

# Advanced Object-Oriented Programming

CPT204 – Lecture 0  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

# CPT204 Advanced Object-Oriented Programming

## Lecture 0

**Introduction & Course Logistics**

Hi all !

---

- Welcome to CPT204 Hyflex Edition!
  - I am Erick, your instructor
  - I hope you all are safe and healthy !
- 
- As you know, the campus is closed for 1 week for pandemic mitigation
    - so we will have to do online lecture and online labs
  - In week 2, we have on-site lecture but *online-only labs*
  - Starting from week 3, we plan to have *both* on-site and online lectures and labs
    - you may choose to attend on-site or online with Zhumu
    - in addition, recordings will be made available
  - Please continue to read this guide ...



# Course Goals and Prerequisites

---

- We're going to learn how to:
  - Write code efficiently
    - Design, build, test, and debug large programs
    - Use programming tools e.g. IntelliJ, JUnit
  - Write good code
    - Correct
    - Easy to understand for other programmers
    - Ready to be changed in the future
  - Write efficient code
    - Good algorithms
    - Good data structure
- Assume solid foundation in programming fundamentals (CPT104, CPT105)
  - Basic OOP, recursion, big O notation, stacks, queues, lists, trees, ...

# Where to Get Information ?

---

- Learning Mall
  - Announcement, Lecture and Lab materials
  - Discussion Forum
  - In-class Quizzes
  - Lab Exercises
  - Theory Quizzes
  - Programming Exercises
- Office Hour and Email
  - only for private matters
- Optional Textbooks
  - Object-Oriented Problem Solving Java by Ralph Morelli, Ralph Walde
  - Effective Java by Joshua Bloch

# Teaching Team and Schedule

---



- Instructor : Erick Purwanto
  - [erick.purwanto@xjtlu.edu.cn](mailto:erick.purwanto@xjtlu.edu.cn) SD545 Thursday 14.00-16.00
- Teaching Assistants :
  - Kai, Ruben, Xue, Yuanying, Yue, Yida
- Online Weeks
  - Online Lectures: Tuesday 13-15/16-18
  - Online Labs: materials in Learning Mall, released Thursday/Friday
- Hyflex Weeks
  - Lectures: Tuesday 13-15/16-18, SC169
  - Labs: Friday 15-16/16-17/17-18, SD554/SD546

# Teaching Assistant

---

- Kai Yao
  - His research interest is Deep Learning based Computer Vision
  - His current work is Dense Instance Segmentation and Classification



- Ruben Ng
  - His research interest is Information Visualisation and Explainable AI
  - His recent work is Visual Explainable AI for Biochip Classification Interfaces



# Teaching Assistant

---

- Xue Wang
  - Her research interests are about Electrical and Electronics
  - She is currently working on Differential Power Processing Converter in PV and Solid-state Circuit Breaker



- Yuanying Qu
  - Her research interest is on Integrated Detection System of Acoustic Wave
  - Her recent work is Indoor Human-centered Acoustic Recognition and Detection



# Teaching Assistant

---

- Yue Hu
  - His research interest is Computer Vision, Object Recognition, Data Science, and Financial Computing
  - He is currently working on Multimodal feature fusion, Object recognition based on Radar and LiDAR Sensor



- Yida Yin
  - His research interest is Machine Learning and Data Science
  - His current work is the application of Neural Network in Finance Analysis



# Assessment

---

1. Coursework : 50%
  - o Coding Exam
  - o in last Lab session, details TBA
  - o Related to weekly Lab Exercise and Programming Exercise
  
2. Final Exam : 50%
  - o Theory and Programming
  - o in Exam Hall/Lab, details TBA

# How do you learn?

---

- Learn **some** from
  - Listening to lectures
  - Reading the materials
- Learn **more** from
  - Trying out the codes from lectures and readings yourself
    - Discussing them with your friends / in forum
  - Doing the lecture and lab exercises
  - Doing the theory quiz + programming exercises
  - Experiment on your own
- You are encouraged to discuss the exercises with your study group
  - But write the code *yourself* !

## Lab Exercises

---

- Not graded, but will be related to Coding Exam
- Closed in 1 week
- Weekly lab exercise:
  - Attempts allowed: 1
  - Must pass **all** test cases
  - No penalty
    - Unlimited Checks
- Solution is available after closing
- Will be discussed in the slides (online) / by TAs (hyflex in labs)
  - but you should first try to solve it on your own!
- Can be discussed in Discussion Forum

# Programming Exercise and Theory Quiz

---

- Not graded, but will be related to Coding Exam
- Closed in 1 week
- Weekly programming exercise:
  - Attempts allowed: 1
  - Must pass **all** test cases
  - First check: no penalty
  - Second check and so on: cumulative 15% penalty
- Theory quizzes:
  - Attempts allowed: 1
- Solution is available after the closing
- Ask about the question if it is not clear in forum
  - but **do not post** the solution code

# Attending Lectures and Labs

---

- You will see explanation and sample codes in the slides
  - Various tools in Learning Mall
    - LMO In-class Quiz
    - LMO Autograded Quiz for Exercises (see Lab 1)
  - Copy paste the sample codes into IntelliJ IDE in your computer
    - Modify and run it, play with it yourself !
- Complete the slides
  - Print and scribble, or use a pdf annotation program
  - Recordings provided
- If you have further questions, post your questions in the LMO Forum

# Challenges and Forum

---

- Challenges :
  - It's a challenging course with ambitious learning goals
  - We have a large cohort of ~ 400 students
  - We are *still* in the middle of a once-in-a-century pandemic
- Don't email, post your questions in the Discussion Forum



- You are also **encouraged** to **answer** your friends' questions
  - let us use forum to effectively answer your questions!
- Remember, **do not** post code or solution
  - it will ruin other students learning development

# Discussion Forum and General Attitude

---

- We are all in this unpredicted and quite difficult situation now
- Please in posting in the forum or email, always remember to:
  - Be kind
  - Be patient
  - Be helpful
- In addition:
  - Search first, maybe your friend has asked the same thing before
  - Post in English, be specific, provide enough information
    - For example, don't just write that some error happens, but explain what did you do that might cause it, write the error message, OS/browser you use
    - Provide screenshot if necessary, but make sure it does not contain your code

# Your Long-term Goals

---

- Not just good grade
- Job interview, graduate study, recommendation letter

**Recommendation Rankings**

Please rank the applicant in all categories listed below.

I am comparing this applicant to:

Knowledge in subject of proposed study      Intellectual ability/Problem solving      Ability to learn

Ability to work with others      Responsiveness to feedback      Originality/Creativity

Logic/Analytical/Mathematical ability      English written communication      English oral communication

Perseverance towards goals      Laboratory skills (for lab-related disciplines)

Overall Recommendation

Recommendation Letter

In addition to the ratings above, the Duke University Graduate School requests a candid statement from you concerning this applicant. Please type the statement on official letterhead, if possible.

- Please explain in detail the applicant's accomplishments, abilities, character, and capacity for success as a graduate student in the proposed field of study.
- If the applicant's native language is not English, please comment on the applicant's English proficiency.

You may attach your recommendation letter below as a Word or PDF document.

Recommendation Letter

No file chosen

Exceptional (Top 2%)  
Superior (Top 5%)  
Excellent (Top 15%)  
Above Average (Top 25%)  
Average (Top 50%)  
Below Average (Lower 50%)  
Unable to Rate

# Thank you for your attention !

---

- Next, please go to Lab 0 to install Java, IntelliJ, its plugin and libraries
- And then, go to Lecture 1 and start reading this week's material
- Finally, go to Lab 1 to start exploring our tools and do this week's programming exercises...

# Acknowledgement

---

- Some of the materials are taken from courses by
  - Dr Grant Malcolm, University of Liverpool
  - Dr Josh Hug, UC Berkeley
  - Prof Max Goldman, Prof Rob Miller, MIT



# Advanced Object-Oriented Programming

CPT204 – Lecture 1  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

# CPT204 Advanced Object-Oriented Programming

## Lecture 1

**Basic Java Review,  
Checking 1, Testing 1**

# Welcome to your first online lecture !

---

- Welcome to Lecture 1 !
  - It is recommended that you finish Lecture 0 and Lab 0 first
- In this lecture we are going to
  - review some basic concepts you've learned in Intro Java and Data Structure
  - learn the first part about checking
  - learn the first part about testing
- We will continue learning about checking and testing in future lectures

# Hailstone Sequence

---

- As an example in this lecture, we use the hailstone sequence
- Hailstone sequence starts with a number **n**
  - the next number in the sequence is  **$n/2$**  if  $n$  is even, or  **$3n+1$**  if  $n$  is odd
  - it ends when it reaches **1**
  - for example:
    - $\text{hailstone}(5) = [5 \ 16 \ 8 \ 4 \ 2 \ 1]$
    - $\text{hailstone}(3) = [3 \ 10 \ 5 \ 16 \ 8 \ 4 \ 2 \ 1]$
    - $\text{hailstone}(8) =$
    - $\text{hailstone}(2^n) =$
- Interestingly, we still don't know if every hailstone sequences always reaches 1



# Printing Hailstone

---

- Here's a Java code to simply print a hailstone sequence

```
int n = 5;
while (n != 1) {
    System.out.println(n);
    if (n % 2 == 0) {
        n = n/2;
    }
    else {
        n = 3*n + 1;
    }
}
System.out.println(n);
```

# Types

---

- In Java, you have to declare the type of a variable before starting using it
  - not in Python
- A **type** is a set of values, along with operations that can be performed on those values
- Java has **8 primitive types**
  - boolean, byte, char, short, int, long, float and double
- Java also has **object types**
  - String, BigInteger
- By Java convention, primitive types are lowercase, while object types start with a capital letter

# Operations

---

- Operations are functions that take input values and produce output values
- There are three different syntaxes for an operation in Java:
  - As an infix, prefix, or postfix operator
    - e.g., `a + b` is infix operation `+` : `int × int → int`
  - As a method of an object
    - e.g., `bigint1.add(bigint2)` calls the operation `add` : `BigInteger × BigInteger → BigInteger`
  - As a function
    - e.g., `Math.sin(x)` calls the operation `sin` : `double → double`
- Some operations are **overloaded**, same operation name is used for different types
  - e.g., arithmetic operators `+, -, *, /` are overloaded for numeric primitive types

# Static Typing

---

- Java is a **statically-typed** language
- The types of all variables have to be known at compile time (before the program runs), and the compiler can therefore deduce the types of all expressions as well
  - For example, if a and b are declared as ints, then the compiler concludes that  $a+b$  is also an int
  - In fact, you will shortly see in our lab, IntelliJ environment does this *while you're still typing* your code !
- In **dynamically-typed** languages like Python or Javascript, this kind of checking is deferred until runtime (while the program is running)

# Static Checking

---

- Static typing is a particular kind of **static checking**, which means checking for bugs at **compile time**
- Static typing prevents a large class of bugs from infecting your program, in particular, bugs caused by applying an operation to the wrong types of arguments
  - for example, 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 coding*, rather than waiting until the line is reached during execution
- We will explore **checking** again in more detail next week

# Review: String

---

- A string is a series of characters gathered together
  - Create a string by writing its chars out between double quote

```
String hello = "Hello!";
```
- Operations on String types include:
  - length `hello.length()`
  - indexing `hello[0]`
  - concatenation `"a" + "bc"`
  - substring `hello.substring(3)`  
`hello.substring(3, 5)`
  - check presence `hello.contains("lo")`
  - search `hello.indexOf("lo")`
  - equality test `hello.equals("hello!")`

## Review: Arrays

---

- Arrays are fixed-length sequences of another type
- To declare an array variable and construct an array value to assign to it:
  - `int[] a = new int[100];`
  - it includes all possible int array values, but once an array is created, we can never change its length
- Operations on array types include:
  - indexing              `a[2]`
  - assignment          `a[2] = 0`
  - length                `a.length`

# Hailstone with Array

---

- We want to store the sequence in an array, instead of just printing it out

```
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;
```

## Review: List and ArrayList

---

- Instead of a fixed-length array, let's use the List type!  
Lists are variable-length sequences of another type
- To declare a List variable and make a list value:
  - `List<Integer> list = new ArrayList<Integer>();`
- Some of its operations:
  - indexing            `list.get(2)`
  - assignment        `list.set(2, 5)`
  - add                `list.add(5)`
  - length            `list.size()`
- Create a list from an array  
`List<Integer> list = Arrays.asList(10, 20, 30)`

## Review: List and ArrayList

---

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

- List is an **interface**, a type that can't be constructed directly with new, but that instead *specifies the operations* that a List must provide
  - ArrayList is a class, a concrete type that provides *implementations of those operations*
  - ArrayList isn't the only implementation of the List type, though it's the most commonly used one
    - LinkedList is another implementation

## Review: List and ArrayList

---

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

- We wrote `List<Integer>` instead of `List<int>`  
Lists only know how to deal with *object types*, not *primitive types*
  - In Java, each of the primitive types (lowercase, abbreviated) has an equivalent object type (capitalized, fully spelled out)
  - Java requires us to use these object type equivalents when we **parameterize** a type with <angle brackets>

# Hailstone with List

---

- Here is the hailstone code written with Lists:

```
List<Integer> list = new ArrayList<Integer>();  
int n = 5;  
while (n != 1) {  
    list.add(n);  
    if (n % 2 == 0) {  
        n = n/2;  
    }  
    else {  
        n = 3*n + 1;  
    }  
}  
list.add(n);
```

# Iterate through List

---

- You can use the enhanced for loops to iterate through a list
  - for example:

```
// print elements of a hailstone sequence stored in list
for (int x : list) {
    System.out.print(x + " ");
}
```

- As a simple exercise, write a code to find the maximum element in a list of integers using the enhanced for loop !

# Methods

---

- In Java, statements are inside a method, and every methods has to be in a class; so, the simplest code of our hailstone program is:

```
public class Hailstone {  
    /**  
     * 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.  
    */  
    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;  
    }  
}
```

# Testing

---

- Now suppose you have written a code like the one in the previous slide
- How do you know that you have written a **correct** code?
  - You write a test code!
- You may add a main method containing a test code to print the **actual** result against the **expected** output like the following:

```
public static void main(String[] args) {  
    int n = 5;  
    List<Integer> list = hailstone(n);  
    System.out.println("Expected: 5 16 8 4 2 1");  
    System.out.print("Actual:    ");  
    for (int x : list) {  
        System.out.print(x + " ");  
    }  
}
```

# Test with JUnit

---

- Automate your tests using JUnit test

```
import java.util.Arrays;
import java.util.List;
import org.junit.Test;
import static org.junit.Assert.*;

public class HailstoneTest {
    @Test
    public void testHailstone() {
        List<Integer> expected = Arrays.asList(5, 16, 8, 4, 2, 1);
        List<Integer> actual = Hailstone.hailstone(5);
        assertEquals(expected, actual);
    }

    @Test
    public void testMaxHailstone() {
        int expectedMax = 16;
        assertEquals(expectedMax, Hailstone.maxHailstone(5));
    }
}
```

# Multiple assertEquals

---

- You can create many test methods, and many assertEquals inside a test method

```
import java.util.Arrays;
import java.util.List;
import org.junit.Test;
import static org.junit.Assert.*;

public class HailstoneTest {
    @Test
    public void testHailstone() {
        List<Integer> expected = Arrays.asList(5, 16, 8, 4, 2, 1);
        List<Integer> actual = Hailstone.hailstone(5);
        assertEquals(expected, actual);

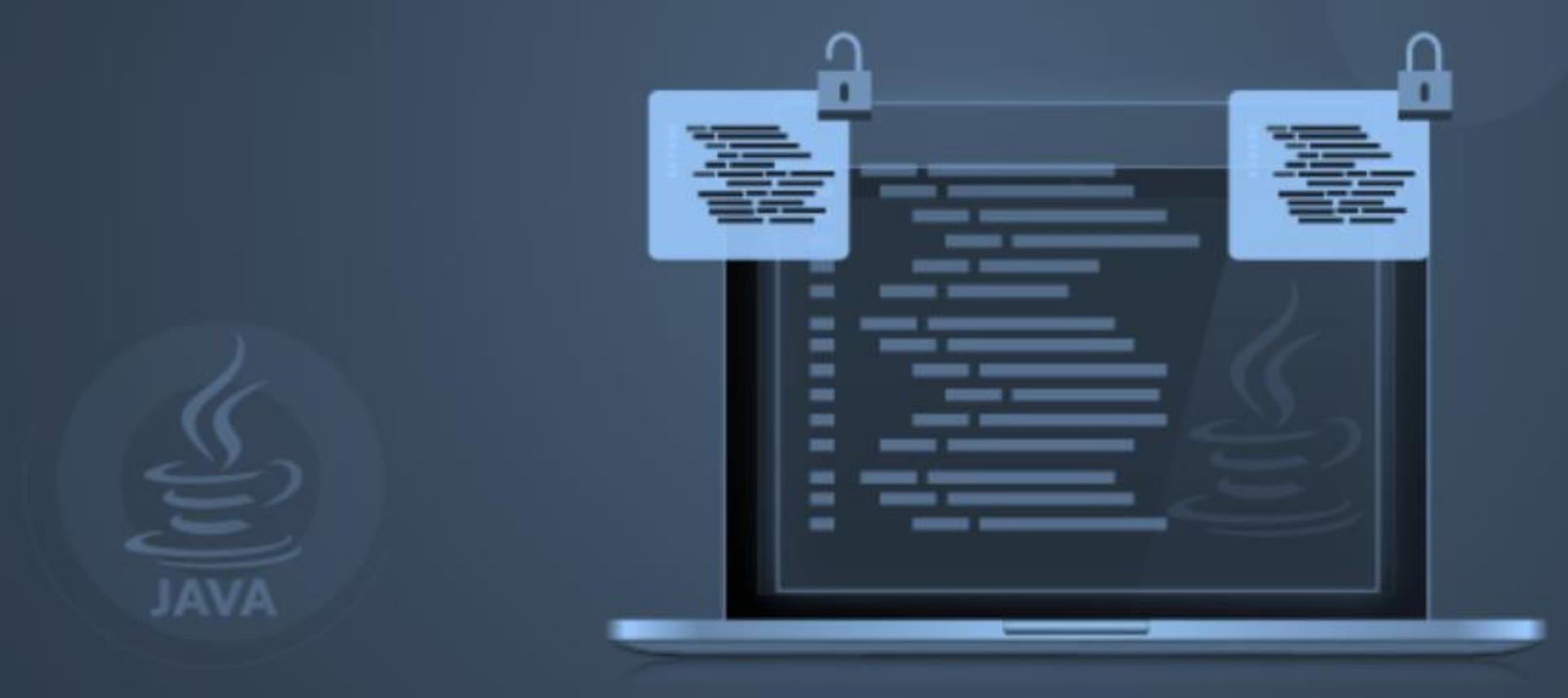
        expected = Arrays.asList(3, 10, 5, 16, 8, 4, 2, 1);
        actual = Hailstone.hailstone(3);
        assertEquals(expected, actual);
    }
}
```

- We will explore testing more in future lectures!

# Thank you for your attention !

---

- In this lecture, you have learned:
  - Review 1
    - Types, Operators, Arrays, Lists
  - Checking 1
    - Static Typing, Static Checking
  - Testing 1
    - Expected vs Actual, JUnit Testing, assertEquals
- Please continue to Lab 1,
  - to practice testing with JUnit,
  - to do Lab Exercise 1.1, and
  - to do Exercise 1.1, 1.2



# 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

```
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)`

```
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

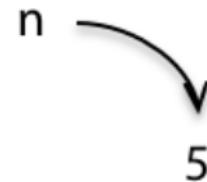
# Primitive Values

---

- Primitive values are represented by bare constants



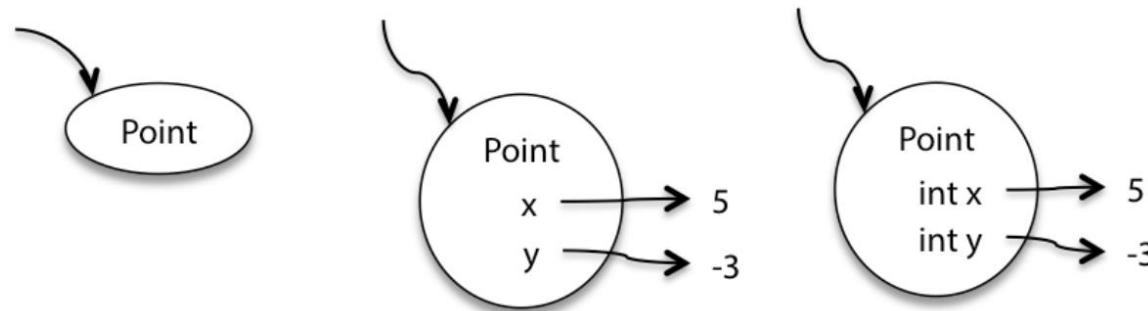
- the incoming arrow is a reference to the value from a variable or an object field
- In its simplest form, a snapshot diagram shows a variable with an arrow pointing to its value



# 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



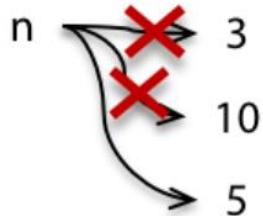
# Reassigning Variables vs Mutating Values (1)

---

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;
```

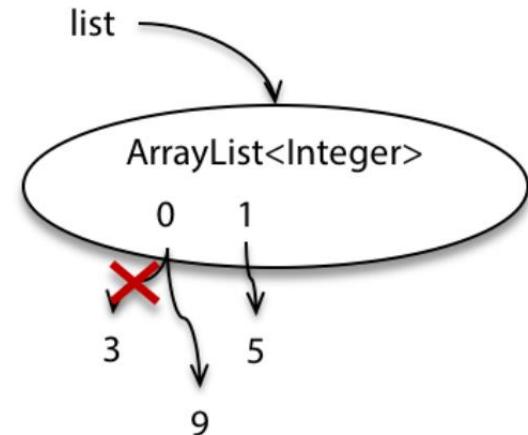


## Reassigning Variables vs Mutating Values (2)

---

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

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

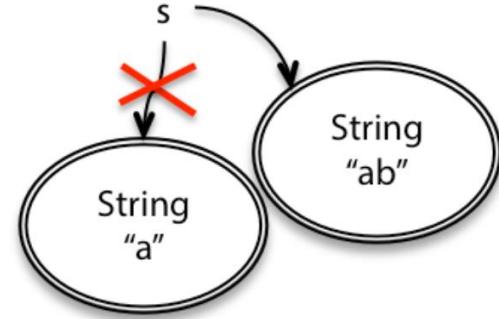


# Reassignment and Immutable Values

---

- 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

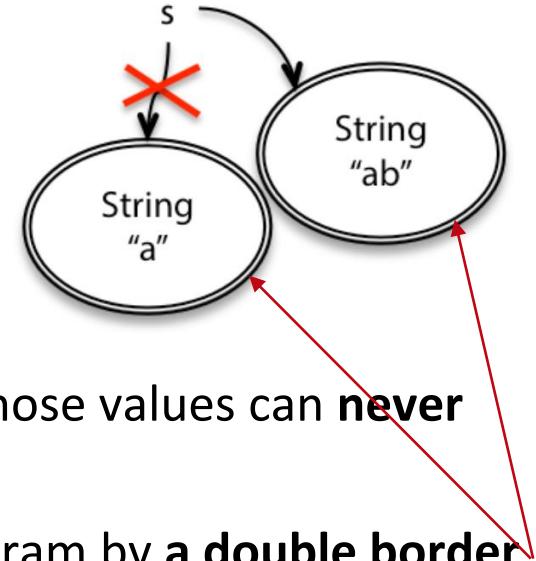
---

- 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



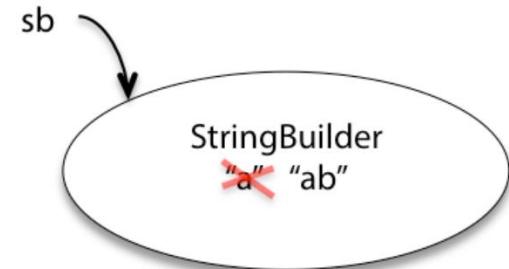
# Mutable Values

---

- 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");
```



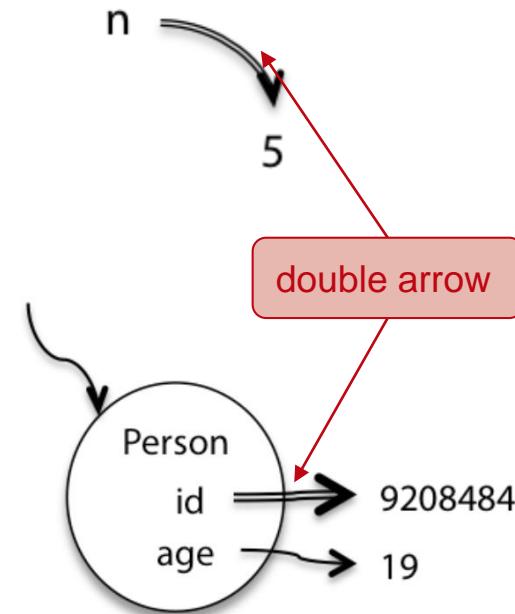
- Those two snapshot diagrams look ***very different***, which is good: the difference between mutability and immutability will play an important role in making our code *safe from bugs*

# 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

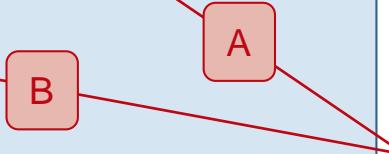
---

- 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*

```
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;  
}
```



There are two variables in our hailstone method.

Can we declare them `final`, or not?

# Final

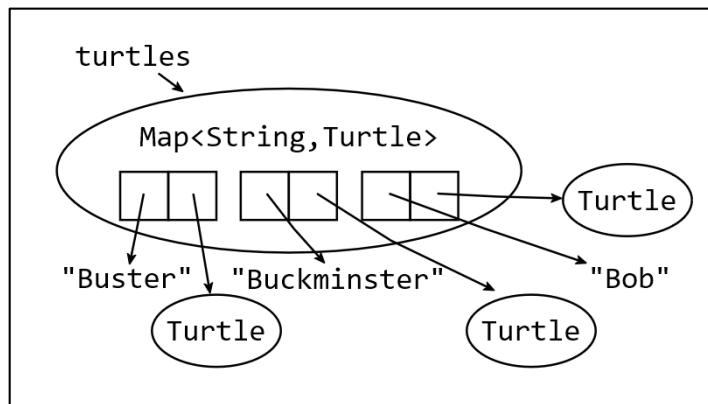
---

- `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<>();
```

create an empty map

```
turtles.put("Buster", turtleBust);
```

```
turtles.put("Buckminster", turtleBuck);
```

```
turtles.put("Bob", turtleBob);
```

```
Turtle myTurtle;
```

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

get returns null if key is not in the map

```
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));
```

```
}
```

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

```
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



# Advanced Object-Oriented Programming

CPT204 – Lecture 3  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming

### Lecture 3

**Coding Rules, Testing 3,  
Recursion**

# Welcome !

---

- Welcome to Lecture 3 !
- In this lecture we are going to
  - learn about Coding Rules
  - continue our discussion about Testing
    - how to choose the test suite
  - review Recursion and Recursion + Helper Method

## Part 1: Coding Rules and Testing 3

---

There are sensible **coding rules** that help us to write good code: making it safe from bugs, easy to understand, and ready for change:

- Don't Repeat Yourself (DRY)
- Comments where needed
- Fail fast
- Avoid magic numbers
- One purpose for each variable
- Use good names
- Don't use global variables
- Return results, don't print them
- Use whitespace for readability

# Don't Repeat Yourself

---

- Duplicated code is a risk to safety: if you have *identical* or *very similar* code in *two* places, then the fundamental risk is that there's a bug in both copies, and some maintainer fixes the bug in one place but not the other
- Avoid duplication like you'd avoid crossing the street without looking. Copy-and-paste is an enormously tempting programming tool, and you should be careful every time you use it!
  - The longer the block you're copying, the riskier it is
- Don't Repeat Yourself, or DRY for short, has become a programmer's mantra
- The dayOfYear example in the next slides is **full of identical code**
  - How would you DRY it out?

