`) whose value can change even though we're always pointing to the same object

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

Example of Immutable Reference

- It's good practice to use `final` for declaring the parameters of a method and *as many* local variables *as possible*

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

A

B

There are two variables in our hailstone method.

Can we declare them `final`, or not?

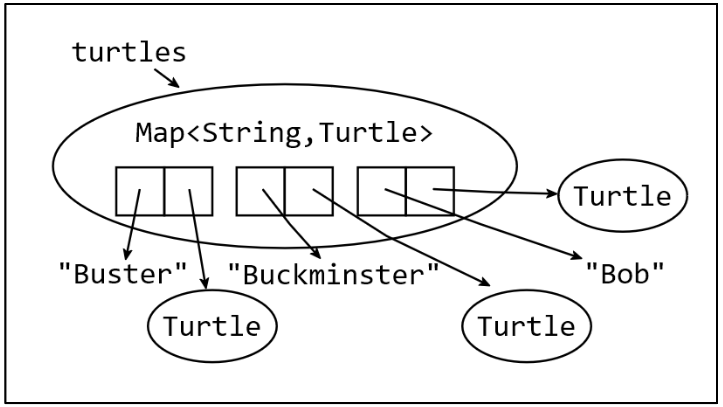
Final

- `final` can be used on both ***parameters*** and ***local variables***
  - when used on a parameter, `final` means that the parameter is assigned when the method is called, and then cannot be reassigned during the body of the method
  - when used on a local variable, `final` means that the variable cannot be reassigned after its first assignment, until the variable's scope ends
- `final` can be used on variables of ***any type*** — not just immutable types like `int`, but also mutable types like `List`
  - if a final variable points to a mutable object, then the variable ***cannot be reassigned***, **but** the object it points to ***can still be mutated***, for example by calling `add()` on a `List`

Map

- A **map** stores key/value pairs, where each **key** has an associated **value**
  - Given a particular key, the map can look up the associated value very quickly !

- Here is a snapshot diagram of a map where each key is a String such as "Bob", and the associated values are the Turtle objects assigned to their names



Map Operations and Implementations

- Operations in maps include:

`map.put(key, val)`	adding 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

- Our choice for the implementation of maps is `HashMap`

Map Operations Examples and Iteration

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

turtles.put("Buster", turtleBust);
turtles.put("Buckminster", turtleBuck);
turtles.put("Bob", turtleBob);

Turtle myTurtle;
myTurtle = turtles.get("Buster");

if (turtles.containsKey("Bob")) {
    Turtle turtle = turtles.get("Bob");
    // do something with turtle
}

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

create an empty map

get returns `null` if key is not in the map

we can iterate over the keys of a map

Map Example : NumWords

- Suppose we have a list of strings of multiple duplicate words, such as ["a", "b", "a", "c", "b", "word", "a", "word"]
- We want to find out which words appear in the list and how many times each one appears
- Here is an algorithm NumWords using Map:
    - Create a Map<String, Integer>
    - Loop through all the strings in the list
    - Use each string as a key into the map
    - Use the Integer value to store the number of times that string has been seen
    - There are 2 cases:
        - The first time we see a word, it is not yet in the map, we add it
        - The later times we see the word, it is already in the map, we update it

NumWords Code

```java
import java.util.HashMap;
import java.util.Map;

public class NumWords {
    public Map<String, Integer> wordCount(List<String> list) {
        Map<String, Integer> map = new HashMap<>();
        int num;
        for (String word : list) {
            if (!map.containsKey(word)) {
                map.put(word, 1);
            }
            else {
                num = map.get(word);
                map.put(word, num + 1);
            }
        }
        return map;
    }
}
```

the first time we've seen string `word`

we increment the number of `word`

Test-Driven Development and Corner Cases

- In Test-Driven Development (TDD), you start by writing the test code first *even before* writing the implementation code
  - How do get the input and the expected output you need to create the test cases?
  - By reading the specification in the problem description and the Javadoc, and by looking at the method signature — they are enough to create test cases!

- Always create test cases for the *corner cases*
  - For example: empty list, singleton list (list with just one element), smallest list with answer equals zero/non-zero or false/true

Thank you for your attention !

- In this lecture, you have learned:
  - Review 2
    - Public, Static, Javadoc Comments, Collections, Lists, Map
  - Checking and Testing 2
    - Static Checking, Dynamic Checking, No Checking
    - Assumptions, Test-Driven Development, Corner Cases
  - Snapshot Diagrams, Values vs Reference
  - Mutability vs Immutability
    - Mutable/Immutable Values, Final Immutable Reference

- Please continue to Lecture Quiz 2 in LMO; and Lab 2:
  - to practice testing with JUnit,
  - to do Lab Exercise 2.1 - 2.2, and Exercise 2.1 - 2.4