# Don't Repeat Yourself

---

- The code below is full of identical codes: 1) why is this a BAD code? 2) how do you DRY it ?

```
public static int dayOfYear(int month, int dayOfMonth, int year) {  
    if (month == 2) {  
        dayOfMonth += 31;  
    } else if (month == 3) {  
        dayOfMonth += 59;  
    } else if (month == 4) {  
        dayOfMonth += 90;  
    } else if (month == 5) {  
        dayOfMonth += 31 + 28 + 31 + 30;  
    } else if (month == 6) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31;  
    } else if (month == 7) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;  
    } else if (month == 8) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;  
    } else if (month == 9) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;  
    } else if (month == 10) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;  
    } else if (month == 11) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;  
    } else if (month == 12) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;  
    }  
    return dayOfMonth;  
}
```

# Don't Repeat Yourself

- The code below is full of identical codes: 1) why is this a BAD code? 2) how do you DRY it ?

```
public static int dayOfYear(int month, int dayOfMonth, int year) {  
    if (month == 2) {  
        dayOfMonth += 31;  
    } else if (month == 3) {  
        dayOfMonth += 59;  
    } else if (month == 4) {  
        dayOfMonth += 90;  
    } else if (month == 5) {  
        dayOfMonth += 31 + 28 + 31 + 30;  
    } else if (month == 6) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31;  
    } else if (month == 7) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;  
    } else if (month == 8) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;  
    } else if (month == 9) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;  
    } else if (month == 10) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;  
    } else if (month == 11) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;  
    } else if (month == 12) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;  
    }  
    return dayOfMonth;  
}
```

1) because:  
there repeated values e.g. 28,  
and code, e.g. the dayofMonth +=  
if you change a value/code,  
you have to change all of them!

# Don't Repeat Yourself

- The code below is full of identical codes: 1) why is this a BAD code? 2) how do you DRY it ?

```
public static int dayOfYear(int month, int dayOfMonth, int year) {  
    if (month == 2) {  
        dayOfMonth += 31;  
    } else if (month == 3) {  
        dayOfMonth += 59;  
    } else if (month == 4) {  
        dayOfMonth += 90;  
    } else if (month == 5) {  
        dayOfMonth += 31 + 28 + 31 + 30;  
    } else if (month == 6) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31;  
    } else if (month == 7) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;  
    } else if (month == 8) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;  
    } else if (month == 9) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;  
    } else if (month == 10) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;  
    } else if (month == 11) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;  
    } else if (month == 12) {  
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;  
    }  
    return dayOfMonth;  
}
```

2) use an array or list to store  
the number of days once, and  
use loop to add and return the value

# Comments Where Needed: Specification

- Good comments should make the code easier to understand, safer from bugs (because important assumptions have been documented), and ready for change
- One kind of crucial comment is a **specification (spec)**, which appears above a method or above a class and *documents the behavior* of the method or class
  - In Java, this is conventionally written as a Javadoc comment, meaning that it starts with `/**` and includes `@`-syntax, like `@param` and `@return` for methods
  - Here's an example of a spec:

```
/**  
 * 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.  
 */
```

specs document assumptions

# Comments Where Needed: Document Source

---

- Another crucial comment is one that specifies the source of a piece of code that was copied or adapted from elsewhere. This is important for practicing software developers, when you adapt code you found on the web, for example:

```
// read a web page into a string
// see http://stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-javascript
String homepage = new Scanner(new URL("http://www.aaa.com")).openStream(), "UTF-8")
    .useDelimiter("\A").next();
```

- One reason for documenting sources is *to avoid violations of copyright*
  - Small snippets of code on Stack Overflow are typically in the public domain, but code copied from other sources may be proprietary or covered by other kinds of open source licenses, which are more restrictive
- Another reason for documenting sources is that *the code can fall out of date*; the Stack Overflow answer from which this code came has evolved significantly in the years since it was first answered

# Comments Where Needed: Bad Comments

- Some comments are bad and unnecessary
- Direct transliterations of code into English, for example, do nothing to improve understanding, because you should assume that your reader at least knows Java
  - For example:

```
while (n != 1) { // test whether n is 1
    ++i; // increment i
    l.add(n); // add n to l
}
```

don't write  
comment like this !

- But, obscure code should get a comment:

```
// base cases
if (n == 0 || n == 1) {
    return 1;
}
```

# Fail Fast

---

- Fail fast means that code should *reveal its bugs as early as possible*
  - The earlier a problem is observed (the closer to its cause), the easier it is to find and fix
  - As we saw in Lecture 2, static checking fails faster than dynamic checking, and dynamic checking fails faster than no checking at all, which may produce a wrong answer that could corrupt subsequent computation

```
public static int dayOfYear(int month, int dayOfMonth, int year) {  
    ...  
}
```

- The `dayOfYear` method ***doesn't fail fast*** — if you pass it the arguments in *the wrong order*, it will **quietly** return *the wrong answer*, as you will see in Lecture Quiz this week
  - It needs more checking — either static checking or dynamic checking

# Avoid Magic Numbers

---

- There are really only two constants that computer scientists recognize as valid in and of themselves: 0, 1, and maybe 2 (ok, three constants -\_- )
- All other constants are called **magic**, because they appear as if out of thin air with no explanation
  - One way to explain a number is with a comment, but a far better way is to declare the number as **a named constant with *a good, clear name***
- Our previous example, `dayOfYear`, is full of magic numbers

```
...  
} else if (month == 5){  
    dayOfMonth += 31 + 28 + 31 + 30;  
...  
...
```

magic numbers

# Avoid Magic Numbers: Better Ways

---

- The months 2, ..., 12 would be far more readable as FEBRUARY, ..., DECEMBER
- The days-of-months 30, 31, 28 would be more readable (and DRY: eliminate duplicate code) if they were in a data structure like an array, list, or map
  - e.g. MONTH\_LENGTH[month]
- The mysterious numbers 59 and 90 are particularly damaging examples of magic numbers!
  - Not only are they *uncommented* and *undocumented*,  
but they are also the result of a computation *done by hand* by the programmer
  - Don't hardcode constants that you've computed by hand!  
Java is better at arithmetic than you are!  
Explicit computations like 31 + 28 make the origin of these mysterious numbers  
much clearer  
MONTH\_LENGTH[JANUARY] + MONTH\_LENGTH[FEBRUARY] would be clearer still!

# One Purpose for Each Variable

- In the `dayOfYear` example, the parameter `dayOfMonth` is *reused* to compute a very different value — the return value of the method, which is **not** the day of the month

```
public static int dayOfYear(int month, int dayOfMonth, int year) {  
    if (month == 2) {  
        dayOfMonth += 31;  
    } ...  
    return dayOfMonth;  
}
```

unnecessary  
reuse

- Don't reuse parameters, and don't reuse variables
  - Variables are not a scarce resource in programming
  - Introduce them freely, give them good names, and just stop using them when you stop needing them
  - You will **confuse** your reader if a variable that used to mean one thing suddenly starts meaning something different a few lines down

# One Purpose for Each Variable: Method Parameter

---

- Method parameters, in particular, should generally be *left unmodified*
  - This is important for being ready-for-change — in the future, some other part of the method may want to know what the original parameters of the method were, so you should *not* change them while you're computing
- It's a good idea to use `final` for method parameters, and as many other variables as you can
  - The `final` keyword says that the variable should *never be reassigned*, and the Java compiler will check it *statically*
  - For example:

```
public static int dayOfYear(final int month, final int dayOfMonth, final int year) {  
    ...  
}
```

# Use Good Names

---

- Good method and variable names are long and self-descriptive
  - Comments can often be avoided entirely by making the code itself more readable, with better names that describe the methods and variables
  - For example, you can rewrite

```
int tmp = 86400; // tmp is the number of seconds in a day
```

as:  
`int secondsPerDay = 86400;`
- In general, variable names like tmp, temp, and data are awful symptoms of extreme programmer laziness: every local variable is temporary, and every variable is data, so those names are generally meaningless
  - It is better to use a longer, more descriptive name, so that your code reads clearly all by itself

# Use Good Names: Naming Conventions

---

- Follow the lexical naming conventions of the language
- In Java:
  - methodsAreNamedWithCamelCaseLikeThis
  - variablesAreAlsoCamelCase
  - CONSTANTS\_ARE\_IN\_ALL\_CAPS\_WITH\_UNDERSCORES
  - ClassesAreCapitalized
  - packages.are.lowercase.and.separated.by.dots

# Use Good Names: Names

---

- Method names are usually verb phrases, like `getDate` or `isUpperCase`, while variable and class names are usually noun phrases
  - Choose short words, and be concise, but avoid abbreviations
  - For example, `message` is clearer than `msg`, and `word` is so much better than `wd`
    - Keep in mind that abbreviations can be even harder for non-native speakers
- `ALL_CAPS_WITH_UNDERSCORES` is used for static final constants
- All variables declared inside a method, including final ones, use `camelCaseNames`
- We have seen that a method named `leap` has a bad name
  - We named it `isLeapYear`

# Don't Use Global Variables

---

- A global variable is:
  - a variable, a name whose meaning can be changed
  - that is global, accessible and changeable from *anywhere* in the program
- The following website has a good list of the dangers of global variables:  
<http://web.archive.org/web/20160902115611/http://c2.com/cgi/wiki?GlobalVariablesAreBad>
- In Java, a global variable is declared **public static**: the **public** modifier makes it accessible anywhere, and **static** means there is **a single instance** of the variable
- In general, **change** global variables into parameters and return values, or **put** them inside objects that you're calling methods on (make it instance/local variable)

# Don't Use Global Variables: An Example

---

- How do you change this code, so that it doesn't use any global variables?

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

public static void countLongWords(List<String> words) {
    int n = 0;
    longestWord = "";
    for (String word: words) {
        if (word.length() > LONG_WORD_LENGTH) ++n;
        if (word.length() > longestWord.length()) longestWord = word;
    }
    System.out.println(n);
}
```

# Return Results, Don't Print Them

---

- `countLongWords` in previous slide also isn't ready for change!
  - It sends some of its result to the console, `System.out`
  - That means that if you want to use it in another context — where the number is needed for some other purpose, like computation rather than human eyes — it would have to be rewritten!
- In general, only the highest-level parts of a program should interact with the human user or the console
  - Lower-level parts should take their input as parameters and return their output as results
  - The sole exception here is *debugging* output, which can of course be printed to the console. However, that kind of output shouldn't be a part of your design, only a part of how you debug your design

# Use Whitespace for Readability

- Use *consistent* indentation
  - A very very bad example :

```
public static boolean leap(int y) {  
    String tmp = String.valueOf(y);  
    if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == 5 || tmp.charAt(2) == '7' || tmp.charAt(2) == '9') {  
        if (tmp.charAt(3)=='2'||tmp.charAt(3)=='6') return true;  
        else  
            return false;  
    }else{  
        if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {  
            return false;  
        }  
        if (tmp.charAt(3)=='0'||tmp.charAt(3)=='4'||tmp.charAt(3)=='8')return true;  
    }  
    return false;  
}
```

so painful to read,  
don't know where  
the if's end

- Never use tab characters for indentation, only space characters
  - Always set your programming editor to insert space characters when you press the Tab key

# Testing and Bugs

---

- Testing means running the program on carefully selected inputs and checking the results
- Even with the best testing, it is very hard to achieve perfect quality in software
- Here are some **typical residual defect rates** (bugs left over after the software has shipped) per kloc (one thousand lines of source code):
  - 1 - 10 defects/kloc: Typical industry software
  - 0.1 - 1 defects/kloc: High-quality validation.  
The Java libraries might achieve this level of correctness.
  - 0.01 - 0.1 defects/kloc: The very best, safety-critical validation.  
NASA and aerospace companies can achieve this level.
- This can be discouraging for large systems
  - for example, if you have shipped a million lines of typical industry source code (1 defect/kloc), it means you missed 1000 bugs!

# Why Software Testing is Hard (1)

---

Here are some approaches that unfortunately ***don't work well*** in the world of *software*:

- **Exhaustive testing** is *infeasible*  
The space of possible test cases is generally too big to cover exhaustively.  
Imagine exhaustively testing a 32-bit floating-point multiply operation,  $a*b$ .  
There are  $2^{64}$  test cases!
- **Haphazard testing** (“just try it and see if it works”) is *less likely to find bugs*,  
unless the program is so buggy that an arbitrarily-chosen input is more likely to fail  
than to succeed.  
It also doesn't increase our confidence in program correctness.

# Why Software Testing is Hard (2)

---

- Random or statistical testing *doesn't work well for software*

Other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer the defect rate for the whole production lot.

- **Physical systems** can use many tricks to speed up time, like opening a refrigerator 1000 times in 24 hours instead of 10 years.

These tricks give known failure rates (e.g. mean lifetime of a hard drive), but they assume continuity or uniformity across the space of defects.

This is true for physical artifacts.

- But, it's **not** true for **software**.

Software behavior varies discontinuously and discretely across the space of possible inputs.

Although the system may seem to work fine across a broad range of inputs, it could still abruptly fail at a single boundary point.

## Why Software Testing is Hard (3)

---

The famous Pentium division bug affected approximately ***1 in 9 billion divisions***. Stack overflows, out of memory errors, and numeric overflow bugs tend to happen abruptly, and always in the same way, not with probabilistic variation.

That's different from *physical systems*, where there is often visible evidence that the system is approaching a failure point (cracks in a bridge) or failures are distributed probabilistically near the failure point (so that statistical testing will observe some failures even before the point is reached).

- Instead, test cases must be chosen ***systematically***
  - **A test case** is a particular choice of inputs, along with the expected output behavior required by the specification
  - **A test suite** is a set of test cases for an implementation

# Systematic Testing (1)

---

Systematic testing means that we are choosing test cases in a principled way, with the goal of designing a test suite with three desirable properties:

- **Correct.** A correct test suite is a legal client of the specification, and it accepts all legal implementations of the spec without complaint
  - This gives us the freedom to change how the module is implemented internally without necessarily having to change the test suite
- **Thorough.** A thorough test suite finds actual bugs in the implementation, caused by mistakes that programmers are likely to make

## Systematic Testing (2)

---

- **Small.** A small test suite, with few test cases, is faster to write in the first place, and easier to update if the specification evolves
  - Small test suites are also faster to run
  - You will be able to run your tests more frequently if your test suites are small and fast
- By these criteria,
  - Exhaustive testing is thorough but infeasibly large
  - Haphazard testing tends to be small but not thorough
  - Randomized testing can achieve thoroughness only at the cost of large size

# Putting on Your Testing Hat

---



- Testing requires having the *right attitude*
- When you're coding, your goal is to make the program work, but as a tester, you want to **make it fail**
- That's a subtle but important difference
  - it's all too tempting to treat code you've just written as a *precious* thing, a fragile eggshell, and test it ever so lightly just to see it work
- Instead, you have to be *brutal*
  - A good tester wields a sledgehammer and beats the program everywhere it might be vulnerable, so that those vulnerabilities can be eliminated

# Test-First Programming

---

In test-first-programming, you write tests before you even write any code

The development of a *single* function proceeds in this order:

1. **Spec:** Write a *specification* for the function
2. **Test:** Write *tests* that exercise the specification
3. **Implement:** Write the actual code

Once your code passes the tests you wrote, you're done

# The Specification

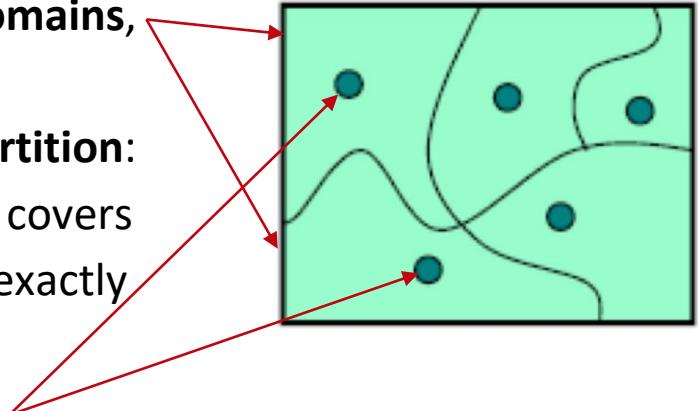
---

- The specification (**spec**) describes the input and output behavior of the function
  - It gives the types of the parameters and any additional constraints on them (e.g. `sqrt`'s parameter must be nonnegative)
  - It also gives the type of the return value and describes how the return value relates to the inputs
  - You've already seen and used specifications on your *programming exercises* and *assignments* in this class
  - In code, the specification consists of ***the method signature*** and ***the comment above it that describes what it does***
- Writing tests first is a good way to understand the specification. The specification can be buggy, too — *incorrect, incomplete, ambiguous, missing corner cases*  
Trying to write tests can uncover these problems early, before you've wasted time writing an implementation of a buggy spec

# Choosing Test Cases by Partitioning

---

- Creating a good test suite is a challenging and interesting design problem
  - We want to pick a set of test cases that is *small enough* to run quickly, yet *large enough* to validate the program
- To do this, we divide the input space into **subdomains**, each consisting of a set of inputs
  - Taken together the subdomains form a **partition**: a collection of *disjoint* sets that *completely* covers the input space, so that every input lies in exactly one subdomain
  - Then we choose **one test case from each subdomain**, and that's our test suite



# Subdomains

---

- The idea behind subdomains is to partition the input space into sets of *similar inputs* on which the program has *similar behavior*
  - Then we use **one** representative of each set
  - This approach makes the best use of limited testing resources by choosing dissimilar test cases, and
  - forcing the testing to explore parts of the input space that random testing might not reach
- We can also partition the *output space* into subdomains (similar outputs on which the program has similar behavior) if we need to ensure our tests will explore different parts of the output space
  - Most of the time, partitioning the input space is sufficient

# Example 1: BigInteger.multiply()

Let's look at an example:

- BigInteger is a class built into the Java library that can represent integers of *any* size, unlike the primitive types int and long that have only limited ranges.  
BigInteger has a method multiply that multiplies two BigInteger values together:

```
/**  
 * @param val another BigInteger  
 * @return a BigInteger whose value is (this * val).  
 */  
public BigInteger multiply(BigInteger val)
```

its spec

- Here's how it might be used:

```
BigInteger a = new BigInteger("9500000000"); // 9.5 billion  
BigInteger b = new BigInteger("2");  
BigInteger c = a.multiply(b); // should be 19 billion
```

## BigInteger.multiply() Arguments

---

- The example shows that even though only one parameter is explicitly shown in the method's declaration, `multiply` is actually a function of *two arguments*:
  - the object you're calling the method on (`a` in the example), and the parameter that you're passing in the parentheses (`b` in the example)
- In Java, you don't mention the receiving object in the parameters, and it's called **this**
- So we should think of `multiply` as a function taking two inputs, each of type `BigInteger`, and producing one output of type `BigInteger`:
  - `multiply : BigInteger × BigInteger → BigInteger`

## Partition the Input Space (1)

---

So we have a two-dimensional input space, consisting of all the pairs of integers  $(a, b)$

Now let's partition it!

Thinking about how multiplication works, we might start with these partitions:

- $a$  and  $b$  are both positive
- $a$  and  $b$  are both negative
- $a$  is positive,  $b$  is negative
- $a$  is negative,  $b$  is positive

There are also some special cases for multiplication that we should check: 0, 1 and -1

- $a$  or  $b$  is 0, 1 or -1

## Partition the Input Space (2)

---

Finally, as a suspicious tester trying to find bugs, we might suspect that the implementer of BigInteger might try to make it faster by using int or long internally when possible, and only fall back on an expensive general representation (like a list of digits) when the value is too big.

So we should definitely also try integers that are very big, bigger than the biggest long:

- a or b is small
- the absolute value of a or b is bigger than Long.MAX\_VALUE  
the biggest possible primitive integer in Java, which is roughly  $2^{63}$

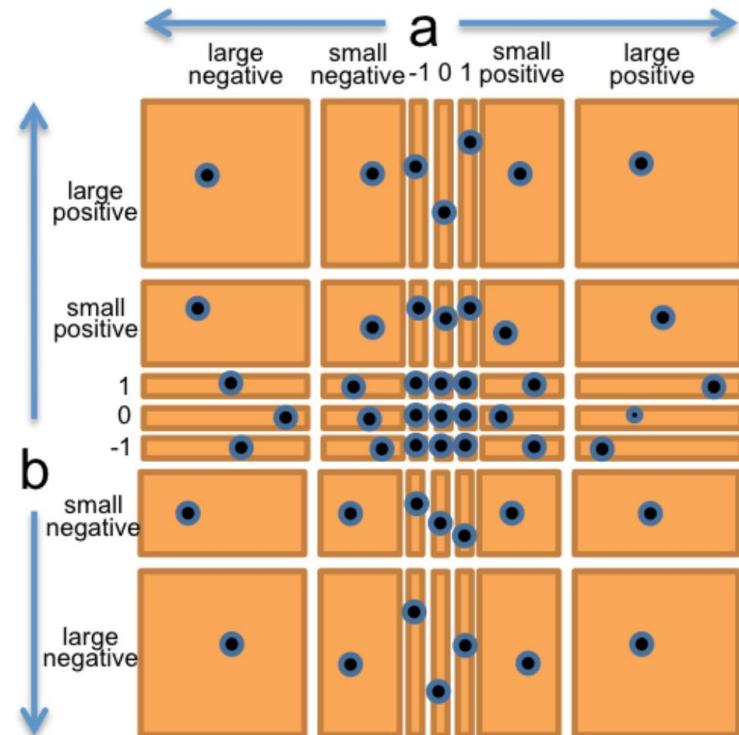
## Partition the Input Space (3)

Let's bring all these observations together into a straightforward partition of the whole  $(a, b)$  space.

We'll choose  $a$  and  $b$  independently from:

- 0
- 1
- -1
- small positive integer
- small negative integer
- huge positive integer
- huge negative integer

So this will produce  $7 \times 7 = 49$  partitions that completely cover the space of pairs of integers



## Partition the Input Space (4)

---

To produce the test suite,  
we would pick an arbitrary pair  $(a, b)$  from each square of the grid, for example:

- $(a, b) = (-3, 25)$  to cover (small negative, small positive)
- $(a, b) = (0, 30)$  to cover (0, small positive)
- $(a, b) = (2^{100}, 1)$  to cover (large positive, 1)
- and so on

## Example 2: max()

---

Let's look at another example from the Java library:  
the integer `max()` function, found in the `Math` class

```
/**  
 * @param a  an argument  
 * @param b  another argument  
 * @return the larger of a and b.  
 */  
public static int max(int a, int b)
```

- Mathematically, this method is a function of the following type:
  - `max : int × int → int`

# Partition the Input Space

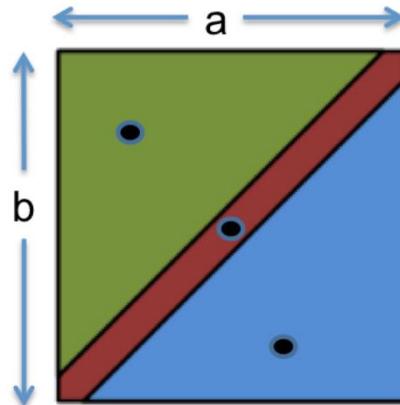
---

From the specification, it makes sense to partition this function as:

- $a < b$
- $a = b$
- $a > b$

Our test suite might then be:

- $(a, b) = (1, 2)$  to cover  $a < b$
- $(a, b) = (9, 9)$  to cover  $a = b$
- $(a, b) = (-5, -6)$  to cover  $a > b$



# In-Class Quiz 1

---

- You want to partition the input space of this integer square root function:

```
/**  
 * @param x is a nonnegative integer  
 * @return nearest integer to the square root of x  
 */  
public static int intSqrt(int x)
```

- Which one is a **good partition?**
  - Partition:  $x < 0$  and  $x \geq 0$
  - Partition:  $x$  is a perfect square and  $x$  is an integer  $> 0$  but not a perfect square
  - Partition:  $x = 0, x = 1, x = 5, x = 16$
  - Partition:  $x$  even,  $x$  odd,  $x \geq 100$

## Include Boundaries in the Partition (1)

---

Bugs often occur at boundaries between subdomains

Some examples:

- 0 is a boundary between positive numbers and negative numbers
- the maximum and minimum values of numeric types, like `int` and `double`
- emptiness (the empty string, empty list, empty array) for collection types
- the first and last element of a collection

# Include Boundaries in the Partition (2)

---

- Why do bugs often happen at boundaries?
  - One reason is that programmers often make **off-by-one mistakes** (like writing `<=` instead of `<`, or initializing a counter to `0` instead of `1`)
  - Another is that some boundaries may need to be handled as special cases in the code
  - Another is that boundaries may be places of discontinuity in the code's behavior  
When an `int` variable grows beyond its maximum positive value, for example, it abruptly becomes a negative number
- It's important to ***include boundaries*** as subdomains in your partition, so that you're choosing an input from the boundary

## Example 3: max() version 2 (1)

---

Let's redo  $\text{max} : \text{int} \times \text{int} \rightarrow \text{int}$ .

Partition into:

- relationship between a and b
  - $a < b$
  - $a > b$
- value of a
  - $a = 0$
  - $a < 0$
  - $a > 0$
  - $a = \text{minimum integer}$
  - $a = \text{maximum integer}$
- value of b
  - $b = 0$
  - $b < 0$
  - $b > 0$
  - $b = \text{minimum integer}$
  - $b = \text{maximum integer}$

## Example 3: max() version 2 (2)

---

Now let's pick test values that cover all these classes:

- (1, 2) covers  $a < b$ ,  $a > 0$ ,  $b > 0$
- (-1, -3) covers  $a > b$ ,  $a < 0$ ,  $b < 0$
- (0, 0) covers  $a = b$ ,  $a = 0$ ,  $b = 0$
- (Integer.MIN\_VALUE, Integer.MAX\_VALUE)  
covers  $a < b$ ,  $a = \text{minint}$ ,  $b = \text{maxint}$
- (Integer.MAX\_VALUE, Integer.MIN\_VALUE)  
covers  $a > b$ ,  $a = \text{maxint}$ ,  $b = \text{minint}$

## In-Class Quiz 2

---

- Consider the following function and values:

```
/**  
 * @param winsAndLosses is a string of length at most 5 consisting of 'W' or 'L' characters  
 * @return the fraction of characters in winsAndLosses that are 'W'  
 */  
public static double winLossRatio(String winsAndLosses)
```

- (i) ""      (ii) "LLLLL"      (iii) "WLWL"      (iv) "WwwwWW"      (v) "xxxxx"

- Which are appropriate **boundary values** for testing this function ?

- (i)
- (ii), (iv), (v)
- (i), (ii), (iv)
- (i), (ii), (iv), (v)

# Two Extremes for Covering the Partition (1)

---

After partitioning the input space,  
we can choose how exhaustive we want the test suite to be:

## 1. Full Cartesian product

*Every legal combination* of the partition dimensions is covered by one test case.

This is what we did for the *BigInteger multiply* example,  
and it gave us  $7 \times 7 = 49$  test cases.

For the *max* example that included boundaries, which has three dimensions with 3 parts, 5 parts, and 5 parts respectively, it would mean up to  $3 \times 5 \times 5 = 75$  test cases.

In practice not all of these combinations are possible, however.

For example, there's no way to cover the combination  $a < b$ ,  $a = 0$ ,  $b = 0$ ,  
because  $a$  can't be simultaneously less than zero and equal to zero.

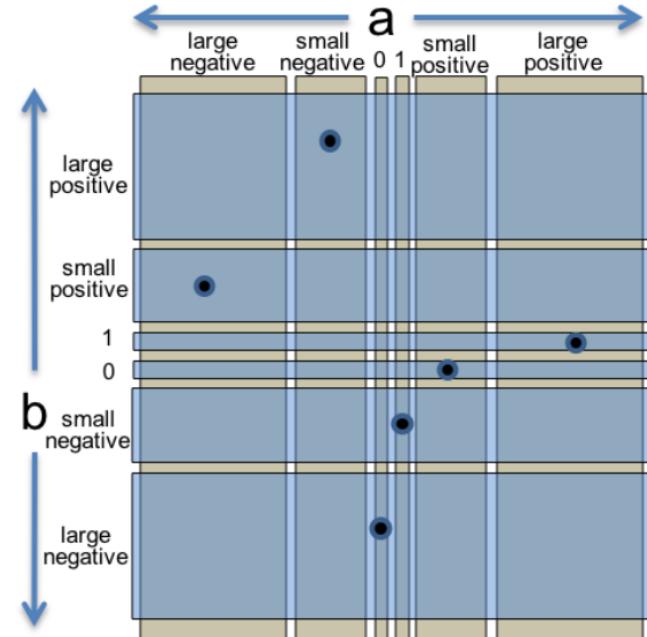
# Two Extremes for Covering the Partition (2)

## 2. Cover each part

Every part of each dimension is covered by at least *one* test case,  
but not necessarily every combination.

With this approach, the test suite for max might be as small as 5 test cases if carefully chosen.

That's the approach we took in `max()` version 2, which allowed us to choose 5 test cases; and 6 test cases for the BigInteger example, as shown on the right.



Often we strike some compromise *between* these two extremes, based on human judgement and caution.

## Part 2: Recursion

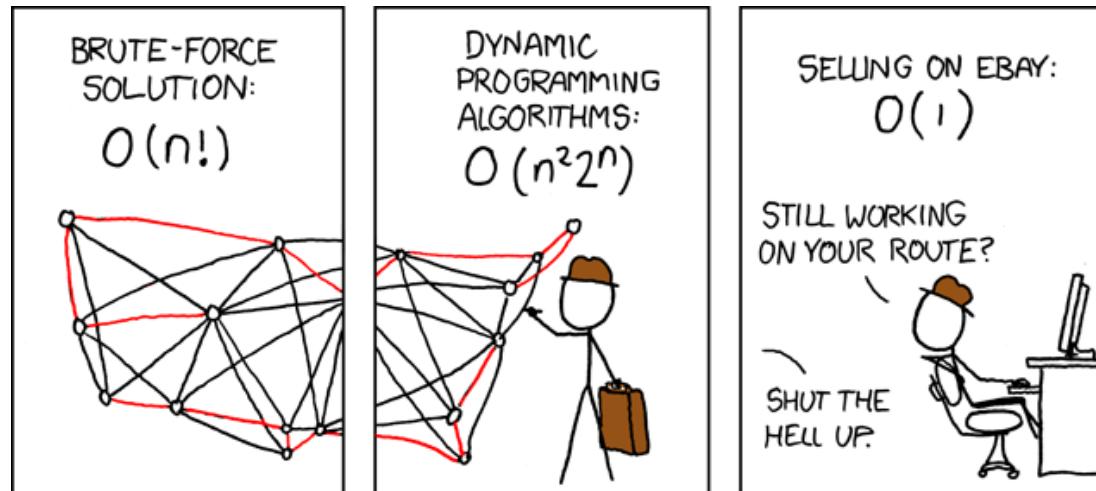
---

- In this part of the lecture, we're going to talk about how to implement a method, focussing on one particular technique: ***recursion***
- Recursion is not appropriate for every problem, but it's an important tool in your software development toolbox (and one that many people scratch their heads over)
- We want you to be comfortable and competent with recursion, because you will encounter it over and over (that's a joke, but it's also true)
- Since you've taken **CSE105**, recursion is not completely new to you, and you have seen and written recursive functions like ***factorial*** and ***fibonacci*** before
  - But don't worry if you forget, we will review them again here...

# Travelling Salesman Problem

- Asks what is the shortest tour that visits all n cities
- Brute-force solution: try all tours
- How many possible tours are there ?  $n!$  possible tours

this number is called  
"n factorial"



# Factorial

---

- Consider writing a method to compute factorial
- We can define factorial in two different ways
- The first one using **product**:  $n! = n \times (n - 1) \times \cdots \times 2 \times 1 = \prod_{k=1}^n k$ 
  - which leads to **iterative** implementation:

```
public static int factorial(int n) {  
    int fact = 1;  
    for (int i = 1; i <= n; i++) {  
        fact = fact * i;  
    }  
    return fact;  
}
```

# Factorial Quiz 1

---

What is  $\text{factorial}(4)$  ?

- a)  $\text{factorial}(4) = 4$
- b)  $\text{factorial}(4) = 12$
- c)  $\text{factorial}(4) = 24$

## Factorial Quiz 2

---

What is  $\text{factorial}(5)$  ?

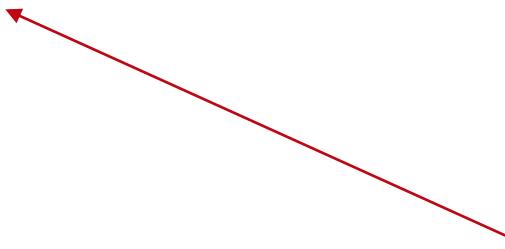
- a)  $\text{factorial}(5) = 5$
- b)  $\text{factorial}(5) = 24$
- c)  $\text{factorial}(5) = 120$

# Factorial Quiz 2

---

What is  $\text{factorial}(5)$  ?

- a)  $\text{factorial}(5) = 5$
- b)  $\text{factorial}(5) = 24$
- c)  $\text{factorial}(5) = 120$



you can compute again:  
 $5 \times 4 \times 3 \times 2 \times 1$   
but have computed:  
 $4 \times 3 \times 2 \times 1$   
from the previous quiz !

so, you just need to compute:  
 $5 \times \text{factorial}(4)$

this brings us to the second  
definition on the next slide...

# Recursive Factorial

---

- The second definition using **recurrence relation**:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n - 1)! \times n & \text{if } n > 0 \end{cases}$$

- which leads to recursive implementation:

```
public static int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    }  
    else {  
        return n * factorial(n-1);  
    }  
}
```

# Recursive Factorial

---

- The second definition using **recurrence relation**:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n - 1)! \times n & \text{if } n > 0 \end{cases}$$

- which leads to recursive implementation:

```
public static int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    }  
    else {  
        return n * factorial(n-1);  
    }  
}
```

But how do we get  
the answer to this  
`factorial(n-1)` ?

# Recursion God



- A good way to think about this, is that you have an all-powerful entity called **Recursion God**:
  - that if you solve a problem of original size, say size n
  - Recursion God will answer **correctly** problems of size **less than** n !

```
public static int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    }  
    else {  
        return n * factorial(n-1);  
    }  
}
```

Now since this `factorial(n-1)` is a problem of size smaller than the original problem, Recursion God will freely answer it ! For example, for n = 5:



`factorial(4) = 24`

# Recursion God



- A good way to think about this, is that you have an all-powerful entity called **Recursion God**:
  - that if you solve a problem of original size, say size  $n$  !
  - Recursion God will answer **correctly** problems of size ***less than***  $n$  !

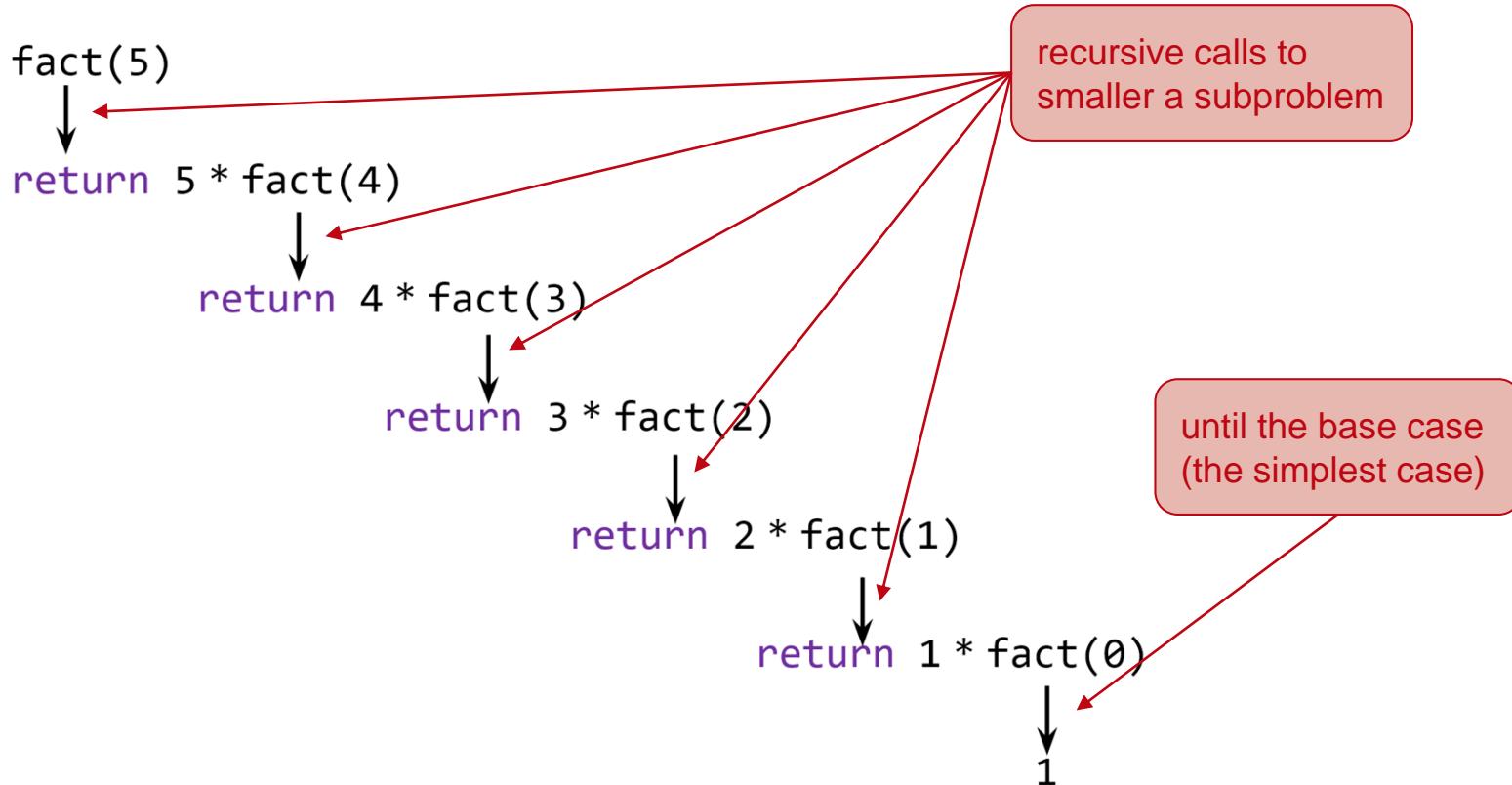
```
public static int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    }  
    else {  
        return n * factorial(n-1);  
    }  
}
```



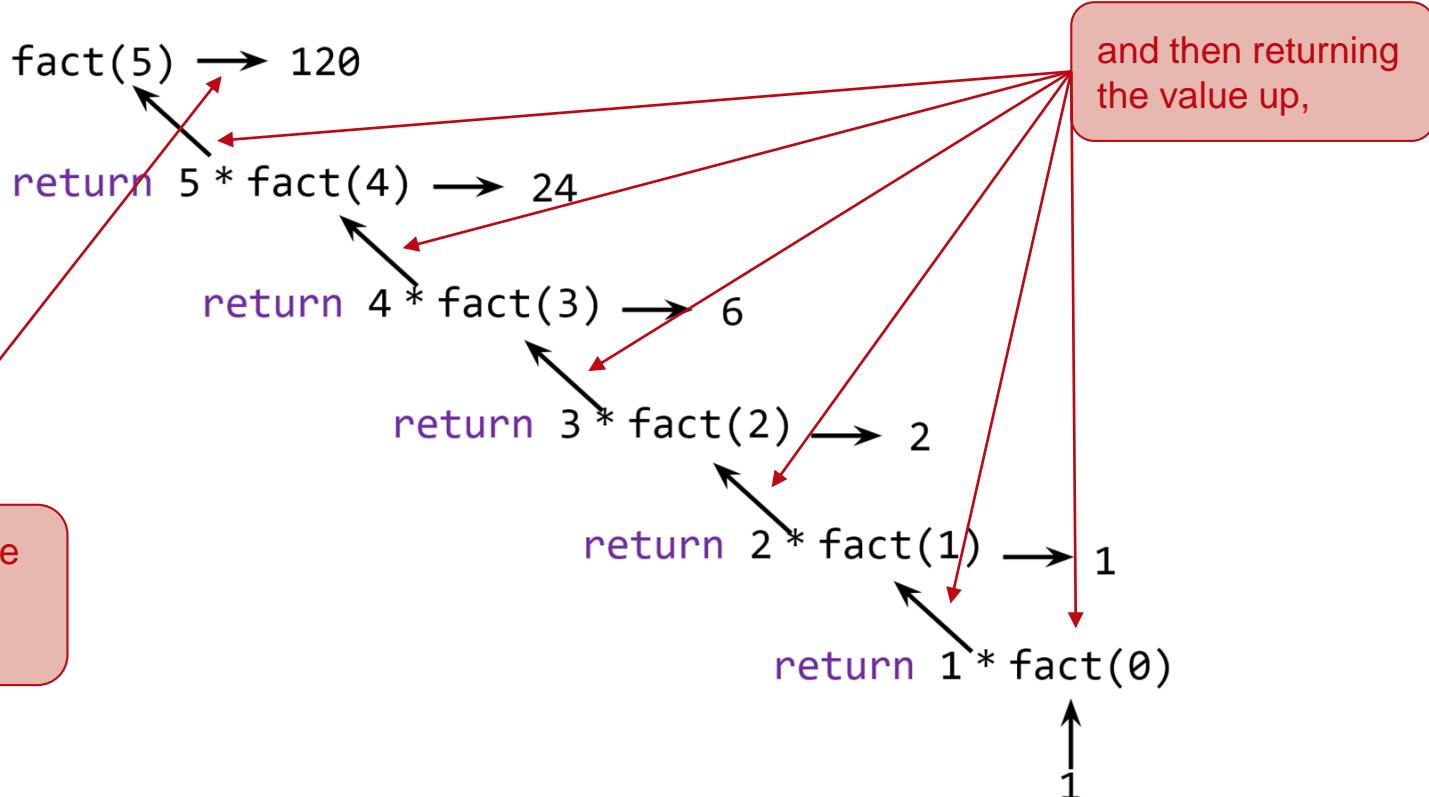
factorial(4) = 24

Then, having the answer from Recursion God, you believe it, and to compute the answer to the original problem, you just multiply his answer with  $n$

# What really happened behind the scene...



# What really happened behind the scene...



# Recipe for Recursion

---



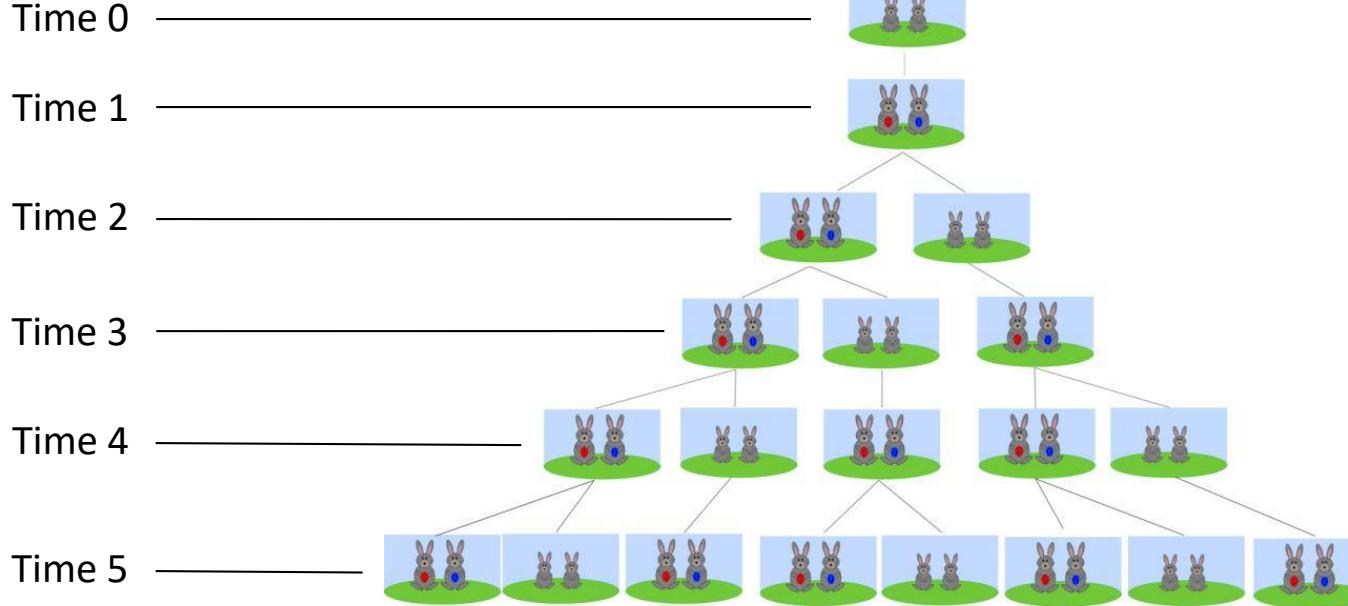
- First, check that the problem can be reduced into ***smaller*** and ***easier*** subproblem(s)
  - that is, the solution to a problem depends on the solution(s) to smaller instance(s) of the *same* problem
- Next, find the **Base Case** :
  - Find the *simplest instance* that can be directly solved
- Finally, compute the **Recursive Step** :
  - Try a couple of cases to see the *recursive relation* among instances
  - Formulate generally for  $n$ , trust Recursion God to solve the smaller instances
  - Make sure to reach the base case

# Recipe for Recursion



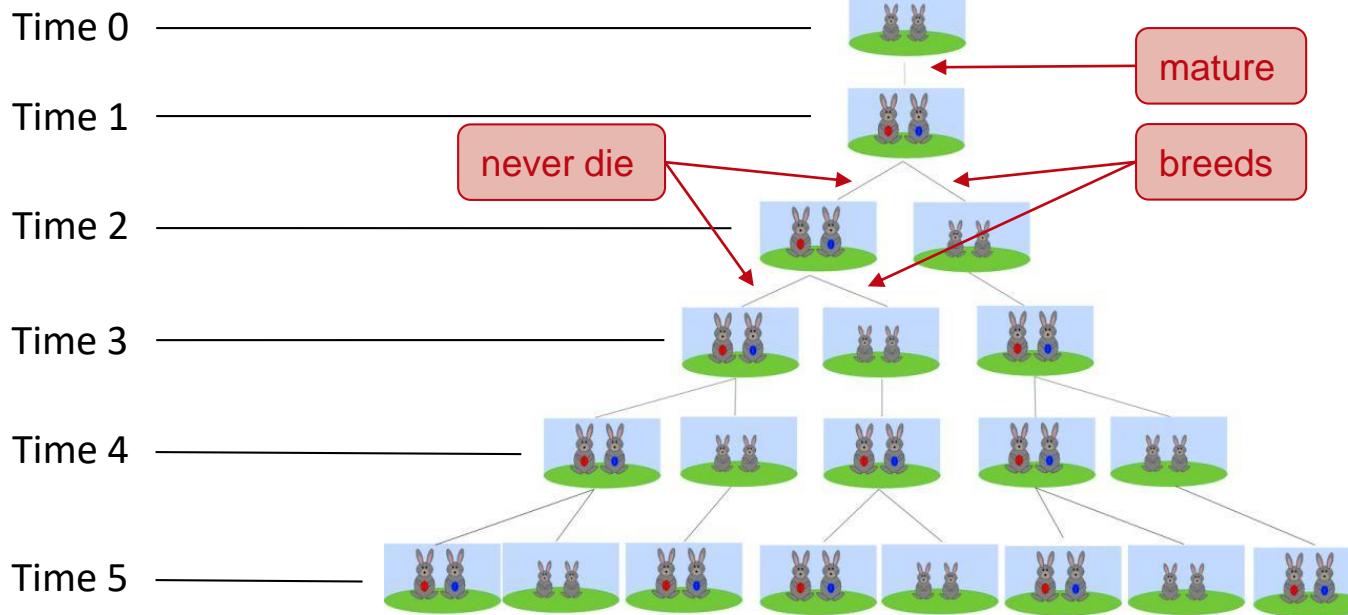
- First, check that the problem can be reduced into *smaller* and *easier* subproblem(s)
    - that is, the solution to a problem depends on the instance(s) of the *same* problem
  - Next, find the **Base Case** :
    - Find the *simplest instance* that can be directly solved
  - Finally, compute the **Recursive Step** :
    - Try a couple of cases to see the *recursive relation* among instances
    - Formulate generally for  $n$ , trust Recursion God to solve the smaller instances
    - Make sure to reach the base case
- $\text{factorial}(n)$  can be computed if we know how to compute  $\text{factorial}(n-1)$ , and  $n-1 < n$
- $\text{factorial}(0)$  is 1
- try factorial of 4, of 5, ...
- given the answer of  $\text{factorial}(n-1)$ , I can compute  $\text{factorial}(n)$  by multiplying it with  $n$
- $n, n-1, n-2, \text{ and so on, will always reach base case} = 0$

# Rabbit Breeding Problem



- Assume a pair of rabbits take one time period to mature and breed, always conceive a male female pair, never die
- How many total pairs of rabbits after time n ?

# Rabbit Breeding Problem



- Assume a pair of rabbits take one time period to mature and breed, always conceive a male female pair, never die
- How many total pairs of rabbits after time n ?

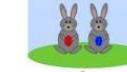
# Fibonacci Number

---

fibo(0) \_\_\_\_\_



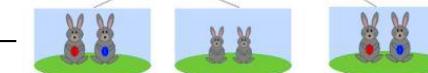
fibo(1) \_\_\_\_\_



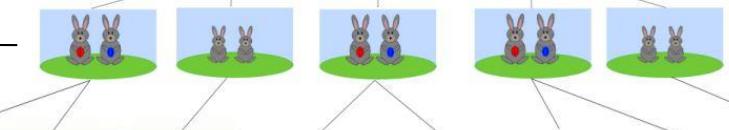
fibo(2) \_\_\_\_\_



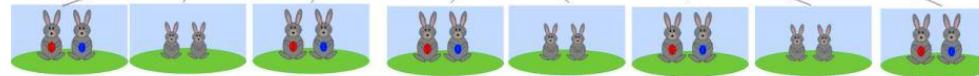
fibo(3) \_\_\_\_\_



fibo(4) \_\_\_\_\_



fibo(5) \_\_\_\_\_



- That total pairs of rabbits at time n is the **fibonacci number**  $\text{fibo}(n)$

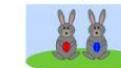
# Fibonacci Number

---

fibo(0) \_\_\_\_\_



fibo(1) \_\_\_\_\_



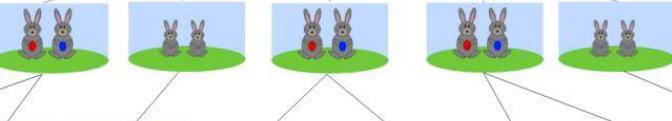
fibo(2) \_\_\_\_\_



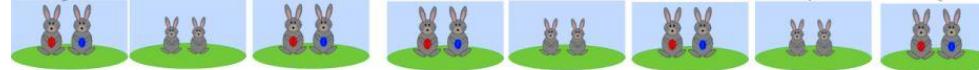
fibo(3) \_\_\_\_\_



fibo(4) \_\_\_\_\_



fibo(5) \_\_\_\_\_



- That total pairs of rabbits at time  $n$  is the **fibonacci number**  $\text{fibo}(n)$ 
  - what is the base case ?
  - what is the recursive step ?

# Fibonacci Number

fibo(0) \_\_\_\_\_



fibo(1) \_\_\_\_\_



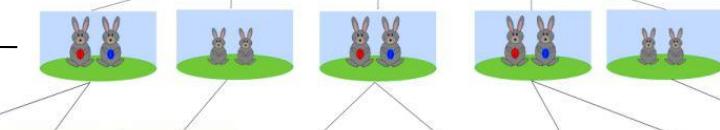
fibo(2) \_\_\_\_\_



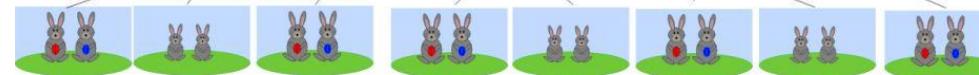
fibo(3) \_\_\_\_\_



fibo(4) \_\_\_\_\_



fibo(5) \_\_\_\_\_



- That total pairs of rabbits at time  $n$  is the **fibonacci number**  $\text{fibo}(n)$

- what is the base case ?
- what is the recursive step ?

the simplest case(s) where  $\text{fibo}(n)$  is just some number

# Fibonacci Number

fibo(0) \_\_\_\_\_

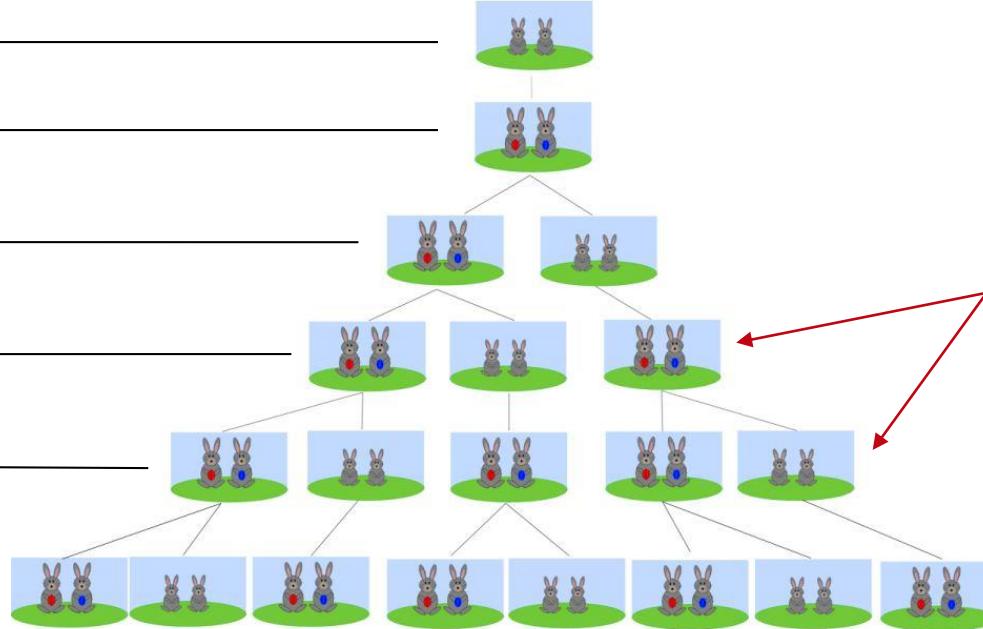
fibo(1) \_\_\_\_\_

fibo(2) \_\_\_\_\_

fibo(3) \_\_\_\_\_

fibo(4) \_\_\_\_\_

fibo(5) \_\_\_\_\_



recursive step:  
knowing fibo of smaller  
sizes, can you compute  
fibo of larger size ?

- That total pairs of rabbits at time  $n$  is the **fibonacci number**  $\text{fibo}(n)$ 
  - what is the base case ?
  - what is the recursive step ?

# Fibonacci Number

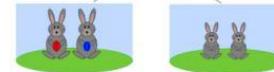
fibo(0) \_\_\_\_\_



fibo(1) \_\_\_\_\_



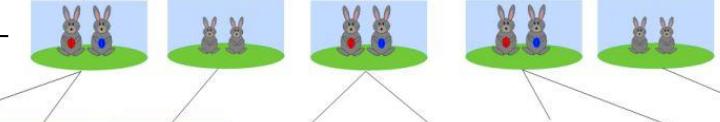
fibo(2) \_\_\_\_\_



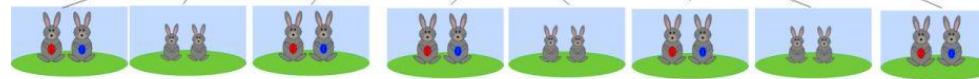
fibo(3) \_\_\_\_\_



fibo(4) \_\_\_\_\_



fibo(5) \_\_\_\_\_



- That total pairs of rabbits at time  $n$  is the **fibonacci number**  $\text{fibo}(n)$ 
  - two bases cases: when  $n = 0$  and  $n = 1$ ,  $\text{fibo}(n) = 1$
  - recursive step:  $\text{fibo}(n) = \text{fibo}(n-1) + \text{fibo}(n-2)$

# Recursive Fibonacci Code

---

```
/**  
 * Compute a Fibonacci number  
 * @param n >= 0  
 * @return the nth Fibonacci number  
 */  
public static int fibo(int n) {  
    // base cases  
    if (n == 0 || n == 1) {  
        return 1;  
    }  
    // recursive step  
    return fibo(n-1) + fibo(n-2);  
}
```

# Sum of Digits

---

- Given a non-negative integer  $n$ , write a method that returns the sum of its digits ***recursively***.

Do **not** use any loops anywhere within your code.

```
/**  
 * Compute the sum of digits of a non-negative int  
 * @param n an integer >= 0  
 * @return the sum of digits  
 */  
public static int sumDigits(int n) {  
    // base case  
  
    // recursive step  
}
```

# Sum of Digits

---

- Given a non-negative integer  $n$ , write a method that returns the sum of its digits ***recursively***.  
Do **not** use any loops anywhere within your code.
- Test cases:
  - $\text{sumDigits}(123) \rightarrow 6$
  - $\text{sumDigits}(99999) \rightarrow 45$
  - $\text{sumDigits}(0) \rightarrow 0$
  - $\text{sumDigits}(1) \rightarrow 1$
  - $\text{sumDigits}(10) \rightarrow 1$
  - $\text{sumDigits}(101010101) \rightarrow 5$

# Sum of Digits

---

- Given a non-negative integer  $n$ , write a method that returns the sum of its digits ***recursively***.

Do **not** use any loops anywhere within your code.

no for, no while, will be penalized even if written in comment

- Test cases:

- sumDigits(123) → 6
- sumDigits(99999) → 45
- sumDigits(0) → 0
- sumDigits(1) → 1
- sumDigits(10) → 1
- sumDigits(101010101) → 5

a good set of test cases, includes corner cases

# Sum of Digits Recursion Recipe: Base Case

---

- What is the base case?
  - What is the simplest case that sumDigits can immediately return?

**stop!** try to think and answer by yourself first!

## Sum of Digits Recursion Recipe: Base Case

---

- What is the base case?
  - What is the simplest case that sumDigits can immediately return?
  - One idea is when n is 0
    - it is easy to check (just  $n==0$ )
    - it is the smallest instance, since n is non-negative by the assumption
    - and the return value is just 0

# Sum of Digits Recursion Recipe: Recursive Step

---

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem sumDigits, how do I use it to compute the answer to original problem?
  - And, how do I compute that smaller instance?
  - Will the base case always be reached?

**stop!** try to think and answer by yourself first!

# Sum of Digits Recursion Recipe: Recursive Step

---

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem `sumDigits`, how do I use it to compute the answer to original problem?
    - Say  $n = 123$ , smaller instance, one digit less, would be  $12$
    - How to compute  $12$ ?
    - Use integer division,  $123 / 10$  is  $12$
    - Given  $\text{sumDigits}(12)$ , how do I get  $\text{sumDigits}(123)$ ?
    - By adding  $3$  to  $\text{sumDigits}(12)$
    - How do I get  $3$  from  $123$ ?
    - Use modulo,  $123 \% 10$  is  $3$
  - Will the base case always be reached?
    - $123 / 10$  is  $12$ ,  $12 / 10$  is  $1$ ,  $1 / 10$  is  $0$  which is the base case

# Recursive Sum of Digits Code

---

- After applying the recipe, we can write (and test, debug) the code

```
/**  
 * Compute the sum of digits of a non-negative int  
 * @param n an integer >= 0  
 * @return the sum of digits  
 */  
public static int sumDigits(int n) {  
    // base case  
    if (n == 0) {  
        return 0;  
    }  
    // recursive step  
    return (n % 10) + sumDigits(n / 10);  
}
```

# Number of Substrings

---

- Given an input string and a non-empty substring `subs`, write a method to compute **recursively** the number of times that `subs` appears in the string. The substrings must **not** overlap and **no** loops used.

```
/**  
 * Compute the number of substrings in an input string  
 * without overlapping  
 * @param input the input string  
 * @param subs the substring, a non-empty string  
 * @return the number of substrings  
 */  
public static int numSubs(String input, String subs) {  
  
    // base case  
  
    // recursive step  
  
}
```

# Number of Substrings

---

- Given an input string and a non-empty substring `subs`, write a method to compute **recursively** the number of times that `subs` appears in the string. The substrings must **not** overlap and **no** loops used.
- Test cases:
  - `numSubs("abcxxxxabc", "abc") → 2`
  - `numSubs("abcdef", "xyz") → 0`
  - `numSubs("aaaaaaaaaa", "a") → 10`
  - `numSubs("aaaaaaaaaa", "aa") → 5`
  - `numSubs("", "a") → 0`
  - `numSubs("a", "a") → 1`

# Number of Substrings Recursion Recipe: Base Case

---

- What is the base case?
  - What is the simplest case that numSubs can immediately return?

**stop!** try to think and answer by yourself first!

# Number of Substrings Recursion Recipe: Base Case

---

- What is the base case?
  - What is the simplest case that numSubs can immediately return?
  - One idea is when input string is strictly shorter than subs
    - it is easy to check, just by `input.length() < subs.length()`
    - it includes the smallest instance, since the input string can be empty, while the substring subs is not
    - and the return value is just 0, since there is no way subs appear in input

# Number of Substrings Recursion Recipe: Recursive Step

---

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem numSubs, how do I use it to compute the answer to original problem?
  - And, how do I compute that smaller instance?
  - Will the base case always be reached?

**stop!** try to think and answer by yourself first!

# Number of Substrings Recursion Recipe: Recursive Step

---

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem numSubs, how do I use it to compute the answer to original problem?
    - Say we check whether subs is the prefix of input
    - There are 2 cases:
      - subs is the prefix of input, which means we find 1 occurrence, so we return  $1 + \text{numSubs}$  of substring of line that starts *after* subs, because the next occurrence cannot overlap
      - subs is not the prefix of input, which means we don't find it this time, so we return  $0 + \text{numSubs}$  of substring of input one char to the right
  - Will the base case always be reached?
    - in both cases, we *always* reduce input length of numSubs, so at some point it will be shorter than subs and reaches the base case

# Recursive Number of Substrings Code

---

- After applying the recipe, we can write (and test, debug) the code

```
/**  
 * Compute the number of substrings in an input string  
 * without overlapping  
 * @param input the input string  
 * @param subs the substring, a non-empty string  
 * @return the number of substrings  
 */  
public static int numSubs(String input, String subs) {  
    // base case  
    if (input.length() < subs.length()) {  
        return 0;  
    }  
    // recursive step  
    if (input.substring(0, subs.length()).equals(subs)) {  
        return 1 + numSubs(input.substring(subs.length()), subs);  
    }  
    return numSubs(input.substring(1), subs);  
}
```

# Recursion + Helper Method

---

- We have learned about Recursion technique
  - Recursion Recipe: Base Case and Recursive Step
- Next, we are going to learn about additional technique called **helper method** that will make our recursive step more *simple* and *elegant*

# Subsequences Problem

---

- We will use the Subsequences Problem as our running example
- Given a word consisting only of letters A-Z or a-z
  - Return all subsequences of word, separated by commas, where a subsequence is a string of letters found in word in the same order that they appear in word
  - For example:  
`subsequences("abc") → "abc,ab,bc,ac,a,b,c,"`
    - Note the trailing comma preceding the empty string "", which is also a valid subsequence

# Subsequence Problem : Direct Recursive Solution

- We can solve it directly using recursion as follows:

```
public static String subseq(String word) {  
    // base case  
    if (word.equals(""))  
        return "";  
    // recursive step  
    char firstLetter = word.charAt(0);  
    String restOfWord = word.substring(1);  
    String subsequencesOfRest = subseq(restOfWord);  
    String result = "";  
    for (String subsequence : subsequencesOfRest.split(",,")) {  
        result += "," + subsequence;  
        result += "," + firstLetter + subsequence;  
    }  
    result = result.substring(1); // remove extra leading comma  
    return result;  
}
```

However, as you can see,  
it looks complicated and  
unclear at the first glance

Is there a better way?

# Helper Method

---

- For the Subsequences Problem, instead of solving it with direct recursive implementation:

```
public static String subseq(String word)
```

one parameter

two parameters

it is easier to create **additional Helper Method**:

```
private static String subseqHelper(String partialSubseq, String word)
```

- Helper method ***introduces another parameter(s)*** to give you more flexibility in doing (more creative) recursion calls

# Subsequence Problem with Helper Method

---

- Idea: we build up a *partial subsequence* using the initial letters of the word  
then use recursive calls to complete that partial subsequence using the remaining letters of the word
- For example, word = "abc"
  - First, select "a" to be in the partial subsequence,  
and recursively extend it with all subsequences of "bc"
  - Then, use "" as the partial subsequence,  
and again recursively extend it with all subsequences of "bc"
- In other words, recursively build up subsequences that,  
either use the first letter 'a' or not

# Subsequence Problem with Helper Method Code (1)

---

```
private static String subseqHelper(String partialSubseq, String word) {  
    // base case  
    if (word.equals(""))  
        return partialSubseq;  
    // recursive step  
    return subseqHelper(partialSubseq + word.charAt(0), word.substring(1))  
        + ","  
        + subseqHelper(partialSubseq, word.substring(1));  
}
```

- *Base case* when all letters in word has been used, partialSubseq is the subsequences
- *Recursive steps* call the ones that either use the first letter of word, or not
- Very simple and elegant!

# Subsequence Problem with Helper Method Code (2)

- Don't forget to implement the original problem:

```
public static String subseq(String word) {  
    // call the recursive helper method  
    return subseqHelper("", word);  
}
```

partial subseq starts with empty string

- Hide the helper method from client; you can overload it by naming it subseq as well

```
private static String subseqHelper(String partialSubseq, String word) {  
    // base case  
    ...  
}
```

# Reverse String

---

- Given a string, return the reverse of that string ***recursively***
- Test cases:
  - `reverseStr("abcde")` → "edcba"
  - `reverseStr("")` → ""
  - `reverseStr("a")` → "a"
  - `reverseStr("ab")` → "ba"

# Reverse String

- Given a string, return the reverse of that string **recursively**
- Let's solve this by using a recursive helper method that also has a second parameter to store the current reversal result (partial result)

```
/**  
 * Reverse the input string.  
 * @param str is the input string.  
 * @return the reverse of str.  
 */  
public static String reverseStr(String str) {  
  
    // call your recursive helper method  
    return reverseStrHelper(str, "");  
  
}
```

in the beginning, the partial result is nothing = empty string

# Reverse String Recursive Helper Method: Base Case

- We are going to remove and reverse the letter of str one by one, so the base case is when there is finally no letter left
  - in that case, we are done, and we just return the reversal result

```
private static String reverseStrHelper(String str, String reverse) {  
    // base case  
    if (str.equals(""))  
        return reverse; ← the partial result is now the full result  
    // recursive step  
    return reverseStrHelper(str.substring(1), str.charAt(0) + reverse);  
}
```

# Reverse String Recursive Helper Method: Recursive Step

- We call recursively the reverseStrHelper on str except the first letter, and append first letter on the partial result
  - e.g. ("abc", "") → ("bc", "a") → ("c", "ba") → ("", "cba") → "cba"

```
private static String reverseStrHelper(String str, String reverse) {  
    // base case  
    if (str.equals(""))  
        return reverse; ← the partial result is now the full result  
    // recursive step  
    return reverseStrHelper(str.substring(1), str.charAt(0) + reverse);  
}
```

# Subset Sum

---

- Given a list of integers, we want to know whether it is possible to choose a subset of some of the integers, such that the integers in the subset adds up to the given sum *recursively*
- Test cases:
  - subsetSum([2, 5, 8], 10) → true
  - subsetSum([2, 5, 8], 15) → true
  - subsetSum([2, 5, 8], 12) → false
  - subsetSum([], 0) → true
  - subsetSum([], 1) → false
  - subsetSum([1], 1) → true

# Subset Sum

---

- Given a list of integers, we want to know whether it is possible to choose a subset of some of the integers, such that the integers in the subset adds up to the given sum *recursively*

```
/**  
 * Decide whether there is a subset in the input list  
 * that adds up to the target sum.  
 * For example, subsetSum([2, 5, 8], 10) → true.  
 * @param list is the input list.  
 * @param sum is the target sum.  
 * @return true iff there is a subset of list that adds to sum.  
 */  
public static boolean subsetSum(List<Integer> list, int sum) {  
  
    // call your recursive helper method  
}
```

# Subset Sum with Helper Method

---

- Rather than looking at the whole list, we consider the part of the list starting at index start and continuing to the end of the list
  - subsetSum can call subsetSumHelper by passing the target sum and the whole list starting at 0

```
private static boolean subsetSumHelper(List<Integer> list, int start, int sum) {  
    // base case  
  
    // recursive step  
}
```

# Subset Sum Recursion Recipe: Base Case

---

- What is the base case?
  - What is the simplest case that subsetSum can immediately return?

**stop!** try to think and answer by yourself first!

# Subset Sum Recursion Recipe: Base Case

---

- What is the base case?
  - What is the simplest case that subsetSum can immediately return?
  - Since we are incrementing start, the base case is when start exceeds the last index of the list, which means the only subset is the empty set
  - It is easy to check, by comparing start with the size of the list
  - There are two cases:
    - when sum is zero  
the empty set sums up to zero, so return true
    - when sum is not zero  
no way we can get that sum, so return false

# Subset Sum Recursion Recipe: Recursive Step

---

- What is the recursive step?
  - How do I construct the "smaller instance"?
  - Knowing answer to a "smaller instance" of the same problem `subsetSum` , how do I use it to compute the answer to original problem?
  - Will the base case always be reached?

**stop!** try to think and answer by yourself first!

# Subset Sum Recursion Recipe: Recursive Step (1)

---

- What is the recursive step?
  - How do I construct the "smaller instance"?
  - There are two cases
    - We include element at start in our subset  
smaller instance = list starting at start+1,  
and target sum becomes sum - element at start
    - We don't include element at start in our subset  
smaller instance = list starting at start+1,  
and target sum remains sum

# Subset Sum Recursion Recipe: Recursive Step (2)

---

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem subsetSum, how do I use it to compute the answer to original problem?
    - If either of those two cases return true,  
it means at least one of the subsets adds up to sum,  
so we return true
    - If neither, no subsets add up to sum, so we return false

## Subset Sum Recursion Recipe: Recursive Step (3)

---

- What is the recursive step?
  - Will the base case always be reached?
    - Initial value of start is always 0
    - In both cases, we always increment start,  
at some point it will be the same as the size of the list

# Recursive Subset Sum Code

---

- After applying the recipe, we can write (and test, debug) the code

```
private static boolean subsetSumHelper(List<Integer> list, int start, int sum) {  
    // base case  
    if (start == list.size()) {  
        if (sum == 0) {  
            return true;  
        }  
        return false;  
    }  
    // recursive step  
    if (subsetSumHelper(list, start+1, sum-list.get(start)))  
        return true;  
    if (subsetSumHelper(list, start+1, sum))  
        return true;  
    return false;  
}
```

# Recursive Subset Sum Code

- After applying the recipe, we can write (and test, debug) the code

```
private static boolean subsetSumHelper(List<Integer> list, int start, int sum) {  
    // base case  
    if (start == list.size()) {  
        if (sum == 0) {  
            return true;  
        }  
        return false;  
    }  
    // recursive step  
    if (subsetSumHelper(list, start+1, sum-list.get(start)))  
        return true;  
    if (subsetSumHelper(list, start+1, sum))  
        return true;  
    return false;  
}
```

A red curly brace groups the entire base case logic: the if statement where start equals list.size(), the nested if statement where sum equals 0, and the return statements for both cases. A red arrow points from a callout box containing the text "or simply return (sum == 0)" to the nested if statement.

or simply  
return (sum == 0)

# Recursive Subset Sum Code

- After applying the recipe, we can write (and test, debug) the code

```
private static boolean subsetSumHelper(List<Integer> list, int start, int sum) {  
    // base case  
    if (start == list.size()) {  
        if (sum == 0) {  
            return true;  
        }  
        or simply  
        return (subsetSum(list, start+1, sum-list.get(start)) || subsetSum(list, start+1, sum))  
    }  
    // recursive step  
    if (subsetSumHelper(list, start+1, sum-list.get(start)))  
        return true;  
    if (subsetSumHelper(list, start+1, sum))  
        return true;  
    return false;  
}
```



# Thank you for your attention !

---

- In this lecture, you have learned:
  - Useful Coding Rules
  - Testing 3
    - Partitioning Input Space, Boundaries
    - Covering Partitions, Choosing Test Suite
  - Recursion
    - Factorial, Fibonacci, Sum of Digits, Number of Substrings
    - Helper Method: Subsequence Problem, Reverse String, Subset Sum
- Please continue to Lecture Quiz 3 and Lab 3:
  - to practice recursion, helper method, and testing with JUnit,
  - to do Lab Exercise 3.1, 3.2, and
  - to do Exercise 3.1 - 3.5



# Advanced Object-Oriented Programming

CPT204 – Lecture 4  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

# CPT204 Advanced Object-Oriented Programming

## Lecture 4

**Testing 4, Linked List 1**

# Welcome !

---

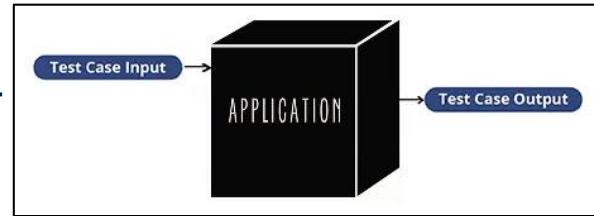
- Welcome to Lecture 4 !
- In this lecture we are going to
  - continue our discussion about Testing
    - how to produce a good test suite
    - when and which part of your code you should test
  - study the mutating or not mutating methods,
    - iteratively or recursively
    - with the running example of our homemade linked list!

## Part 1: Testing 4

---

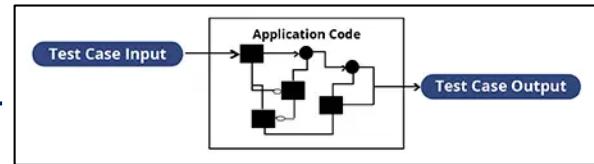
- We are going to continue to learn testing techniques:
  - Black-box vs White-box Testing
  - Unit vs Integration Testing
  - Automated Regression Testing
  - Testing Documents and Coverage

# Black-box and White-box Testing (1)



- Recall that the *specification* is the description of the method's behavior — the types of parameters, type of return value, and constraints and relationships between them
- **Black-box testing** means choosing test cases **only** from the specification, **not** the implementation of the method
  - That's what we've been doing in our lectures and labs so far!
  - We partitioned and looked for corner cases/boundaries in the exercises and assignments, *without* looking at the actual code for these methods

# Black-box and White-box Testing (2)



- **White-box testing** (also called *glass box testing*) means choosing test cases with *knowledge* of how the method is *actually implemented*
  - For example, if the implementation selects different algorithms depending on the input, then you should partition according to those domains
  - Another example, if you know that the implementation keeps an internal cache that remembers the answers to previous inputs, then you should test repeated inputs

# White-box Testing Example

---

- One more example, for the case of `BigInteger.multiply`,
  - when we finally implemented it, we may have decided to represent small integers with `int` values and large integers with a list of decimal digits
  - this decision introduces new boundary values, presumably at `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, and a new partition around them

# White-box Testing Test Cases

---

- When doing white-box testing, you must take care that your test cases don't require specific implementation behavior that isn't specifically called for by the spec
  - For example, if the spec says "throws an exception if the input is poorly formatted," then your test *shouldn't* check specifically for a `NullPointerException` just because that's what the current implementation does
  - The specification in this case allows *any* exception to be thrown, so your test case should likewise be general to preserve the implementor's freedom

# Documenting Your Testing Strategy (1)

---

- Document your test strategy at the top of your test class, for example:

```
/*
 * Testing strategy
 *
 * Partition the inputs as follows:
 * text.length(): 0, 1, > 1
 * start:          0, 1, 1 < start < text.length(),
 *                 text.length() - 1, text.length()
 * text.length()-start: 0, 1, even > 1, odd > 1
 *
 * Include even- and odd-length reversals because
 * only odd has a middle element that doesn't move.
 *
 * Exhaustive Cartesian coverage of partitions.
 */
```

## Documenting Your Testing Strategy (2)

---

- Document how each test case was chosen, including white-box tests:

```
// covers test.length() = 0,  
//           start = 0 = text.length(),  
//           text.length()-start = 0  
@Test public void testEmpty() {  
    assertEquals("", reverseEnd("", 0));  
}  
  
// ... other test cases ...
```

# Coverage (1)

- One way to judge a test suite is to ask how thoroughly it exercises the program
- This notion is called **coverage**

The screenshot shows an IDE interface with the following details:

- Title Bar:** spring-petclinic – PetValidator.java [spring-petclinic]
- Project Tree:** Shows the project structure with various packages and their coverage percentages.
- Code Editor:** Displays the `PetValidator.java` file with code annotations indicating coverage status (green for covered, red for uncovered).
- Code Coverage Table:** A table on the right side provides a detailed breakdown of coverage metrics across different elements.
- Status Bar:** At the bottom, it shows "Tests ignored: 1, passed: 39 (5 minutes ago)" and system information like "51:12 LF UTF-8 4 spaces ⌂ master ⌂ 1".

**Code Editor Content (PetValidator.java):**

```
java - org.springframework.samples.petclinic 100% classes, 93% lines covered
    model 100% classes, 100% lines covered
        BaseEntity 100% methods, 100% lines covered
        NamedEntity 100% methods, 100% lines covered
        Person 100% methods, 100% lines covered
        package-info.java
    owner 100% classes, 94% lines covered
        OwnerRepository
        PetRepository
        Owner 92% methods, 81% lines covered
        OwnerController 100% methods, 100% lines covered
        Pet 100% methods, 95% lines covered
        PetController 100% methods, 96% lines covered
        PetType
        PetTypeFormatter 100% methods, 100% lines covered
        PetValidator 100% methods, 90% lines covered
    VisitController 100% methods, 100% lines covered
    system 100% classes, 77% lines covered
        CacheConfiguration 100% methods, 100% lines covered
        CrashController 0% methods, 50% lines covered
        WelcomeController 0% methods, 50% lines covered
    vet 100% classes, 92% lines covered
        VetRepository
        Specialty
        Vet 80% methods, 83% lines covered
        VetController 100% methods, 100% lines covered
        Vets 100% methods, 100% lines covered
    visit 100% classes, 100% lines covered
    PetClinicApplication 0% methods, 33% lines covered

    @Override
    public void validate(Object obj, Errors errors) {
        Pet pet = (Pet) obj;
        String name = pet.getName();
        // name validation
        if (!StringUtil.hasLength(name)) {
            errors.rejectValue("name", "required");
        }
        // type validation
        if (pet.isNew() &amp; pet.getType() == null) {
            errors.rejectValue("type", "required");
        }
        // birth date validation
        if (pet.getBirthDate() == null) {
            errors.rejectValue("birthDate", "required");
        }
    }

    This Validator validates *just* Pet instances.

    @Override
    public boolean supports(Class<?> clazz) {
        return clazz.equals(Pet.class);
    }
}

Tests ignored: 1, passed: 39 (5 minutes ago)
51:12 LF UTF-8 4 spaces ⌂ master ⌂ 1
```

## Coverage (2)

---

- There are three common kinds of coverage:
  - **Statement coverage**: is every statement run by some test case?
  - **Branch coverage**: for every if or while statement in the program, are both the true and the false direction taken by some test case?
  - **Path coverage**: is every possible combination of branches — every path through the program — taken by some test case?
- Branch coverage is stronger (requires more tests to achieve) than statement coverage, and path coverage is stronger than branch coverage
- In industry, 100% statement coverage is a common goal, but even that is rarely achieved due to unreachable defensive code (like “should never get here” assertions)
- However, 100% path coverage is infeasible, requiring exponential-size test suites to achieve

## Coverage (3)

---

- A standard approach to testing is to add tests until the test suite achieves adequate statement coverage, that is, so that every reachable statement in the program is executed by *at least one test case*
- In practice, statement coverage is usually measured by *a code coverage tool*, which counts the number of times each statement is run by your test suite  
With such a tool, white box testing is easy; you just measure the coverage of your black box tests, and add more test cases until all important statements are logged as executed
- You can use a code coverage tool for IntelliJ, for example:
  - IntelliJ IDEA Code Coverage Runner  
<https://www.jetbrains.com/help/idea/code-coverage.html>
  - EMMA or JaCoCo

# Unit Testing and Integration Testing (1)

---

- A well-tested program will have tests for every individual module (where a module is a method or a class) that it contains
- A test that tests an individual module, in *isolation* if possible, is called a **unit test**
- Testing modules in isolation leads to much easier debugging
  - When a unit test for a module fails, you can be more confident that the bug is found *in that module*, rather than anywhere in the program

# Unit Testing and Integration Testing (2)

---

- The opposite of a unit test is **an integration test**, which tests a combination of modules, or even the entire program
- If all you have are integration tests, then when a test fails, you have to hunt for the bug
  - it might be anywhere in the program
- Integration tests are still important, because a program can fail at *the connections between modules*
  - for example, one module may be expecting different inputs than it's actually getting from another module
- But if you have a thorough set of unit tests that give you confidence in the correctness of individual modules, then you'll have much less searching to do to find the bug

# Unit Testing and Integration Testing Example (1)

---

- As an example, suppose you're building a web search engine
- Two of your modules are `getWebPage()`, which downloads web pages, and `extractWords()`, which splits a page into its component words:

```
/** @return the contents of the web page downloaded from url
 */
public static String getWebPage(URL url) {...}

/** @return the words in string s, in the order they appear,
     where a word is a contiguous sequence of
     non-whitespace and non-punctuation characters
 */
public static List<String> extractWords(String s) { ... }
```

## Unit Testing and Integration Testing Example (2)

- Those two methods are used by another module `makeIndex()` as part of the web crawler that makes the search engine's index:

```
/** @return an index mapping a word to the set of URLs
     containing that word, for all webpages in the input set
 */
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {
    ...
    for (URL url : urls) {
        String page = getWebPage(url);
        List<String> words = extractWords(page);
        ...
    }
    ...
}
```

## Unit Testing and Integration Testing Example (3)

---

- In our test suite, we would want:
  - unit tests just for `getWebPage()` that test it on various URLs
  - unit tests just for `extractWords()` that test it on various strings
  - unit tests for `makeIndex()` that test it on various sets of URLs

# Unit Testing and Integration Testing Example (4)

---

- One mistake that programmers sometimes make is writing test cases for `extractWords()` in such a way that the test cases depend on `getWebPage()` to be correct
- It's better to think about and test `extractWords()` *in isolation*, and partition it
  - Using test partitions that involve web page content might be reasonable, because that's how `extractWords()` is actually used in the program
  - But don't actually call `getWebPage()` from the test case, because `getWebPage()` may be buggy!
  - Instead, store web page content as a literal string, and pass it directly to `extractWords()`
  - That way you're writing an isolated unit test, and if it fails, you can be more confident that *the bug is in the module it's actually testing*, `extractWords()`

# Unit Testing and Integration Testing Example (5)

---

- Note that the unit tests for `makeIndex()` can't easily be isolated in this way
- When a test case calls `makeIndex()`, it is testing the correctness of ***not only*** the code inside `makeIndex()`, ***but also*** all the methods called by `makeIndex()`
  - If the test fails, the bug might be in any of those methods
  - That's why we want separate tests for `getWebPage()` and `extractWords()`, to increase our confidence in those modules individually and **localize** the problem to the `makeIndex()` code that connects them together

```
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {  
    ...  
    for (URL url : urls) {  
        String page = getWebPage(url);  
        List<String> words = extractWords(page);  
        ...  
    }  
}
```

# Unit Testing and Stub

---

- Isolating a higher-level module like `makeIndex()` is possible if we write **stub versions** of the modules that it calls
  - For example, a stub for `getWebPage()` wouldn't access the internet at all, but instead would return mock web page content no matter what URL was passed to it
- A stub for a class is often called a **mock object**
- Stubs are an important technique when building large systems

# Automated Testing

---

- Nothing makes tests easier to run, and more likely to be run, than complete automation
- **Automated testing** means running the tests and checking their results automatically
  - A test driver should ***not*** be an interactive program that prompts you for inputs and prints out results for you to manually check
  - Instead, a test driver should ***invoke the module itself*** on fixed test cases and automatically check that the results are correct
  - The result of the test driver should be either “all tests OK” or “these tests failed: ...”
  - A good testing framework, like **JUnit**, helps you build automated test suites
- Note that automated testing frameworks, like the one we use: **JUnit**, make it easy to run the tests, but *you still have to come up with good test cases yourself*
  - *Automatic test generation* is a hard problem, still a subject of active computer science research

# Regression Testing (1)

---

- Once you have test automation, it's very important to *rerun your tests when you modify your code*
  - This prevents your program from regressing — introducing other bugs when you fix new bugs or add new features
  - Running *all* your tests after *every* change is called **regression testing**
- Whenever you find and fix a bug, take the input that elicited the bug and add it to your automated test suite as a test case, called a regression test
  - This helps to populate your test suite with good test cases.
  - Remember that a test is good if it elicits a bug — and every regression test did find a bug in one version of your code!
  - Saving regression tests also protects against reversions that reintroduce the bug
  - The bug may be an easy error to make, since it happened once already

## Regression Testing (2)

---

- This idea also leads to *test-first debugging*:
  - When a bug arises, immediately write a test case for it that elicits it, and *immediately add it* to your test suite
  - Once you find and fix the bug, all your test cases will be passing, and you'll be done with debugging and have a regression test for that bug
- In practice, these two ideas, automated testing and regression testing, are almost always used in combination
- Regression testing is only practical if the tests can be run often, automatically
- Conversely, if you already have automated testing in place for your project, then you might as well use it to prevent regressions
- So **automated regression testing** is a *best-practice* of modern software engineering

## In-class Quiz

---

- Which of the following are good times to re-run all your JUnit tests?
- Select one or more:
  - after rewriting a correct method to make it faster
  - when using a code coverage tool
  - after an attempt to fix a bug
  - before submitting your code to LM Autograder

## Part 2: Recursive Linked List

---

- The codes of the next part of the lecture can be found in **lecture4.zip**
- In addition, we are also going to use the **Java Visualizer** plugin,  
installed in the first week
  - Read and follow Lab0.pdf if you still have not done so...

# My Linked List (1)

- As a running example for our next topics, let us *build a linked list from scratch*
  - you can find the lecture codes in LMO

```
MyList1.java X
1 ► public class MyList1 {
2     private int value; ←
3     private MyList1 next; ←
4
5 ►     public static void main(String[] args) {
6         MyList1 list = new MyList1();
7         list.value = 5;
8         list.next = null;
9
10        list.next = new MyList1();
11        list.next.value = 2;
12
13        list.next.next = new MyList1();
14        list.next.next.value = 10;
15    }
16 }
```

we start with the simplest one  
MyList1 has only 2 member variables / instance variables

an integer value, which is the data we wish to store inside

and a reference to another MyList1 object  
(people also call this pointer or address to MyList1 object)

set them to be private so they can only be accessed and modified from inside class

# My Linked List (2)

- As a running example for our next topics, let us *build a linked list from scratch*
  - you can find the lecture codes in LMO

```
MyList1.java
1  public class MyList1 {
2      private int value;
3      private MyList1 next;
4
5  public static void main(String[] args) {
6      MyList1 list = new MyList1();
7      list.value = 5;
8      list.next = null;
9
10     list.next = new MyList1();
11     list.next.value = 2;
12
13     list.next.next = new MyList1();
14     list.next.next.value = 10;
15 }
16 }
```

turns out those two are enough to implement a linked list!

first, we create our list, set its value, set reference to null since no other object follows

if we want to store another int, we create a new object, and store its reference in list.next

and so on ...  
can we do better than writing list.next.next.next ?

# Java Visualizer (1)

- But first, let me introduce you to a way to help us understand what happened

The screenshot shows a Java code editor with a file named `MyList1.java`. The code defines a class `MyList1` with a private integer `value` and a private `MyList1` reference `next`. The `main` method creates a new `MyList1` object, sets its `value` to 5, and sets its `next` to `null`. It then creates two more `MyList1` objects and links them together, setting the `value` of the second to 2 and the third to 10. The code is numbered from 1 to 16. A red arrow points from the text "first, run the code" to the start of the `public static void main` line. Another red arrow points from the text "next, set a breakpoint by clicking on this area a red circle will appear" to the line where the variable `list` is assigned.

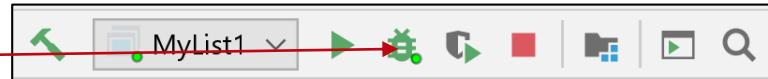
```
1  public class MyList1 {  
2      private int value;  
3      private MyList1 next;  
4  
5      public static void main(String[] args) {  
6          myList1 list = new myList1();  
7          list.value = 5;  
8          list.next = null;  
9  
10         list.next = new myList1();  
11         list.next.value = 2;  
12  
13         list.next.next = new myList1();  
14         list.next.next.value = 10;  
15     }  
16 }
```

first, run the code  
you notice that nothing happens

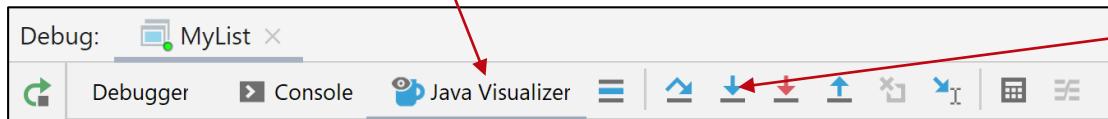
next, set a breakpoint  
by clicking on this area  
a red circle will appear

## Java Visualizer (2)

then click Debug  
on your upper right corner



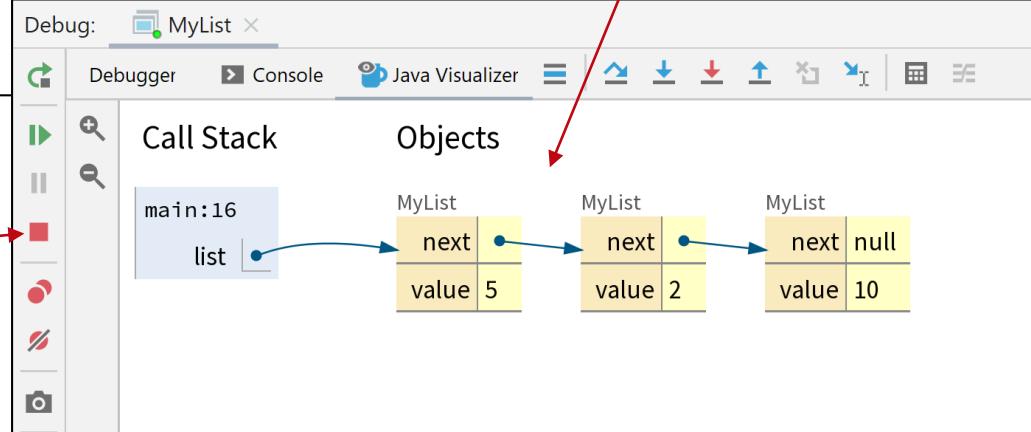
then click Java Visualizer  
below to the left



and click Step Into  
to execute your code  
line-by-line

# Java Visualizer (3)

```
MyList1.java ×  
1 ► public class MyList1 {  
2     private int value;  
3     private MyList1 next;  
4  
5 ►     public static void main(String[] args) {    args: {}  
6     myList1 list = new MyList1();  List: MyList1@790  
7     list.value = 5;  
8     list.next = null;  
9  
10    list.next = new MyList1();  
11    list.next.value = 2;  List: MyList1@790  
12  
13    list.next.next = new MyList1();  
14    list.next.next.value = 10;  
15 }  
16 }
```



click this red box  
to stop the debugger  
and then continue coding

this blue line indicates  
the next line to be executed  
when you click Step Into

you will see a visualization  
of your linked list !

# My Linked List 2 with Constructor

- Here's our second try, this time with a constructor

```
MyList1.java x myList2.java x
public class MyList2 {
    private int value;
    private MyList2 next;
}

public MyList2(int value, MyList2 next) {
    this.value = value;
    this.next = next;
}

public static void main(String[] args) {
    MyList2 list = new MyList2( value: 10, next: null);
    list = new MyList2( value: 2, list);
    list = new MyList2( value: 5, list);
}
```

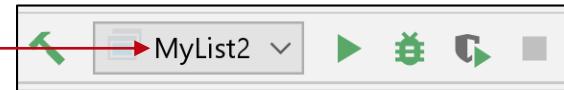
you have learned about constructor in CSE105, it initializes the instance variables

we build the same list as before, **from the end**

each time we update the reference to next to list

and then we update the reference to list

visualize it, change to MyList2 before Debug



# My Linked List 3 with Size: Iterative

- Next, we will equip our linked list with size and get methods

```
/*
 * @return the size of the MyList iteratively.
 */
public int iterSize() {
    MyList3 p = this; ←
    int size = 0;
    while (p != null) {
        size += 1; ←
        p = p.next;
    }
    return size;
}
```

let's first do this using iteration

we use pointer p to the current *MyList3* object

we keep looping, moving the pointer, and increment size until we point to null

**stop!** can you think first, how to implement size *recursively*? what is the base case? what is the recursive step?

# My Linked List 3 with Size: Recursive

- Next, we will equip our linked list with size and get methods

```
/**  
 * @return the size of the myList recursively.  
 */  
public int recSize() {  
    // base case  
    if (next == null) {  
        return 1;  
    }  
    // recursive step  
    return 1 + this.next.recSize();  
}
```

now let's do this using recursion

when next is null, our list only stores one value

otherwise, given the size of the next part of the list, we add 1 to it

you can always use the visualizer and running the code step-by-step to help you understand what's going on

# My Linked List 3 with Get: Recursive

- Next, we will equip our linked list with size and get methods

```
/**  
 * @param i is a valid index of myList.  
 * @return the ith value of this myList.  
 */  
  
public int get(int i) {  
    // base case  
    if (i == 0) {  
        return value;  
    }  
    // recursive step  
    return next.get(i - 1);  
}
```

we implement get recursively,  
to return the  $i$ th value of list,  
assume it exists

if value can be found here,  
(at index 0), return it

otherwise, get the  $(i-1)$ th  
value from the remaining list

as an exercise, you try  
implement get *iteratively*.  
again, use visualizer to help  
you understand this.

# My Linked List 3 Additional Methods: ofEntries

- We also equip it with two additional methods for *testing purposes*
  - we will cover this in future lectures, you **don't** have to understand it now

```
/**  
 * @param args is a variable number of integers.  
 * @return a new MyList containing the integers in args.  
 */  
  
public static MyList3 ofEntries(Integer... args) {  
    MyList3 result, p;  
    if (args.length > 0) {  
        result = new MyList3(args[0], next: null);  
    } else {  
        return null;  
    }  
    int k;  
    for (k = 1, p = result; k < args.length; k += 1, p = p.next) {  
        p.next = new MyList3(args[k], next: null);  
    }  
    return result;  
}
```

this method is used to create *MyList3* objects by specifying the values as sequence of integers

for example, to create [1, 2, 3],  
*MyList3* list =  
*MyList3.ofEntries(1, 2, 3);*

# My Linked List 3 Additional Methods: equals

- We also equip it with two additional methods for *testing purposes*
  - we will cover this in future lectures, you **don't** have to understand it now

```
/**  
 * @param l is a myList object.  
 * @return true iff l is a myList object containing the same sequence of  
 * integers as this.  
 */  
  
public boolean equals(Object l) {  
    if (!(l instanceof myList3)) {  
        return false;  
    }  
    myList3 list = (myList3) l;  
    myList3 p;  
    for (p = this; p != null && list != null; p = p.next, list = list.next) {  
        if (p.value != list.value) {  
            return false;  
        }  
    }  
    if (p != null || list != null) {  
        return false;  
    }  
    return true;  
}
```

this method is used to compare two myList3 objects, will return true iff their values are the same

for example,  
list1.equals(list2) is true iff list1 and list2 are myList3 objects containing same integers

it is used by assertEquals()  
to compare two myList3 objects

# Thank you for your attention !

---

- In this lecture, you have learned:
  - Testing 4
    - White Box vs Black Box, Unit vs Integration, Automated Regression
    - Testing Documents and Coverage
  - Linked List 1
    - Creating your own Linked List
    - Equipping it with methods *iteratively* and *recursively*
- Please do Lecture Quiz 4, and continue to Lab 4:
  - to add more methods to the linked list *iteratively* and *recursively*,
  - to do Lab Exercise 4.1 - 4.4, and
  - to do Exercise 4.1 - 4.4

MyList3 is now MyList in Lab 4



# Advanced Object-Oriented Programming

CPT204 – Lecture 5  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming

### Lecture 5

# Specification, Linked List 2

# Welcome !

---

- Welcome to Lecture 5 !
- In this lecture we are going to
  - continue to learn about specification
    - preconditions and postconditions
    - testing against specs
  - continue improving our homemade linked list, while learning about
    - inner/nested class
    - caching
    - invariants
    - constant-time methods
    - generics

# Part 1: Specification

---

- This week, we are going to continue to discuss specification, which we have seen every week in the **javadoc**
- A **specification** is like a contract for part of your program
  - saying what it can count on from the rest of the program,
  - and what it's expected to do in return

# A Contract between Parties

---

- The specification acts as a contract:
  - **The implementer** is responsible for meeting the contract, and
  - **A client** that uses the method can rely on the contract
  - In fact, we'll see that like real legal contracts, specifications place demands on **both parties**: when the specification has a *precondition*, the client has *responsibilities* too!
- We'll look at the role played by specifications of methods
  - We'll discuss what preconditions and postconditions are, and what they mean for the implementer and the client of a method
  - We'll also talk about how to use **exceptions**, an important language feature found in Java which allows us to make a method's interface safer from bugs and easier to understand

# Why Specification?

---

- Many of the nastiest bugs in programs arise because of misunderstandings about behavior at the interface between two pieces of code
  - Although every programmer has specifications in mind, not all programmers write them down; as a result, different programmers on a same team have different specifications in mind
  - When the program fails, it's hard to determine where the error is. Precise specifications in the code let you apportion blame (to code fragments, not people!), and can spare you the agony of puzzling over where a fix should go
- Specifications are good for the client of a method because *they spare the task of reading code*
  - for example, take a look at some of the standard Java specs
  - compare them to the source code that implements them

# Example of Reading Spec vs Actual Code (1)

- Here's an example of one method from BigInteger:

Specification from the API documentation:

**add**

public BigInteger add(BigInteger val)

Returns a BigInteger whose value is (this + val).

**Parameters:**

val - value to be added to this BigInteger.

**Returns:**

this + val

Method body from Java 8 source:

```
if (val.signum == 0)
    return this;
if (signum == 0)
    return val;
if (val.signum == signum)
    return new BigInteger(add(mag, val.mag), signum);

int cmp = compareMagnitude(val);
if (cmp == 0)
    return ZERO;
int[] resultMag = (cmp > 0 ? subtract(mag, val.mag)
                           : subtract(val.mag, mag));
resultMag = trustedStripLeadingZeroInts(resultMag);

return new BigInteger(resultMag, cmp == signum ? 1 : -1);
```

## Example of Reading Spec vs Actual Code (2)

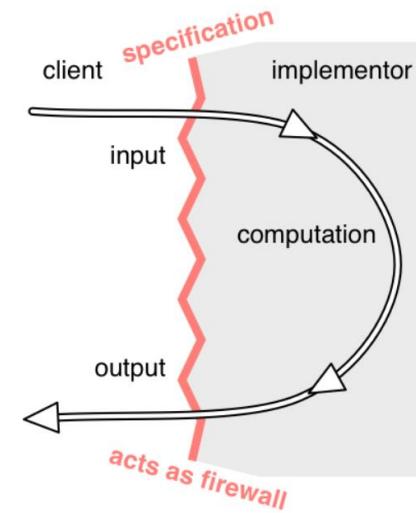
---

- The spec for BigInteger.add is straightforward for clients to understand, and if we have questions about corner cases, the BigInteger class provides additional human-readable documentation
- If all we had was the code, we'd have to read through the BigInteger constructor, compareMagnitude, subtract, and trustedStripLeadingZeroInts just as a starting point

# Spec and Implementer

---

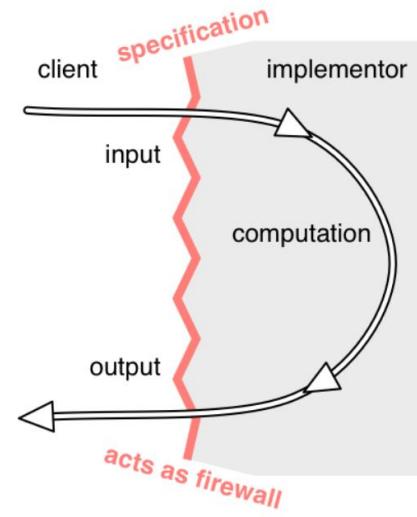
- Specifications are good for the implementer of a method because they give the implementer *freedom to change* the implementation *without* telling clients
- Specifications can make code *faster*, too
  - We'll see that using a weaker specification can rule out certain states in which a method might be called
  - This restriction on the inputs might allow the implementer to skip an expensive check that is no longer necessary and use a more efficient implementation



# Spec and Client

---

- The contract acts as a **firewall** between the client and the implementer
  - It shields the client from the details of the *workings* of the unit — you don't need to read the source code of the procedure if you have its specification
  - And it shields the implementer from the details of the *usage* of the unit
  - The implementer doesn't have to ask every client how they plan to use the unit
- This firewall results in **decoupling**, allowing the code of the unit and the code of a client to be *changed independently*, so long as the changes respect the specification — each obeying its obligation



# Behavioral Equivalence (1)

---

- Consider these two methods: are they the same or different?

```
static int findFirst(int[] arr, int val) {  
    for (int i = 0; i < arr.length; i++) {  
        if (arr[i] == val) return i;  
    }  
    return arr.length;  
}  
  
static int findLast(int[] arr, int val) {  
    for (int i = arr.length -1 ; i >= 0; i--) {  
        if (arr[i] == val) return i;  
    }  
    return -1;  
}
```

## Behavioral Equivalence (2)

---

- Of course the code is different, so in that sense they are different; and we've given them different names, just for the purpose of discussion
  - To determine behavioral equivalence, our question is *whether we could substitute one implementation for the other*
- Not only do these methods have different code, they actually have different behavior:
  - when `val` is missing, `findFirst` returns the length of `arr` and `findLast` returns -1
  - when `val` appears twice, `findFirst` returns the lower index and `findLast` returns the higher

## Behavioral Equivalence (3)

---

- But when `val` occurs at exactly one index of the array,  
the two methods ***behave the same***: they both return that index
  - It may be that clients never rely on the behavior in the other cases:  
whenever they call the method, they will be passing in an arr with exactly  
one element `val`
  - For such clients, these two methods are the same, and we could switch  
from one implementation to the other without issue!

## Behavioral Equivalence (4)

---

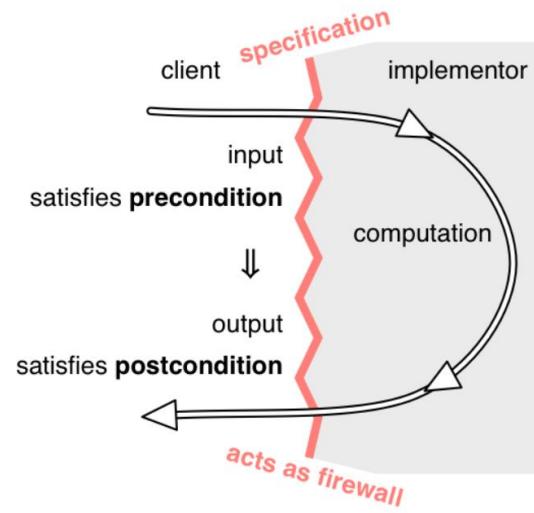
- The notion of equivalence is in the eye of the beholder — that is, *the client*
  - In order to make it possible to substitute one implementation for another, and to know when this is acceptable, we need a specification that states exactly what the client depends on
  - In this case, our specification might be:

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

# Specification Structure (1)

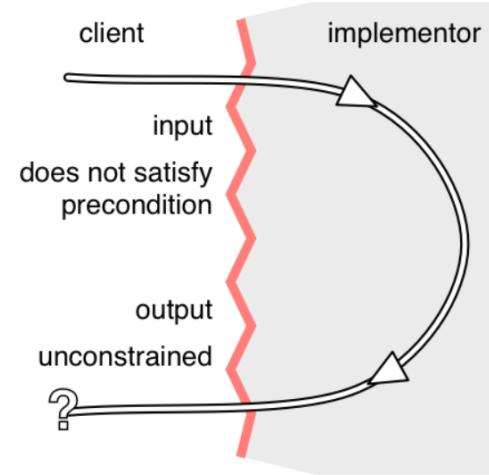
---

- A specification of a method consists of two clauses:
  - *a precondition*, indicated by the keyword **requires**
  - *a postcondition*, indicated by the keyword **effects**
- The precondition is an obligation on the client (i.e., the caller of the method) — it's a condition over the state in which the method is invoked
- The postcondition is an obligation on the implementer of the method
- If the precondition holds for the invoking state, the method is obliged to obey the postcondition, by returning appropriate values, throwing specified exceptions, modifying or not modifying objects, and so on



## Specification Structure (2)

- The overall structure is a logical implication:  
*if the precondition **holds** when the method is called,  
then the postcondition **must hold** when the method  
completes*
- If the precondition **does not hold** when the method is called, the implementation is **not bound** by the postcondition
  - It is free to **do anything**, including not terminating, throwing an exception, returning arbitrary results, making arbitrary modifications, etc



# In-Class Quiz 1

---

- Given the following specification :

```
static int find(int[] arr, int val)
```

**requires:** val occurs exactly once in arr

**effects:** returns index i such that arr[i] == val

select the **legal** behavior that you can then implement `find` with:

- if arr is empty, return 0
- if val occurs twice in arr, set all values in arr to zero, then throw an exception
- if val does not occur in arr, pick random index, set value at index to val, return the index
- if arr[0] is val, continue search, if found another return the index; otherwise return 0

# Specifications in Java (1)



- Some languages (notably Eiffel) incorporate preconditions and postconditions as a fundamental part of the language
  - as expressions that the runtime system (or even the compiler) can automatically check to enforce the contracts between clients and implementers
- Java does *not* go quite so far, but its static type declarations *are* effectively part of the precondition and postcondition of a method, a part that is automatically checked and enforced by the compiler
  - The rest of the contract — the parts that we can't write as types — must be described in **a comment** preceding the method, and generally depends on *human* to check it and guarantee it

## Specifications in Java (2)

---

- Java has a convention for documentation comments, in which parameters are described by `@param` clauses and results are described by `@return` and `@throws` clauses
  - Put the preconditions into `@param` where possible
  - Put postconditions into `@return` and `@throws`

# Specifications in Java (3)

---

- A specification like this:

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

might be rendered in Java like this:

```
/**  
 * Find a value in an array.  
 * @param arr array to search, requires that val occurs exactly once  
 *           in arr  
 * @param val value to search for  
 * @return index i such that arr[i] = val  
 */  
static int find(int[] arr, int val)
```

## In-Class Quiz 2

---

- Given the following specification :

```
static boolean isPalindrome(String word)
```

**requires:** word contains only alphanumeric characters

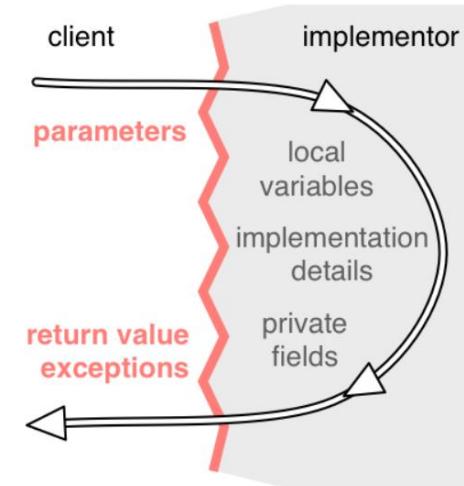
**effects:** returns true if and only if word is a palindrome

which line of the Javadoc comment is **problematic**:

```
/**  
 * Check if a word is a palindrome. (1)  
 * A palindrome is a sequence of characters (2)  
 * that reads the same forwards and backwards. (3)  
 * @param String word to check, must contain only alphanumeric characters (4)  
 * @return true if and only if word is a palindrome (5)  
 */
```

# What a specification may talk about

- A specification of a method can talk about the parameters and return value of the method, but it should **never** talk about local variables of the method or private fields of the method's class
  - you should consider the implementation *invisible* to the reader of the spec
- In Java, the source code of the method is often unavailable to the reader of your spec, because the Javadoc tool extracts the spec comments from your code and renders them as HTML



# Testing and Specifications (1)

---

- In lecture about Testing, we talk about black box tests that are chosen with only the specification in mind, and glass/white box tests that are chosen with knowledge of the actual implementation
  - but it's important to note that even glass box tests **must follow** the specification
  - your implementation may provide stronger guarantees than the specification calls for, or it may have specific behavior where the specification is undefined
  - but your test cases should not count on that behavior
  - test cases must obey the contract, just like every other client
- For example, suppose you are testing this specification of `find`, which is slightly different from the one we've used so far:

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

## Testing and Specifications (2)

---

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

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

## Testing and Specifications (3)

---

- Similarly, even if you implemented `find` so that it (sensibly) throws an exception when `val` isn't found, instead of returning some arbitrary misleading index,
  - your test case can't assume that behavior, because it can't call `find()` in a way that violates the precondition
- So what does glass box testing mean, if it can't go beyond the spec?
  - It means you are trying to find new test cases that *exercise different parts of the implementation*, but still checking those test cases in an *implementation-independent* way

# Testing Units (1)

---

- Recall the web search example from last week lecture with these methods:

```
/** @return the contents of the web page downloaded from url */
public static String getWebPage(URL url) { ... }

/** @return the words in string s, in the order they appear,
 *          where a word is a contiguous sequence of
 *          non-whitespace and non-punctuation characters */
public static List<String> extractWords(String s) { ... }

/** @return an index mapping a word to the set of URLs
 *          containing that word, for all webpages in the input set */
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {
    ...
    calls getWebPage and extractWords
    ...
}
```

## Testing Units (2)

---

- We talked then about ***unit testing***, the idea that we should write tests of each module of our program in isolation
  - A good unit test is focused on just a single specification
  - Our tests will nearly always rely on the specs of Java library methods, but a unit test for one method we've written *shouldn't fail* if a different method fails to satisfy its spec
  - As we saw in the example, a test for extractWords shouldn't fail if getWebPage doesn't satisfy its postcondition

## Testing Units (3)

---

- Good **integration tests**, tests that use a combination of modules, will make sure that our different methods have compatible specifications: callers and implementers of different methods are passing and returning values as the other expects
  - Integration tests cannot replace systematically-designed unit tests
  - From the example, if we only ever test extractWords by calling makeIndex, we will only test it on a potentially small part of its input space: inputs that are possible outputs of getWebPage
  - In doing so, we've left a place for bugs to hide, ready to jump out when we use extractWords for a different purpose elsewhere in our program, or when getWebPage starts returning web pages written in a new format, etc

# Specifications for Mutating Methods (1)

- We previously discussed mutable vs. immutable objects, but our specifications of `find` didn't give us the opportunity to illustrate how to describe side-effects — changes to mutable data — in the postcondition
- Here's a specification that describes a method that mutates an object:

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

- First, look at the postcondition
  - It gives two constraints: the first telling us how `list1` is modified, and
  - the second telling us how the return value is determined

# Specifications for Mutating Methods (2)

- The specification from previous slides:

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

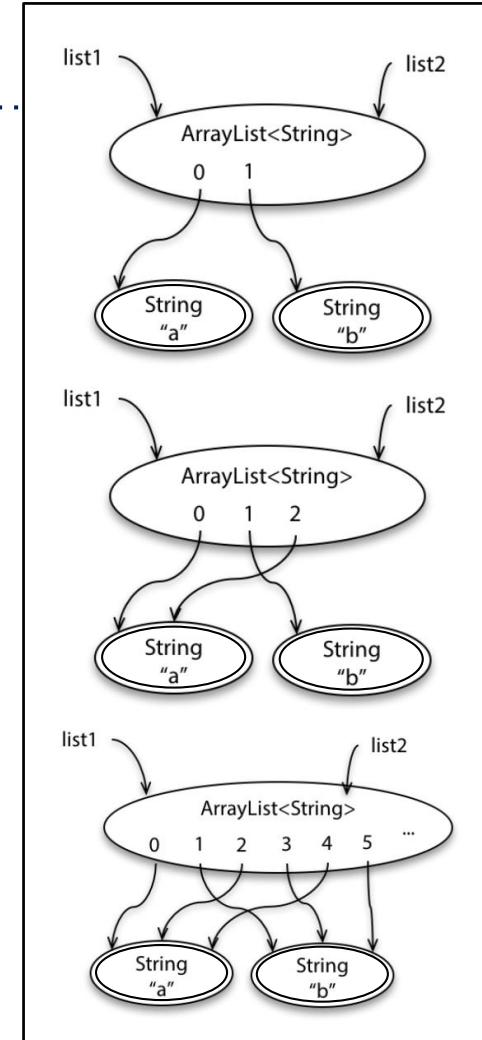
**requires:** list1 != list2

**effects:** modifies list1 by adding the elements of list2 to the end of it,  
and returns true if list1 changed as a result of call

- Second, look at the precondition — it tells us that the behavior of the method if you attempt to add the elements of a list to itself is undefined
  - you can easily imagine why the implementer of the method would want to impose this constraint:
    - it's not likely to rule out any useful applications of the method,
    - and it makes it easier to implement

# Specifications for Mutating Methods (3)

- The specification allows a simple algorithm in which you take an element from list2 and add it to list1, then go on to the next element of list2 until you get to the end of list2
- If list1 and list2 are the same list, then both lists will keep growing, and you will never get to the end
  - the *sequence of snapshot diagrams at right* illustrate this behavior
  - the simple algorithm above will not terminate — or practically speaking it will throw a memory error when the list object has grown so large that it consumes all available memory
  - either outcome, infinite loop or crash, is *permitted* by the specification because of its precondition



# Specifications for Mutating Methods (3)

---

- Remember also our implicit precondition that `list1` and `list2` must be valid objects, rather than `null`
- We'll usually *omit* saying this because it's virtually always required of object references
- Here is another example of a *mutating* method:

```
static void sort(List<String> list)  
  
    requires: nothing  
  
    effects: puts list in sorted order, i.e. list[i] <= list[j] for all  $0 \leq i < j < \text{list.size()}$ 
```

- And an example of a method that *does not mutate* its argument:

```
static List<String> toLowerCase(List<String> list)  
  
    requires: nothing  
  
    effects: returns a new list t where t[i] = list[i].toLowerCase()
```

## Specifications for Mutating Methods (4)

---

- Just as we've said that `null` is implicitly disallowed unless stated otherwise, we will also use the convention that ***mutation is disallowed*** unless stated otherwise
  - The spec of `toLowerCase` could explicitly state as an effect that “list is not modified”, but in the absence of a postcondition describing mutation, we demand no mutation of the inputs

## Part 2: From MyList to Singly Linked List (SLLList)

---

- In the next part of lecture, we will improve our homemade list
- We want users to use our data structure more easily, hiding details from them

```
MyList2 list = new MyList2( value: 10, next: null);
list = new MyList2( value: 2, list);
list = new MyList2( value: 5, list);
```

- We want users to create and add data without needing to specify reference/pointer

```
SLLList1 list = new SLLList1( item: 10);
list.addFirst( item: 2);
list.addFirst( item: 5);
```

# Inner / Nested Class

- For our new first implementation of list SLLList1, we use an inner class called Node, to store the **item** and pointer to the **next** node in the list (or null, if it's the last node)

```
private static class Node {  
    public int item;  
    public Node next;  
  
    public Node(int i, Node n) {  
        item = i;  
        next = n;  
    }  
}
```

class Node is defined **inside** class SLLList

private, since Node is only used by SLLList, and not other classes

static, since Node never uses instance variables/methods of the outer class

for private inner classes, these access modifiers are irrelevant

# Instance Variable and Constructor

---

- We store the pointer to the first node and always start with a list of one item

```
// The first node at the front of the list
private Node first;
```

```
/** Creates an SLList with one item. */
public SLList1(int item) {
    first = new Node(item, n: null);
}
```

# AddFirst and GetFirst

- addFirst is used to store a new item at the beginning of the list
  - while getFirst returns the item stored in the first node of the list

```
/**  
 * Adds a new node with an input item to the front of the List.  
 * @param item is an integer.  
 */  
  
public void addFirst(int item) {  
    first = new Node(item, first);  
}  
  
/**  
 * @return the first item in the front of the SLList.  
 */  
  
public int getFirst() {  
    return first.item;  
}
```

this will be enough to give what we want at the beginning

```
SLList1 list = new SLList1( item: 10);  
list.addFirst( item: 2);  
list.addFirst( item: 5);
```

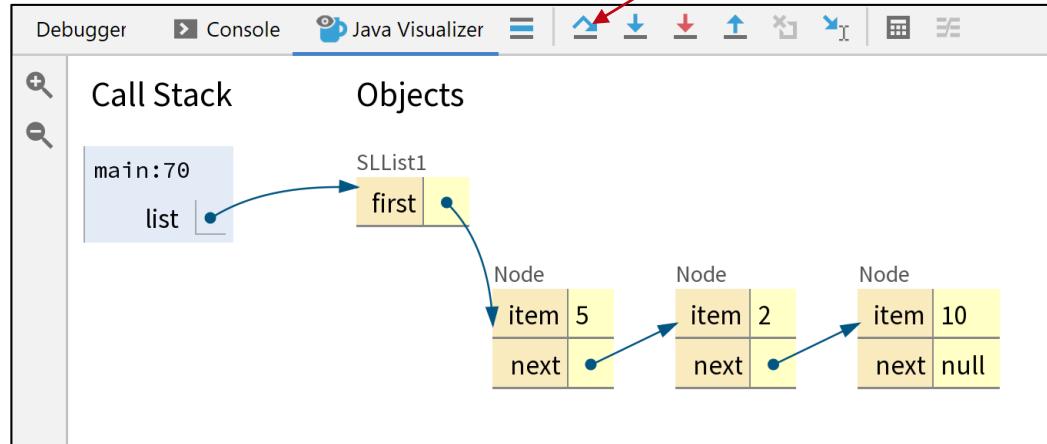
but let's add more functionalities to our list

# Java Visualizer

- Recall in week 4, you can use Java Visualizer to help you understand what's going on

```
66  public static void main(String[] args) { args: {}  
67  SLList1 list = new SLList1( item: 10); List: SLList1@790  
68      list.addFirst( item: 2);  
69      list.addFirst( item: 5);
```

use Step Over to skip going into a method



# AddLast

---

- To add item at the end of a list, we have to go to the last node first

```
/*
 * Adds item to the end of the list.
 * @param item is an integer.
 */
public void addLast(int item) {
    Node p = first;
    // Advance p to the end of the list
    while (p.next != null) {
        p = p.next;
    }
    p.next = new Node(item, n: null);
}
```

# Size

---

- To return the size recursively, we have to use a recursive helper method

```
/**  
 * @return the size of the SLList recursively.  
 */  
  
public int size() { return size(first); }  
  
// Recursive size helper method  
private int size(Node p) {  
    // base case  
    if (p.next == null) {  
        return 1;  
    }  
    // recursive step  
    return 1 + size(p.next);  
}
```

stop!

this implementation of list takes  
*linear time* to return the size

can you think of a way to return  
the size in **constant time**?

# SLList2 with Caching Size Instance Variable

- We store the size information as an instance variable, set and update it accordingly,

```
// The first node at the front of the list  
private Node first;  
private int size;  
  
/** Creates an SLList with one item. */  
public SLList2(int item) {  
    first = new Node(item, null);  
    size = 1;  
}
```

```
public void addFirst(int item) {  
    first = new Node(item, first);  
    size += 1;  
}
```

so can return size in constant time:

```
/**  
 * @return the size of the SLList.  
 */  
public int size() {  
    return size;  
}
```

this technique is called **caching**  
we can also use caching to store max, min, mode

# Empty SLList

- Suppose we want to start with no item. How do we create an empty list?

```
/** Creates an empty SLList with no item. */
public SLList2() {
    first = null;
    size = 0;
}
```

- It works well with:

```
SLList2 list2 = new SLList2();
list2.addFirst( item: 200 );
list2.addFirst( item: 100 );
list2.addLast( item: 300 );
System.out.println(list2.size());
```

**stop!**

it turns out there is **a bug**.  
can you think how to add to the  
list starting with empty list that  
can cause an error?

# Bug in Starting with an Empty SLLlist

- When you do addLast immediately after creating an empty SLLlist, there is a runtime error

```
SLLlist2 list3 = new SLLlist2();
list3.addLast( item: 50);
list3.addFirst( item: 72);
```

```
Exception in thread "main" java.lang.NullPointerException
  at SLLlist2.addLast(SLLlist2.java:52)
  at SLLlist2.main(SLLlist2.java:81)
```

- In addLast:

```
public void addLast(int item) {
    Node p = first;
    // Advance p to the end of the list
    while (p.next != null) {
        p = p.next;
```

we cannot have **null.next**!

stop!

how do you fix this bug in addLast?

# AddLast v2

---

- We modify addLast so that it simply adds a node if it was empty

```
/**  
 * Adds item to the end of the list.  
 * @param item is an integer.  
 */  
  
public void addLast(int item) {  
    if (first == null) {  
        first = new Node(item, n: null);  
        return;  
    }  
    Node p = first;  
    // Advance p to the end of the list  
    while (p.next != null) {  
        p = p.next;  
    }  
    p.next = new Node(item, n: null);  
    size += 1;  
}
```



add the following code

there is another solution

# SLList3 with Sentinel Node: Empty Constructor

- Motivation: we want to avoid special cases, where an empty list is null
- Idea: let the empty list be a list with one sentinel node, a node without real data

```
// The first node at the front of the list is a sentinel node  
// The first item, if it exists, is in the sentinel.next node  
private Node sentinel;  
private int size;
```

```
/** Creates an empty SLList with no item. */  
public SLList3() {  
    sentinel = new Node( i: 0, n: null);  
    size = 0;  
}
```

instead of null,  
now an empty SLList contains  
a sentinel node

the item here can be anything  
it does not matter

so now, the list is never null

# SLLList with Sentinel: Constructor and AddFirst

- The constructor for a one-item list:

```
/** Creates an SLLList with one item. */
public SLLList3(int item) {
    sentinel = new Node(i: 0, n: null);
    sentinel.next = new Node(item, n: null);
    size = 1;
}
```

create a sentinel node,  
and the real node with input data

- Adding in the front of the list:

```
public void addFirst(int item) {
    sentinel.next = new Node(item, sentinel.next);
    size += 1;
}
```

create a new node in between the  
sentinel and the first real node

# SLList with Sentinel: GetFirst and AddLast

- Getting the first item of the list:

```
public int getFirst() {  
    return sentinel.next.item;  
}
```

the first item is in the node  
after the sentinel

- Adding in the end of the list:

```
public void addLast(int item) {  
    Node p = sentinel;  
    // Advance p to the end of the list  
    while (p.next != null) {  
        p = p.next;  
    }  
    p.next = new Node(item, n: null);  
    size += 1;  
}
```

no special cases,  
just find the end of list to add node,  
which could be after the sentinel

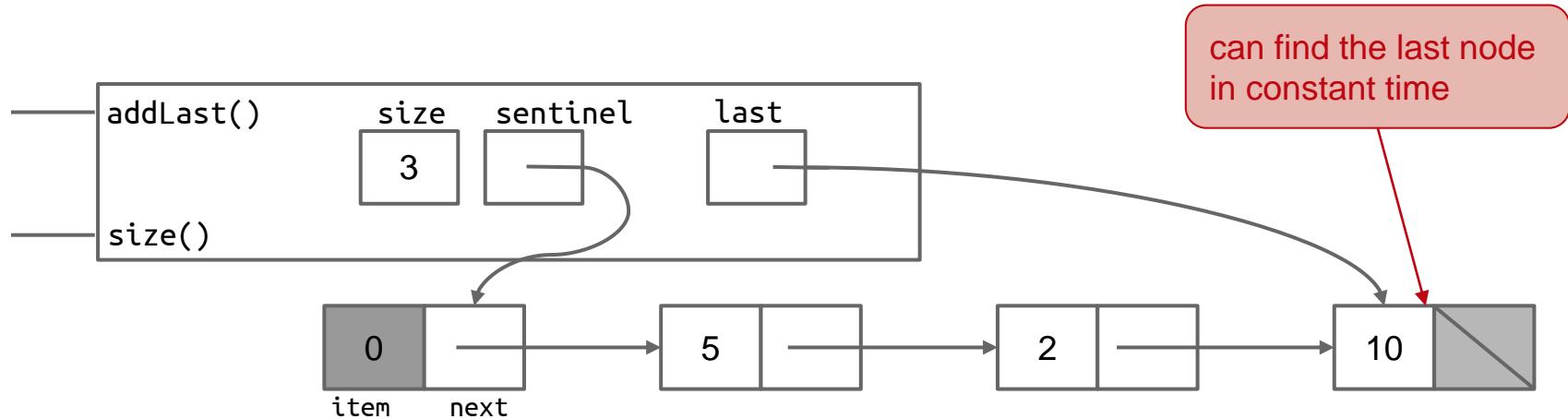
# Invariants

---

- An invariant is a condition that is guaranteed to be ***true*** during code execution
- For example, `SLLList` with Sentinel Node has the following invariants:
  - sentinel instance variable *always* points to a sentinel node
  - the first node, if it exists, is *always* at `sentinel.next`
  - size instance variable is *always* the total number of items added
- Invariants make it easier to reason about code:
  - can assume they are true to simplify code,  
e.g., `addLast` does not need to worry about the null case
  - must ensure that the methods ***preserve invariants***

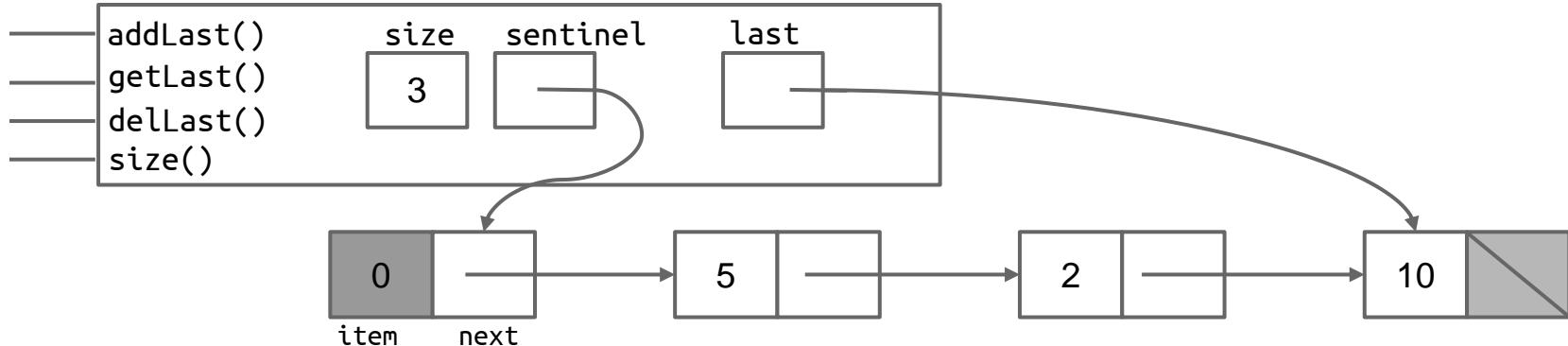
# Next Improvement: AddLast

- AddLast is slow, taking linear time, proportional to size of list
  - What should we do to make it fast?
- We may think, using the caching technique like last time, store a pointer to the last node



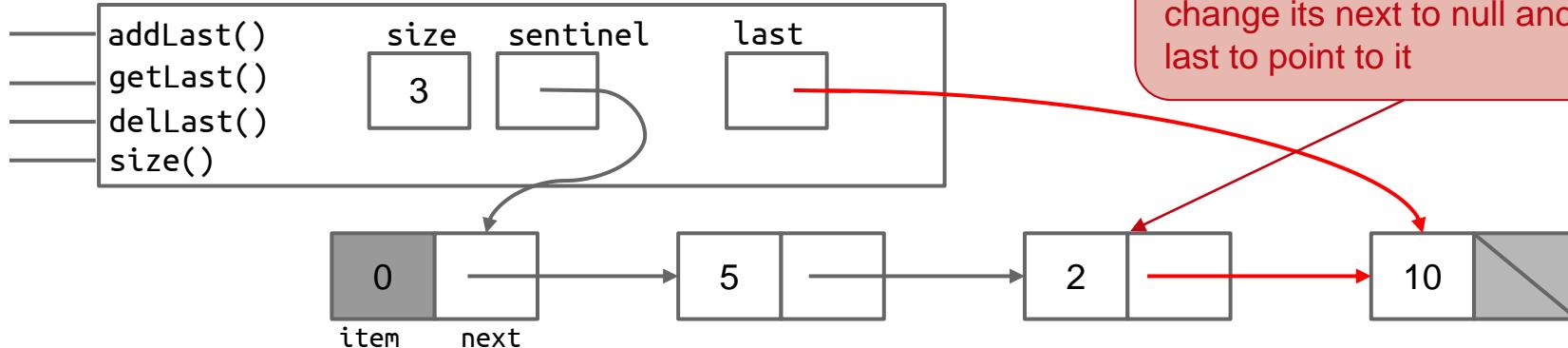
## Next Improvement: AddLast, GetLast, DelLast

- AddLast is slow, taking linear time, proportional to size of list
  - What should we do to make it fast?
- We may think, using the caching technique like last time, store a pointer to the last node
  - Suppose now we want to **addLast**, **getLast**, and **delLast** from the back, would storing pointer to last node make them fast?



# Next Improvement: AddLast, GetLast, DelLast

- AddLast is slow, taking linear time, proportional to size of list
  - What should we do to make it fast?
- We may think, using the caching technique like last time, store a pointer to the last node
  - Suppose now we want to **addLast**, **getLast**, and **delLast** from the back, would storing pointer to last node make them fast?

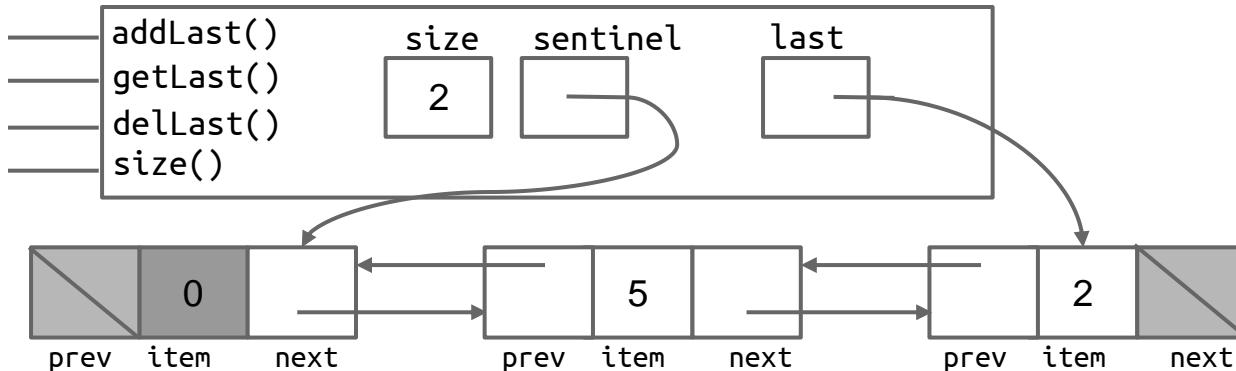


In Java, garbage collector will remove object without reference to it

# Doubly-Linked List

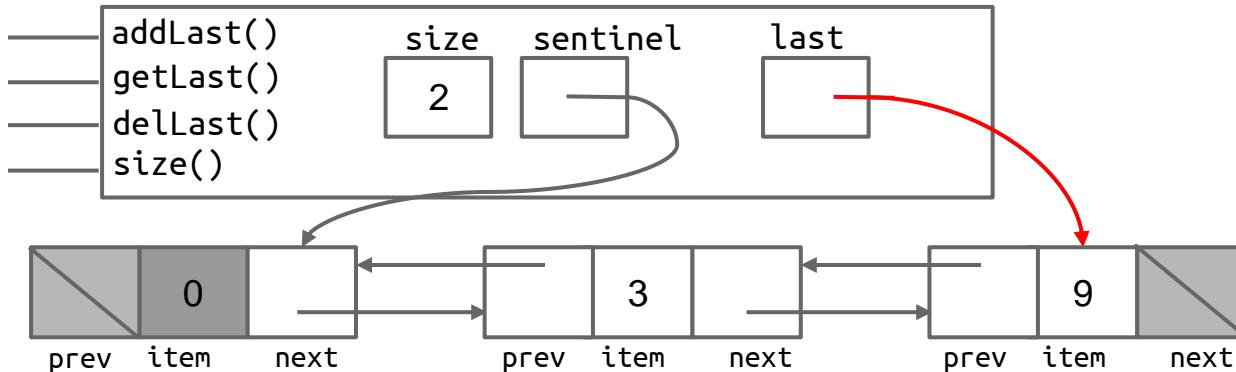
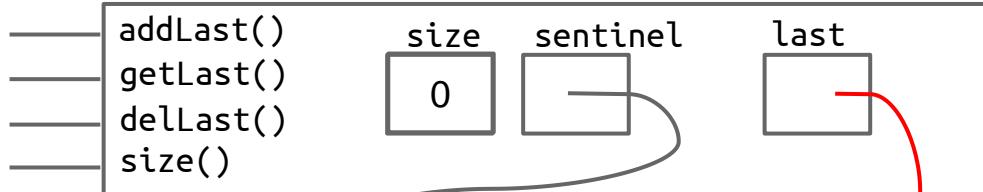
---

- In addition to a pointer to the last node, to make `removeLast` fast, need to add backward link from every node to its previous node, let's call it `prev`
- This results in a Doubly-Linked List (DLLList) :



# Doubly-Linked List with Single Sentinel

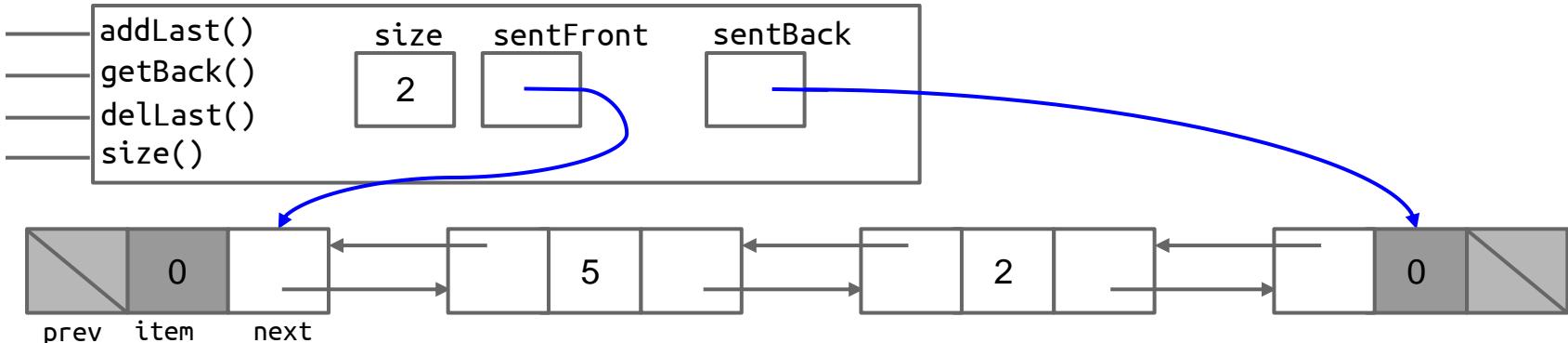
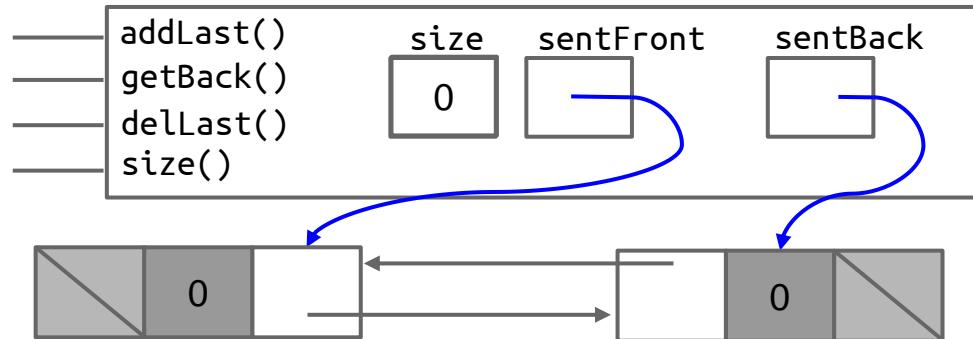
- If you try to code using DLLList with just one sentinel, there could be a special case where sometimes last point to sentinel or real node



# Doubly-Linked List with Double Sentinel

- One possible solution is to have two sentinels:

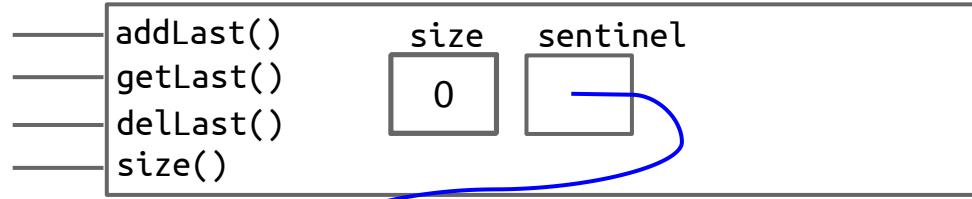
you can use this as an exercise



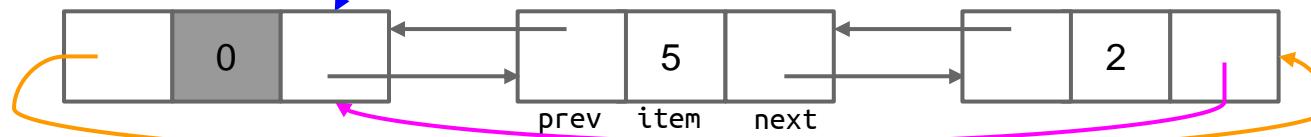
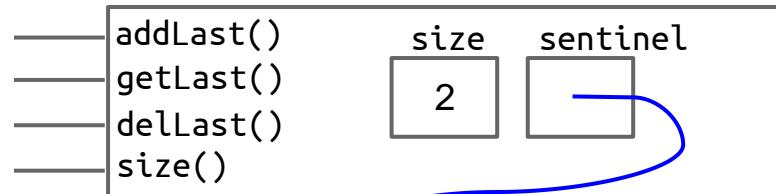
# Doubly-Linked List with Circular Sentinel

- Another possible solution is to use circular sentinel:

you will use this for Lab 5 !



it has **one sentinel** that's both the front and the back



# Another Improvement: Generics

- Notice that our list only supports storing an integer
  - it will generate error if we try, for example, to create a list of String
- How do we improve our SLLList so that it can store any type of item? Use generics!

```
public class SLLList4<T> {  
  
    private class Node {  
        public T item;  
        public Node next;  
  
        public Node(T i, Node n) {  
            item = i;  
            next = n;  
        }  
    }  
}
```

usually people like to use T or E here  
but it can be anything in here

now the class is parameterized by  
type T, inner class cannot be static

change every type of item in class,  
from int to T

# Using SLLList4 with Generics

```
public SLLList4() {  
    sentinel = new Node( i: null, n: null);  
    size = 0;  
}  
  
public void addFirst(T item) {  
    sentinel.next = new Node(item, sentinel.next);  
    size += 1;  
}
```

store null as an item in Sentinel node

set all type of item in class to be T

- To use the list, just use it like you use Java's ArrayList:

```
public static void main(String[] args) {  
    SLLList4<String> listStr = new SLLList4<>();  
    listStr.addFirst( item: "cd");  
    listStr.addFirst( item: "ab");  
    System.out.println(listStr.getFirst());  
}
```

# Thank you for your attention !

---

- In this lecture, you have learned about:
  - Specifications
    - use preconditions and postconditions in method specifications
    - be able to write tests against a specification
  - Linked List 2
    - create and improve your own Linked List
    - equip it with methods that work in constant-time
    - make it able to store items of any data types
- Please continue to Lecture Quiz 5 and Lab 5:
  - to do Lab Exercise 5.1 - 5.4, and
  - to do Exercise 5.1 - 5.4



# Advanced Object-Oriented Programming

CPT204 – Lecture 6  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming

### Lecture 6

# Exception, Linked List 3

# Welcome !

---

- Welcome to Lecture 6 !
- In this lecture we are going to
  - learn about null values
  - learn about exception
    - checked vs unchecked
    - handling exception
  - continue improving our homemade linked list, while learning about
    - copy constructor
    - shallow copy vs deep copy

# Null References (1)

---

- 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
double depth = null; // illegal
```

## Null References (2)

---

- On the other hand, we can assign null to any non-primitive variable and the compiler happily accepts this code at compile time:

```
String name = null;  
int[] points = null;
```

- But you'll get **errors at runtime** because you can't call any methods or use any fields with one of these references:

```
name.length()    // throws NullPointerException  
points.length   // throws NullPointerException
```

## Null References (3)

---

- Note that null is **not** the same as an empty string "" or an empty array
  - On an empty string or empty array, you **can** call methods and access fields
  - The length of an empty array or an empty string is 0
  - The length of a string variable that points to null isn't anything: calling `length()` throws a `NullPointerException`
- Also note that arrays of non-primitives and collections like `List` might be non-null but contain null as a value:

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

- These nulls are likely to cause errors as soon as someone tries to use the contents of the collection!

# Null Values (1)

---

- Null values are troublesome and unsafe, and in fact in most good Java programming, null values are *implicitly disallowed* in *parameters* and *return values*
  - so every method implicitly has a precondition on its object and array parameters that they be non-null, and
  - every method that returns an object or an array implicitly has a postcondition that its return value is non-null
  - if a method allows null values for a parameter, it should *explicitly* state it, or if it might return a null value as a result, it should *explicitly* state it

# Null Values (2)

---

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

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

where it can be checked automatically at compile time or runtime

- Google has their own discussion of null in Guava, the company's core Java libraries.  
The project explains:

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**.

# Exceptions

---

- Next, we discuss how to handle *exceptional cases* in a way that is safe from bugs and easy to understand
- A method's signature — its name, parameter types, return type — is a core part of its specification, and the signature may also include *exceptions* that the method may *trigger*

# Exceptions for Signaling Bugs

---

- We have already seen some exceptions in our Java programming so far, such as
  - `IndexOutOfBoundsException`, thrown when a list index `list.get(i)` is outside the valid range for the list `list`
  - `NullPointerException`, thrown when trying to call a method on a null object reference
- These exceptions generally ***indicate bugs in your code***, and the information displayed by Java when the exception is thrown can help you find and fix the bug
- `IndexOutOfBoundsException` and `NullPointerException` are probably the most common exceptions of this sort
- Other examples include:
  - `ArithmaticException`, thrown for arithmetic errors like integer division by zero
  - `NumberFormatException`, thrown by methods like `Integer.parseInt` if you pass in a string that cannot be parsed into an integer

# Exceptions for Special Results (1)

---

- Exceptions are not just for signaling bugs
- They can be used to improve the structure of code that involves procedures with **special results**
- An unfortunately common way to handle special results is to return special values
- Lookup operations in the Java library are often designed like this:  
you get an index of -1 when expecting a positive integer, or a null reference when expecting an object
- This approach is OK if used sparingly, but it has two problems :
  - First, it's tedious to check the return value
  - Second, it's easy to forget to do it

## Exceptions for Special Results (2)

---

- Also, it's not always easy to find a *special value*.

Suppose we have a `BirthdayBook` class with a `lookup` method.

Here's one possible method signature:

```
class BirthdayBook {  
    LocalDate lookup(String name) { ... }  
}
```

(`LocalDate` is part of the Java API)

- What should the method do if the birthday book *doesn't* have an entry for the person whose name is given?
  - we could return some *special date* that is not going to be used as a real date
  - bad programmers have been doing this for decades; they would return 9/9/99, for example, since it was obvious that no program written in 1960 would still be running at the end of the century

btw, they are wrong,  
caused Y2K bug

## Exceptions for Special Results (3)

---

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

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

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

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

- Now there's no need for any special value, nor the checking associated with it

# Exceptions for Special Results (3)

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

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

advertise the exception  
in method's signature

the condition that triggers throwing exc

an **object** of type NotFoundException  
is thrown by lookup if date not found,  
method lookup execution then ends

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

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

method that throws exception  
is called inside try block here

codes here executed if lookup does  
not throw a NotFoundException object

codes here executed if lookup throws  
a NotFoundException object

- Now there's no need for any special value, nor the checking associated with it

# In-Class Quiz 1

---

- We use BirthdayBook with the lookup method that throws NotFoundException
- Assume we have initialized the birthdays variable to point to a BirthdayBook, and that "Makima" is **not** in that birthday book
- What will happen with the following code:

```
try {  
    LocalDate birthdate = birthdays.lookup("Makima");  
}  
System.out.println("done");
```

- static error caused by incorrect syntax
- static error caused by undeclared variable
- dynamic error caused by NotFoundException
- no errors and it prints "done"

## In-Class Quiz 2

---

- We use BirthdayBook with the lookup method that throws NotFoundException
- Assume we have initialized the birthdays variable to point to a BirthdayBook, and that "Makima" is **not** in that birthday book
- What will happen with the following code:

```
try {  
    LocalDate birthdate = birthdays.lookup("Makima");  
} catch (NotFoundException nfe) {  
    birthdate = LocalDate.now();  
}  
System.out.println("done");
```

- static error caused by incorrect syntax
- static error caused by undeclared variable
- dynamic error caused by NotFoundException
- no errors and it prints "done"

## In-Class Quiz 3

---

- We use BirthdayBook with the lookup method that throws NotFoundException
- Assume we have initialized the birthdays variable to point to a BirthdayBook, and that "Makima" is **not** in that birthday book
- What will happen with the following code:

```
try {  
    LocalDate birthdate = birthdays.lookup("Makima");  
} catch (Exception NotFoundException) {  
}  
System.out.println("done");
```

- static error caused by incorrect syntax
- static error caused by undeclared variable
- dynamic error caused by NotFoundException
- no errors and it prints "done"

# Exception Message (1)

- All exceptions may have a message associated with them in the constructor
  - to pass useful information about the cause of throwing the exception

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

- in the catch block, it is accessed by getMessage()

```
try {  
    LocalDate birthdate = birthdays.lookup("Makima");  
} catch (NotFoundException nfe) {  
    System.err.println("birthdate of " + nfe.getMessage());  
}
```

## Exception Message (2)

---

- All exceptions may have a message associated with them in the constructor
  - if **not** provided in the constructor, the reference to the message string is null
  - this can result in confusing stack traces that start, for example:

```
birthday.NotFoundException: null  
at birthday.BirthdayBook.lookup(BirthdayBook.java:42)
```

- the null is misleading : in this case, it tells you the NotFoundException had no message string, **not that** a null value was responsible for the exception

# Checked and Unchecked Exceptions (1)

---

- We've seen two different purposes for exceptions:
  - special results, and
  - bug detection
- As a general rule, you'll want to use **checked exceptions** to signal *special results* and **unchecked exceptions** to *signal bugs*

# Checked Exception Example

---

- Suppose NotFoundException is a Checked Exception:
  - callee must declare/advertise it:

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

- caller must handle it (with try-catch) : [ or declare/advertise it (to be handled by its caller) ]

```
public void congratulateMakima() {  
    ...  
    try {  
        LocalDate birthdate = birthdays.lookup("Makima");  
    } catch (NotFoundException nfe) {  
        System.err.println("birthdate of " + nfe.getMessage());  
    }  
    ...
```

# Checked and Unchecked Exceptions (2)

---

- *Checked exceptions* are called that because they are *checked by the compiler*:
  - if a method might throw a checked exception, the possibility must be declared in its signature
    - `NotFoundException` would be a checked exception, and that's why the signature ends with `throws NotFoundException`
  - if a method calls another method that may throw a checked exception, it must either handle it (with try-catch), *or* declare the exception itself, since if it isn't caught locally it will be propagated up to callers
- So if you call `BirthdayBook`'s `lookup` method and *forget to handle* the `NotFoundException`, the compiler will **reject** your code
  - this is very useful, because it ensures that exceptions that are expected to occur will be handled

# Checked and Unchecked Exceptions (3)

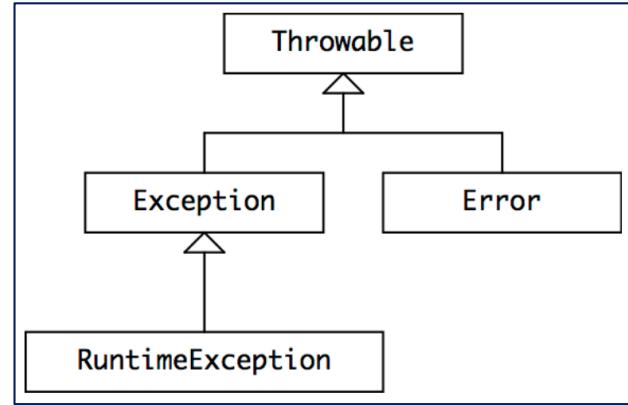
---

- *Unchecked exceptions*, in contrast, are used to signal bugs
  - these exceptions are **not** expected to be handled by the code, except perhaps at the top level
  - we wouldn't want every method up the call chain to have to declare that it (might) throw all the kinds of bug-related exceptions that can happen at lower call levels: index out of bounds, null pointers, illegal arguments, assertion failures, etc
- As a result, for an unchecked exception the compiler will **not** check for try-catch or a throws declaration
  - Java *still allows* you to write a throws clause for an unchecked exception as part of a method signature, but this has **no effect**, we don't recommend doing it

# Java Exception Hierarchy

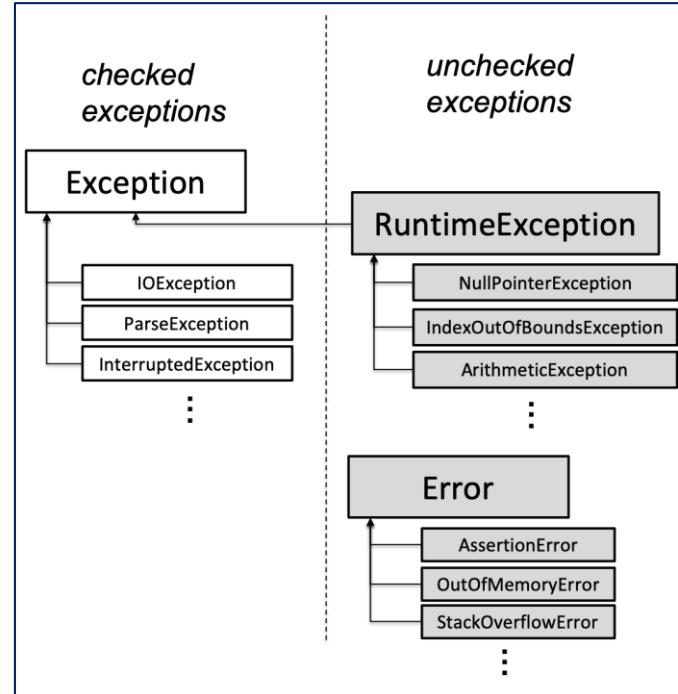
---

- To understand how Java decides whether an exception is checked or unchecked, let's look at the class hierarchy for Java exceptions
- **Throwable** is the class of objects that can be thrown or caught
  - **Throwable**'s implementation records a stack trace at the point where the exception was thrown, along with an optional string describing the exception
  - any object used in a `throw` or `catch` statement, or declared in the `throws` clause of a method, must be a subclass of `Throwable`



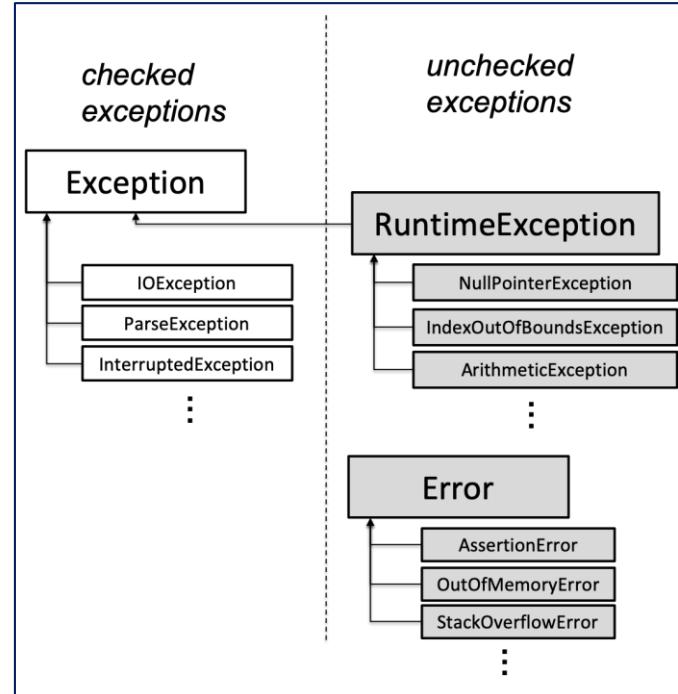
# Checked vs Unchecked Exception in Java (1)

- **Exception** is the normal base class of **checked exceptions**
- The compiler applies **static checking** to methods using these exceptions
  - A checked exception **must** either be caught or declared when it's possible for it to be thrown



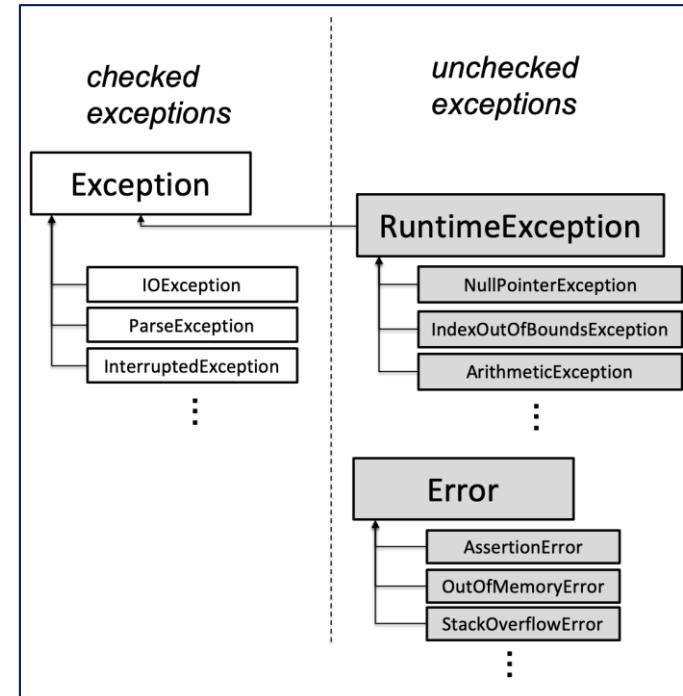
# Checked vs Unchecked Exception in Java (2)

- However, **RuntimeException** and its subclasses are **unchecked exceptions**
- **RuntimeException** and its subclasses ***don't have to*** be declared in the throws clause of a method that throws them,
  - and ***don't have to*** be caught or declared by a caller of such a method



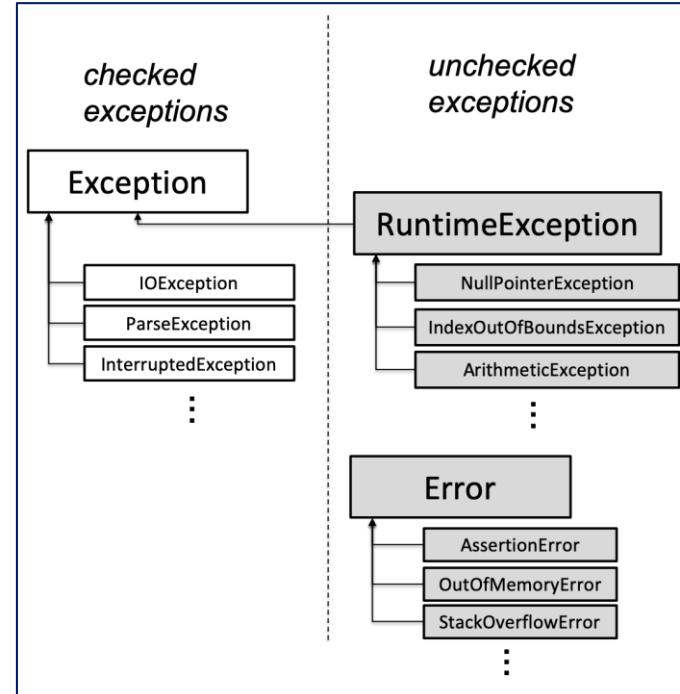
# Checked vs Unchecked Exception in Java (3)

- In addition, `Error` and its subclasses are also **unchecked exceptions**
- This part of the hierarchy is reserved for errors produced by the Java runtime system, such as `StackOverflowError` and `OutOfMemoryError`
- For some reason `AssertionError` also extends `Error`, even though it indicates a bug in user code, not in the runtime
- Errors should be considered unrecoverable, and should **not** be caught by your code



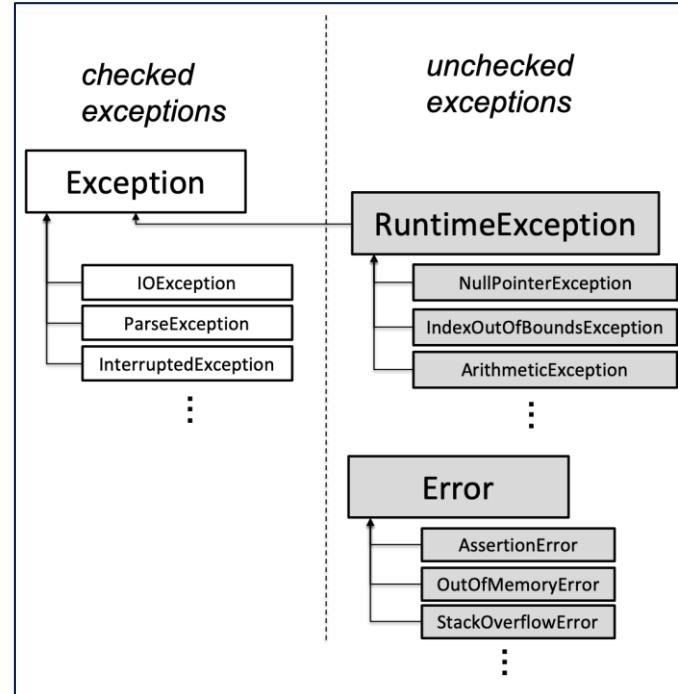
# Creating Exception

- When you define your own exceptions, ***you should either subclass Exception to make it an checked exception, or subclass RuntimeException to make it unchecked exception***
- Don't*** subclass Error or Throwable, because these are reserved by Java itself



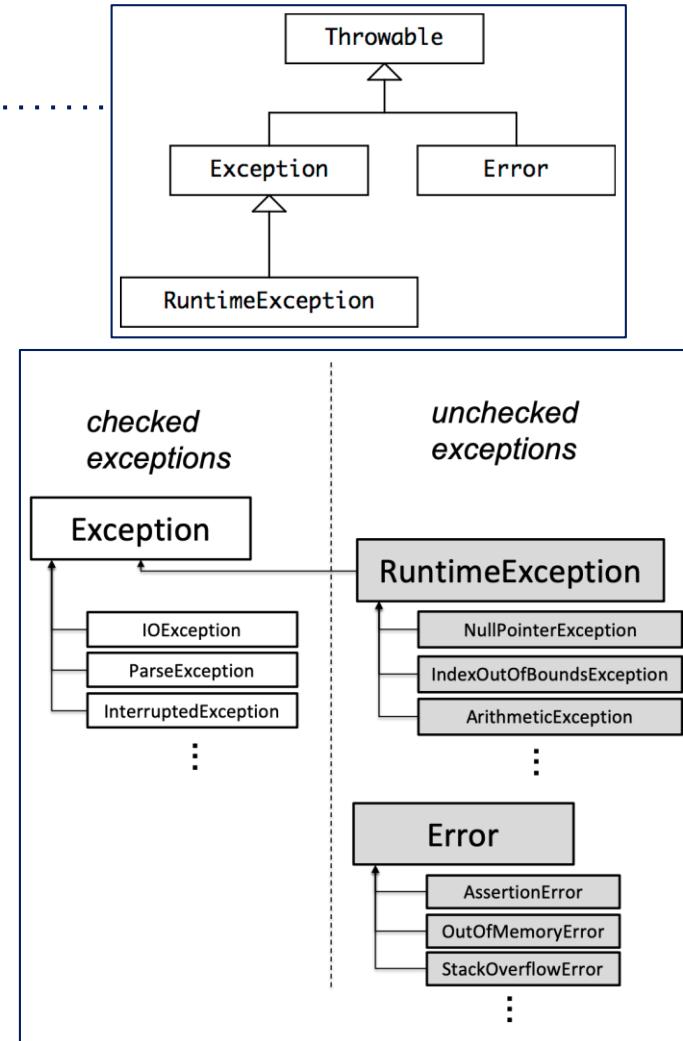
# Catching Exception

- When you catch an exception with a try-catch statement, you should catch the most specific exception class possible
- If you are expecting a `FileNotFoundException`, that's what your catch statement should use
- Catching a broad class of exceptions, like `Exception` or `RuntimeException` or `Error`, is not safe from bugs or ready for change
  - because it catches **every** possible subclass of these exceptions, which may interfere with static checking and hide bugs



# Java Exception Hierarchy

- One of the *confusing aspects* of the Java exception hierarchy is that `RuntimeException` is itself a subclass of `Exception`
- So the whole `Exception` family includes ***both checked exceptions (its direct descendants) and unchecked exceptions (the `RuntimeException` branch of the family)***
- But `Error` is ***not*** a subclass of `Exception`, so all the *unchecked Error-like exceptions* are outside the `Exception` family



# Exception Design Considerations (1)

---

- The rule we have given — use checked exceptions for special results (i.e., anticipated situations), and unchecked exceptions to signal bugs (unexpected failures) — makes sense, but it isn't the end of the story
- Exceptions in Java aren't as lightweight as they might be
  - They are expensive!
- Aside from the performance penalty, exceptions in Java incur another (more serious) cost: they're a pain to use, in both method design and method use
  - If you design a method to have its own (new) exception, you have to create a new class for the exception
  - If you call a method that can throw a checked exception, you have to wrap it in a try-catch statement (*even* if you know the exception will never be thrown)

## Exception Design Considerations (2)

---

- Suppose, for example, you're designing a queue
  - Should *popping* the queue throw a checked exception when the queue is empty?
  - Suppose you want to support a style of programming in the client in which the queue is popped in a loop, until the exception is thrown
  - So you choose *a checked exception*
  - Now some client wants to use the method in a context in which, immediately prior to popping, the client *tests* whether the queue is empty and *only pops if it isn't*
  - That client will ***still need*** to wrap the call in a try-catch statement!

## Exception Design Considerations (3)

---

- This suggests a more refined rule:
  - You should use an ***unchecked exception*** only to signal an *unexpected failure* (i.e. a bug),  
or if you *expect* that clients will *usually write code that ensures the exception will not happen*, because there is a convenient and inexpensive way to avoid the exception
  - Otherwise you should use a ***checked exception***

## Exception Design Considerations (4)

---

- Here are some examples of applying this rule :
  - Queue.pop() throws an **unchecked** EmptyQueueException when the queue is empty
    - because it's reasonable to expect the caller to *avoid* this with a call like Queue.size() or Queue.isEmpty()
  - Url.getPage() throws a **checked** IOException when it can't retrieve the web page
    - because it's not easy for the caller to *prevent* this

# Exception Design Considerations (5)

---

- int integerSquareRoot(int x) throws a **checked** NotPerfectSquareException when x has no integral square root,
  - because testing whether x is a perfect square is *just as hard* as finding the actual square root, so it's *not reasonable* to expect the caller to *prevent* it
- The cost of using exceptions in Java is one reason that many Java API's use the null reference as a special value
  - it's not a terrible thing to do, so long as it's done judiciously, and carefully specified

# Declare Exceptions in Specification (1)

---

- Since an exception is a possible output from a method, it may have to be described in the postcondition for the method
- The Java way of documenting an exception as a postcondition is a `@throws` clause in the Javadoc comment
  - Recall that Java may also require the exception to be included in the method signature, using a throws declaration
- Exceptions that **signal a special result** (checked or unchecked) are always documented with a Javadoc `@throws` clause, specifying the conditions under which that special result occurs

## Declare Exceptions in Specification (2)

---

- For a **checked exception**, Java also requires the exception to be mentioned in a throws declaration in the method signature
- For example, suppose NotPerfectSquareException were a checked exception; You would need to mention it in both @throws in the Javadoc and throws in the method signature:

```
/**  
 * Compute the integer square root.  
 * @param x value to take square root of  
 * @return square root of x  
 * @throws NotPerfectSquareException if x is not a perfect square  
 */  
int integerSquareRoot(int x) throws NotPerfectSquareException
```

## Declare Exceptions in Specification (3)

---

- For an **unchecked exception** that signals a special result, Java allows but doesn't require the throws clause; but it is better to omit the exception from the throws clause in this case, to avoid misleading a human programmer into thinking that the exception is checked
- For example, suppose you defined EmptyQueueException as an unchecked exception, then you should document it with @throws, but **not** include it in the method signature:

```
/**  
 * Pops a value from this queue.  
 * @return next value in the queue, and removes the value from the queue  
 * @throws EmptyQueueException if this queue is empty  
 */  
int pop()
```

## Declare Exceptions in Specification (4)

- Exceptions that **signal unexpected failures** – bugs in either the client or the implementation – are not part of the postcondition of a method, so they should **not** appear in either @throws or throws
- For example, NullPointerException *need never* be mentioned in a spec
  - an implicit precondition already disallows null values, which means that a valid implementation is free to throw it without any warning if a client ever passes a null value
- So this spec, for example, *never* mentions NullPointerException:

```
/**  
 * @param list list of strings to convert to lower case  
 * @return new list t, same length as list,  
 *         where t[i] is list[i] converted to lowercase for all valid indices i  
 */  
static List<String> toLowerCase(List<String> list)
```

# Abuse of Exceptions (1)

---

- Consider the following example from Effective Java, Joshua Bloch:

```
try {
    int i = 0;
    while (true)
        a[i++].f();
} catch (ArrayIndexOutOfBoundsException e) { }
```

- What does this code do?

It is not at all obvious from inspection, and that's reason enough not to use it

- The infinite loop terminates by throwing, catching, and ignoring an `ArrayIndexOutOfBoundsException` when it attempts to access the first array element outside the bounds of the array

## Abuse of Exceptions (2)

---

- It is supposed to be equivalent to:

```
for (int i = 0; i < a.length; i++) {  
    a[i].f();  
}
```

- Or (using appropriate type T) to:

```
for (T x : a) {  
    x.f();  
}
```

- The exception-based idiom, Bloch writes:

- ... is a misguided attempt to improve performance based on the **faulty reasoning** that, since the VM checks the bounds of array accesses, the normal loop termination test ( $i < a.length$ ) is redundant and should be avoided

## Abuse of Exceptions (3)

---

- However, because exceptions in Java are designed for use only under exceptional circumstances, few, if any, JVM implementations attempt to optimize their performance
  - On a typical machine, the exception-based idiom runs 70 times slower than the standard one when looping from 0 to 99
- Much worse than that, the exception-based idiom is not even guaranteed to work!
  - Suppose the computation of `f()` in the body of the loop contains a bug that results in an out-of-bounds access to some unrelated array
  - What happens?
  - If a *reasonable loop idiom* were used, the bug would generate an uncaught exception, resulting in immediate thread termination with a full stack trace
  - If the *misguided exception-based loop* were used, the bug-related exception would be caught and misinterpreted as a normal loop termination!

# Copy Constructor

---

- Copy constructor is a special constructor that takes as input another object of the same type and create a new object that is a copy of the input
  - the other object is usually called other
  - a copy here for a data structure usually means having the same elements
- Let us create a copy constructor for our running example, `SLList<T>`

# Shallow Copy

---

- Suppose you code the Copy Constructor for `SLList<T>` as below

```
// Shallow Copy  
// NOT what we want !  
public SLList(SLList<T> other) {  
    sentinel = other.sentinel;  
    size = other.size;  
}
```

you simply set the sentinel to point to other's sentinel and copy the size

- This is called **shallow copy or aliasing**
- Not what we want, because we want to create a new copy
  - while this is just having a new pointer to the old object, other viewed another way, this is just giving the old object a new name / alias

# Deep Copy

---

- What we want is to create an entirely new `SLList<T>`, with the exact same items as other
- This is called **deep copy**
  - The input and the copy output should be different objects
  - If you change other, the new `SLList<T>` you created should **not** change as well

# SLLList<T> Copy Constructor

- Here is the copy constructor for SLLList<T> that correctly uses deep copy

```
/** Creates a (deep) copy SLLList of other. */
public SLLList(SLLList<T> other) {
    sentinel = new Node( i: null, n: null);
    size = 0;

    Node p = sentinel;
    Node q = other.sentinel;
    while (q.next != null) {
        p.next = new Node(q.next.item, n: null);
        p = p.next;
        q = q.next;
        size += 1;
    }
}
```

start with an empty list

create a new node and  
copy from other one-by-one

# SLLList<T> Copy Constructor 2

- Another way is to use methods that we have previously defined!

```
/** Creates a (deep) copy SLLList of other. */
public SLLList(SLLList<T> other) {
    this(); ← call the empty constructor

    Node q = other.sentinel;
    while (q.next != null) {
        this.addLast(q.next.item); ← use addLast to add the item
        q = q.next;
    }
}
```

call the empty constructor

use addLast to add the item  
from other one-by-one

# NullPointerException

---

- If we try to copy a null, we will get a NullPointerException

```
// NullPointerException  
SLList<String> listStr1 = null;  
SLList<String> listStr2 = new SLList<>(listStr1);  
listStr2.printList();
```

NullPointerException  
thrown when it tries to  
access sentinel of null

# Unchecked IllegalStateException

- We can avoid that by using an unchecked exception `IllegalArgumentException`

```
/*
 * Creates a deep copy SLList of other.
 * @param other an SLList object
 * @throws IllegalArgumentException if other is null
 */
public SLList(SLList<T> other) {
    if (other == null) {
        throw new IllegalArgumentException("other is null");
    }

    sentinel = new Node( i: null, n: null);
    size = 0;

    Node p = sentinel;
    Node q = other.sentinel;
    while (q.next != null) {
        this.addLast(q.next.item);
        q = q.next;
    }
}
```

The code demonstrates the use of `IllegalArgumentException` as an unchecked exception. It includes Javadoc annotations for parameters and throws clauses. Red arrows from callout boxes point to specific parts of the code:

- A red arrow from the callout "no need to advertise" points to the class header `/* Creates a deep copy SLList of other.`
- A red arrow from the callout "check the throwing condition" points to the `if (other == null)` condition.
- A red arrow from the callout "pass a message" points to the string argument in the `IllegalArgumentException` constructor, `"other is null"`.

# Catching `IllegalArgumentException`

---

- The exception can be caught in `main`
  - note that without try-catch the code still compiles, since we use an unchecked exception

```
// Try-Catch IllegalArgumentException
SLList<String> listStr1 = null;
try {
    SLList<String> listStr2 = new SLList<>(listStr1);
} catch (IllegalArgumentException iae) {
    System.out.println("Copy failed, " + iae.getMessage());
}
```

print a message to help identify why the exception happened

# Test Exception is Thrown

---

- We can use try-catch in JUnit and test for the string output
- Alternatively, we can test as follows:

```
@Test(expected = IllegalArgumentException.class)
public void testIllegalArgumentExceptionNullCopy() {
    SLLList<String> listStr1 = null;
    SLLList<String> listStr2 = new SLLList<>(listStr1);
}
```

use the optional 'expected' attribute of Test annotation

# Checked EmptySLLException

- Say we don't allow copy from an empty SLL, and want to enforce it
- We define our own checked exception :

```
public class EmptySLLException extends Exception {  
  
    private String name;  
  
    public EmptySLLException(String name) {  
        super(name);  
        this.name = name;  
    }  
  
    @Override  
    public String getMessage() {  
        return name + " is empty, " + super.getMessage();  
    }  
}
```

inherit Exception functionality

pass the parameter to Exception

override the getMessage() method of Exception

call the overriden getMessage() method of Exception

# Throws EmptySLLListException

```
/**  
 * Creates a deep copy SLLList of other.  
 * @param other an SLLList object  
 * @throws IllegalArgumentException if other is null  
 * @throws EmptySLLListException if other is empty  
 */  
public SLLList<T> other) throws EmptySLLListException {  
  
    if (other == null) {  
        throw new IllegalArgumentException("other is null");  
    }  
  
    if (other.size() == 0) {  
        throw new EmptySLLListException("other");  
    }  
  
    sentinel = new Node( i: null, n: null);  
    size = 0;  
  
    Node p = sentinel;  
    Node q = other.sentinel;  
    while (q.next != null) {  
        this.addLast(q.next.item);  
        q = q.next;  
    }  
}
```

must advertise

pass the parameter name

# Gotta catch 'em all

---

```
// Try-Catch InvalidArgumentException and EmptySLLListException
SLLList<String> listStr1 = new SLLList<>();
try {
    SLLList<String> listStr2 = new SLLList<>(listStr1);
} catch (IllegalArgumentException iae) {
    System.out.println("Copy failed, " + iae.getMessage());
} catch (EmptySLLListException ese) { ←
    System.out.println("Cannot copy, " + ese.getMessage());
}
```

must handle the checked exception  
(since we don't throw it)

try by yourself if you set  
EmptySLLListException to be unchecked!

# Thank you for your attention !

---

- In this lecture, you have learned to:
  - use or not use null values
  - know the difference between checked and unchecked exceptions in Java
  - use exceptions to signal special results
  - write a copy constructor doing deep copy
- Please continue to Lecture Quiz 6 and Lab 6:
  - to do Lab Exercise 6.1 - 6.3, and
  - to do Exercise 6.1 - 6.3



# Advanced Object-Oriented Programming

CPT204 – Lecture 8  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

# CPT204 Advanced Object-Oriented Programming

## Lecture 8

**Specification 2,  
Linked List 4**

# Welcome !

---

- Welcome to Lecture 8 !
- In this lecture we are going to
  - continue to learn about specification
    - discuss the properties of specs
    - learn what makes some specs better than others
  - construct a homemade list, this time using an array as the underlying data structure, while learning about
    - array copy
    - array resizing
    - generic arrays
    - object deletion

# Part 1: Three Dimensions for Comparing Specs

---

- In the first part of the lecture, we'll look at different specs for similar behaviors (e.g. finding an element in an array), and discuss the tradeoffs between them
- We'll look at three dimensions for comparing specs:
  - How **deterministic** it is: does the spec define only *a single possible output* for a given input, or does it allow the implementor to choose from *a set of legal outputs*?
  - How **declarative** it is: does the spec just *characterize what the output should be*, or does it *explicitly say how to compute the output*?
  - How **strong** it is: does the spec have *a small set* of legal implementations, or *a large set*?
- Not all specifications we might choose for a module are equally useful, and we'll explore what makes some specifications *better* than others

# Deterministic vs Underdetermined Specs (1)

---

- Recall the two example implementations of `find` we began with in previous lecture:

```
static int findFirst(int[] arr, int val) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == val) return i;
    }
    return arr.length;
}

static int findLast(int[] arr, int val) {
    for (int i = arr.length - 1 ; i >= 0; i--) {
        if (arr[i] == val) return i;
    }
    return -1;
}
```

- The subscripts `First` and `Last` are not actual Java syntax: we're using them here to distinguish the two implementations for the sake of discussion  
In the actual code, both implementations should be Java methods called `find`

# Deterministic vs Underdetermined Specs (2)

---

- Here is one possible specification of find:

```
static int findExactlyOne(int[] arr, int val)
```

**requires:** val occurs exactly once in arr

**effects:** returns index i such that arr[i] == val

- This specification is **deterministic**: when presented with a state satisfying the precondition, the outcome is completely determined
  - Only one return value and one final state are possible
  - There are no valid inputs for which there is more than one valid output
- Both  $\text{find}_{\text{First}}$  and  $\text{find}_{\text{Last}}$  **satisfy** the specification, so if this is the specification on which the clients relied, the two implementations are equivalent and substitutable for one another

# Deterministic vs Underdetermined Specs (3)

---

- Here is a slightly different specification:

```
static int findOneOrMore, AnyIndex (int[] arr, int val)
```

**requires:** val occurs in arr

**effects:** returns index i such that arr[i] == val

- This specification is **not deterministic**: it *doesn't* say which index is returned if val occurs more than once
  - It simply says that if you look up the entry at the index given by the returned value, you'll find val
  - This specification allows *multiple valid outputs* for the *same input*
- Note that this is different from nondeterministic, for example, when the code's behavior depends on a random number
  - To avoid the confusion, we'll refer to specifications that are **not deterministic** as **underdetermined**

## Deterministic vs Underdetermined Specs (4)

---

- This underdetermined find spec is again **satisfied** by *both* `findFirst` and `findLast`, each resolving the underdeterminedness in its own way !
- A client of `findOneOrMore, AnyIndex` spec **cannot** rely on which index will be returned if `val` appears more than once
- The spec would be satisfied by a nondeterministic/random implementation, too — for example, one that tosses a coin to decide whether to start searching from the beginning or the end of the array
- But in almost all cases we'll encounter, underdeterminism in specifications offers *a choice* that is made by the implementer at implementation time

# Declarative vs Operational Specs (1)

---

- **Operational** specifications give a series of steps that the method performs
  - pseudocode descriptions are operational
- **Declarative** specifications don't give details of intermediate steps
  - instead, they just give properties of the final outcome and how it's related to the initial state
- Almost always, *declarative* specifications are *preferable*
  - they're usually shorter, easier to understand, and
  - most importantly, they don't inadvertently *expose implementation details* that a client may rely on (and then find no longer hold when the implementation is changed)



some of our specs do this,  
for example when I ask for recursive implementation.  
it's for educative purpose; generally, don't do that!

## Declarative vs Operational Specs (2)

---

- For example, if we want to allow either implementation of find, we would not want to say in the spec that the method “*goes down the array until it finds val*”
  - since aside from being rather vague, this spec suggests that the search proceeds from lower to higher indices and that the lowest will be returned, which perhaps the specifier did **not** intend
- One reason programmers sometimes lapse into operational specifications is because they're using the spec comment to explain the implementation for a maintainer
  - don't do that!
  - explain to the user first, not just to the maintainer
  - when it's necessary, use comments within the body of the method, **not** in the spec comment

# Declarative vs Operational Specs (3)

- For a given specification, there may be many ways to express it declaratively:
  - a. static boolean **startsWith**(String str, String prefix)  
**effects:** returns true if and only if there exists String suffix such that prefix + suffix == str
  - b. static boolean **startsWith**(String str, String prefix)  
**effects:** returns true if and only if there exists integer i such that str.substring(0, i) == prefix
  - c. static boolean **startsWith**(String str, String prefix)  
**effects:** returns true if the first prefix.length() characters of str are the characters of prefix, false otherwise
- It's up to us to choose the clearest specification for clients and maintainers of the code

# Stronger vs Weaker Specs (1)

---

- Suppose you want to change a method — either how its implementation behaves, or the specification itself
  - There are already clients that depend on the method's current specification
  - How do you compare the behaviors of two specifications to decide whether it's safe to replace the old spec with the new spec?
- A specification S2 is ***stronger than or equal*** to a specification S1 if
  - S2's precondition is ***weaker than or equal to*** S1's, and
  - S2's postcondition is ***stronger than or equal*** to S1's for the states that satisfy S1's precondition
- If this is the case, then an implementation that satisfies S2 can be used to satisfy S1 as well, and it's *safe* to *replace* S1 with S2 in your program

## Stronger vs Weaker Specs (2)

- These two rules embody several ideas: they tell you that you can *always weaken the precondition*; placing fewer demands on a client will never upset them; and you can always *strengthen the postcondition*, which means making more promises
- For example, this spec for find:

- static int **find**<sup>ExactlyOne</sup>(int[] a, int val)  
**requires:** val occurs ***exactly once*** in a  
**effects:** returns index i such that a[i] == val

the two rules can be hard to understand, but memorize these parts, these actually make sense !

can be replaced with:

- static int **find**<sup>OneOrMore, AnyIndex</sup>(int[] a, int val)  
**requires:** val occurs ***at least once*** in a  
**effects:** returns index i such that a[i] == val

user already gave exactly once, automatically fulfil at least once

which has **a weaker precondition**

## Stronger vs Weaker Specs (3)

- In turn, that last spec:

- static int **find**<sup>OneOrMore, AnyIndex</sup>(int[] a, int val)  
**requires:** val occurs at least once in a  
**effects:** returns index i such that a[i] == val

can be replaced with:

- static int **find**<sup>OneOrMore, FirstIndex</sup>(int[] a, int val)  
**requires:** val occurs at least once in a  
**effects:** returns *lowest* index i such that a[i] == val

which has *a stronger postcondition*

user promised an index before,  
now also promised an index,  
in particular the lowest

# In-Class Quiz 1

---

- Consider the following specification

```
static int findCanBeMissing(int[] a, int val)
```

**requires:** *nothing*

**effects:** returns index  $i$  such that  $a[i] == \text{val}$ , or  $-1$  if no such  $i$

and compare with:

```
static int findOneOrMore,FirstIndex(int[] a, int val)
```

**requires:**  $\text{val}$  occurs at least once in  $a$

**effects:** returns *lowest* index  $i$  such that  $a[i] == \text{val}$

- which is true about their *precondition* and *postcondition* ?

# Incomparable

---

- The specification:

```
static int findCanBeMissing(int[] a, int val)
```

**requires:** *nothing*

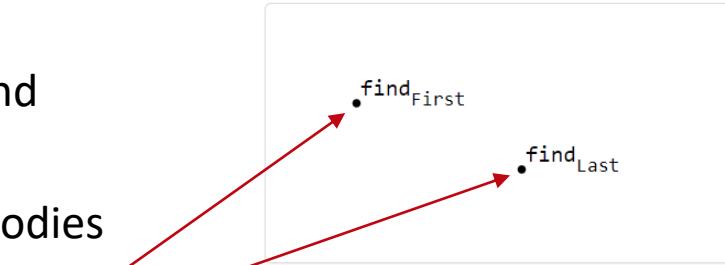
**effects:** returns index  $i$  such that  $a[i] == val$ , or  $-1$  if no such  $i$

compared to **find**<sup>OneOrMore,FirstIndex</sup>

- again the precondition is weaker
- but for inputs that satisfy **find**<sup>OneOrMore,FirstIndex</sup>'s precondition,  
the postcondition is also weaker: the requirement for lowest index has been  
removed
- Neither of these two specifications is stronger than the other: they are  
**incomparable**

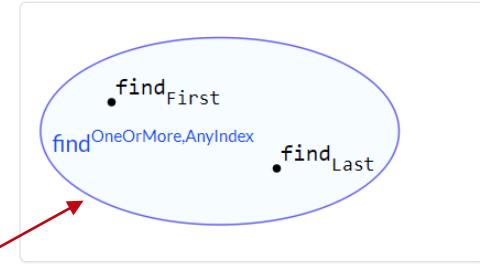
# Diagramming Specification (1)

- Imagine (very abstractly) the space of all possible Java methods
- Each point in this space represents a method implementation
- First we'll diagram  $\text{find}_{\text{First}}$  and  $\text{find}_{\text{Last}}$  defined previously
- Look back at the code and see that  $\text{find}_{\text{First}}$  and  $\text{find}_{\text{Last}}$  are **not** specs
  - they are *implementations*, with method bodies that implement their actual behavior
  - so we denote them as points in the space



## Diagramming Specification (2)

- A specification defines a region in the space of all possible implementations
- A given implementation either behaves according to the spec, satisfying the precondition-implies-postcondition contract (it is inside the region),  
*or* it does not (outside the region)
- Both  $\text{find}_{\text{First}}$  and  $\text{find}_{\text{Last}}$  satisfy  $\text{find}^{\text{OneOrMore}, \text{AnyIndex}}$ , so they are *inside* the region defined by that spec



## Diagramming Specification (3)

---

- We can imagine clients looking in on this space: the specification acts as a *firewall*
  - Implementors have the freedom to move around inside the spec, changing their code without fear of upsetting a client
    - This is crucial in order for the implementer to be able to improve the performance of their algorithm, the clarity of their code, or to change their approach when they discover a bug, etc
  - Clients don't know which implementation they will get
    - They must respect the spec, but also have the freedom to change how they're using the implementation without fear that it will suddenly break

# Diagramming Specification (4)

---

- How will similar specifications relate to one another?
- Suppose we start with specification S1 and adapt it to create a new specification S2
  - If S2 is stronger than S1, how will these specs appear in our diagram?
- Let's start by **strengthening the postcondition**
  - If S2's postcondition is now stronger than S1's postcondition, then S2 is the stronger specification
  - Think about what strengthening the postcondition means for implementers
  - It means they have less freedom, because the requirements on their output are stronger

# Diagramming Specification (5)

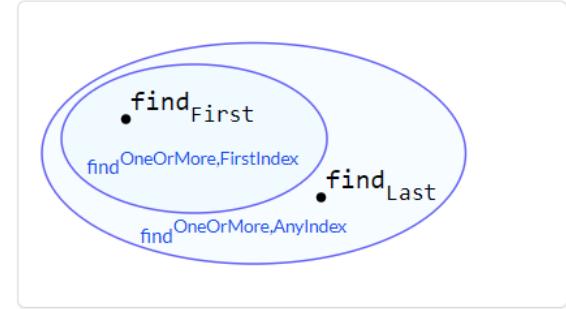
---

- If we **weaken the precondition**, which again makes S2 a stronger specification, then implementations will have to handle new inputs that were previously excluded by the spec
  - If they behaved badly on those inputs before, we wouldn't have noticed, but now their bad behavior is exposed
- So when S2 is stronger than S1, it defines a smaller region in this diagram
  - Fewer implementations satisfy S2 than S1
  - Every implementation that satisfies S2 also satisfies S1, though, so the smaller region S2 is nested inside S1



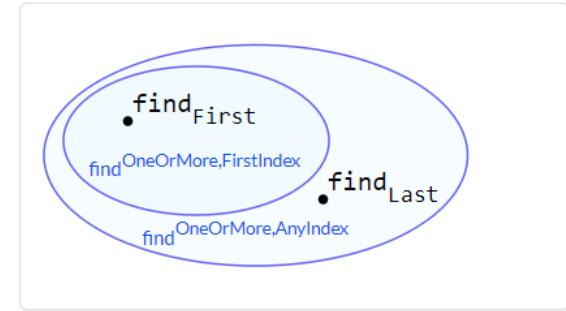
# Diagramming Specification (6)

- Now let's see how this works with our find specifications
- Where does  $\text{find}^{\text{OneOrMore}, \text{FirstIndex}}$  fit on the diagram?
  - Is it stronger or weaker?
- Implementations can satisfy  $\text{find}^{\text{OneOrMore}, \text{AnyIndex}}$  by returning any index  $i$ ,  
but now the spec demands the lowest index  $i$
- So there are now implementations *inside*  $\text{find}^{\text{OneOrMore}, \text{AnyIndex}}$  but *outside*  $\text{find}^{\text{OneOrMore}, \text{FirstIndex}}$



# Diagramming Specification (7)

- Could there be implementations inside  $\text{find}^{\text{OneOrMore}, \text{FirstIndex}}$  but outside  $\text{find}^{\text{OneOrMore}, \text{AnyIndex}}$ ?
  - No, all of those implementations satisfy a stronger postcondition than what  $\text{find}^{\text{OneOrMore}, \text{AnyIndex}}$  demands
- In our figure, since  $\text{find}_{\text{Last}}$  iterates from the end of the array arr, it does not satisfy  $\text{find}^{\text{OneOrMore}, \text{FirstIndex}}$  and is outside that region
- Another specification S3 that is neither stronger nor weaker than S1 ***might overlap*** (such that there exist implementations that satisfy only S1, only S3, and both S1 and S3) or ***might be disjoint***
  - In both cases, S1 and S3 are ***incomparable***



# Designing Good Specifications

---

- What makes a good method?
  - Designing a method means primarily writing a specification
- About the form of the specification: it should obviously be succinct, clear, and well-structured, so that it's easy to read.
- The content of the specification, however, is harder to prescribe
  - There are no infallible rules, but there are some useful guidelines, as follows in the rest of this first part of the lecture

# The Specification should be Coherent (1)

---

- The spec shouldn't have lots of different cases
  - Long argument lists, deeply nested if-statements, and boolean flags are all signs of trouble
- Consider this specification:

```
static int sumFind(int[] a, int[] b, int val)
```

**effects:** returns the sum of all indices in arrays a and b at which val appears

- Is that a well-designed procedure?
  - Probably not: it's **incoherent**, since it does several things (finding in two arrays and summing the indexes) that are not really related
  - It would be better to use *two separate procedures*, one that finds the indexes, and the other that sums them

# The Specification should be Coherent (2)

---

- Here's another example:

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

/**
 * Update longestWord to be the longest element of words, and print
 * the number of elements with length > LONG_WORD_LENGTH to the console.
 * @param words list to search for long words
 */
public static void countLongWords(List<String> words)
```

- In addition to *terrible use of global variables* and *printing instead of returning*, the specification is **not coherent** — it does two different things, counting words and finding the longest word
- Separating those two responsibilities into two different methods will make them simpler (easy to understand) and more useful in other contexts (ready for change)

# The Results of a Call should be Informative

---

- Consider the specification of a method that puts a value in a map, where keys are of some type K and values are of some type V:

```
static V put(Map<K,V> map, K key, V val)
```

**requires:** val may be null, and map may contain null values

**effects:** inserts (key, val) into the mapping, overriding any existing mapping for key, and returns old value for key, unless none, in which case it returns null

- Note that the precondition does ***not*** rule out null values so the map can store nulls
- But the postcondition ***uses*** null as a special return value for a missing key
  - This means that if null is returned, you *can't tell* whether the key was not bound previously, *or* whether it was in fact bound to null
  - This is not a very good design, because the return value is *useless* unless you know for sure that you didn't insert nulls

# The Specification should be Strong Enough

---

- Spec should give clients a strong enough guarantee in the general case — it needs to satisfy their basic requirements
  - we must use extra care when specifying *the special cases*, to make sure they don't undermine what would otherwise be a useful method
- For example, there's no point throwing an exception for a bad argument but allowing arbitrary mutations, because a client won't be able to determine what mutations have actually been made :

```
static void addAll(List<T> list1, List<T> list2)
```

**effects:** adds the elements of list2 to list1, unless it encounters a null element,  
at which point it throws a NullPointerException

- If a NullPointerException is thrown, the client is left to figure out on their own *which elements of list2 actually made it to list1*

# The Specification should be Weak Enough

---

- Consider this specification for a method that opens a file:

```
static File open(String filename)  
  effects: opens a file named filename
```

- This is a bad specification
- It lacks important details: is the file opened for reading or writing?
  - Does it already exist or is it created?
  - And it's ***too strong***, since there's no way it can guarantee to open a file
  - The process in which it runs may lack permission to open a file, or there might be some problem with the file system beyond the control of the program
  - Instead, the specification should say something much weaker: that it attempts to open a file, and if it succeeds, the file has certain properties

# The Specification should use Abstract Types where possible

---

- In Java, we can distinguish between more abstract notions like a List or Set and particular implementations like ArrayList or HashSet (will talk about this in future lecture in detail)
- Writing our specification with abstract types gives more freedom to both the client and the implementer
- In Java, this often means using an *interface* type, like Map or Reader, instead of specific implementation types like HashMap or FileReader, for example this specification:

```
static ArrayList<T> reverse(ArrayList<T> list)
```

**effects:** returns a new list which is the reversal of list, i.e.

```
newList[i] == list[n-i-1] for all 0 <= i < n, where n = list.size()
```

- This forces the client to pass in an ArrayList, and forces the implementer to return an ArrayList, even if there might be alternative List implementations that they would rather use
- Since the behavior of the specification doesn't depend on anything specific about ArrayList, it would be **better** to write this spec in terms of the more abstract List

# Precondition or Postcondition? (1)

---

- Another design issue is whether to use a precondition, and if so, whether the method code should attempt to make sure the precondition has been met before proceeding
- In fact, the most common use of preconditions is to demand a property precisely because it would be hard or expensive for the method to check it
- As mentioned above, a non-trivial precondition inconveniences clients, because they have to ensure that they don't call the method in a bad state (that violates the precondition); if they do, there is no predictable way to recover from the error
- That's why the Java API classes, for example, tend to specify (as a postcondition) that they throw unchecked exceptions when arguments are inappropriate
  - This approach makes it easier to find the bug or incorrect assumption in the caller code that led to passing bad arguments
  - In general, it's better to **fail fast**, as close as possible to the site of the bug, rather than let bad values propagate through a program far from their original cause

## Precondition or Postcondition? (2)

---

- Sometimes, it's not feasible to check a condition without making a method unacceptably slow, and a precondition is often necessary in this case
- If we wanted to implement the find method using binary search, we would have to require that the array be sorted
  - Forcing the method to actually check that the array is sorted would defeat the entire purpose of the binary search: to obtain a result in logarithmic and not linear time
- The decision of whether to use a precondition is an engineering judgment
- The key factors are the cost of the check (in writing and executing code), and the scope of the method
  - If it's only called locally in a class, the precondition can be discharged by carefully checking all the sites that call the method
  - But if the method is public, and used by other developers, it would be less wise to use a precondition — instead, like the Java API classes, you should throw an exception

# About Access Control (1)

---

- The decision to make a method **public** or **private** is actually a decision about the ***contract*** of the class
- Public methods are freely accessible to other parts of the program
  - Making a method public advertises it as a service that your class is willing to provide
- If you make *all* your methods public — including helper methods that are really meant only for local use within the class — then other parts of the program may come to depend on them, which will make it harder for you to change the internal implementation of the class in the future
  - Your code won't be as ready for change

## About Access Control (2)

---

- Making *internal helper methods* public will also add clutter to the visible interface your class offers
  - Keeping internal things private makes your class's public interface smaller and more coherent (meaning that it does one thing and does it well)
  - Your code will be easier to understand
- We will see even stronger reasons to use private in the next lectures, when we start to write classes with persistent internal state
  - Protecting this state will help keep the program safe from bugs

# About Static vs Instance Methods (1)

---

- Static methods are **not** associated with any particular instance of a class, while instance methods (declared without the static keyword) must be called on a particular object
- Specifications for instance methods are written just the same way as specifications for static methods, but they will often refer to properties of the instance on which they were called
- For example, by now we're very familiar with this specification:

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

## About Static vs Instance Methods (2)

---

- Instead of using an `int[]`, what if we had a class `IntArray` designed for storing arrays of integers?
- The `IntArray` class might provide an instance method with the specification:

```
int find(int val)
```

**requires:** val occurs in this array

**effects:** returns index i such that the value at index i in this array is val

## In-Class Quiz 2

---

- Which of the following is a sign of a good specification?
  - the specification is declarative
  - the specification is operational
  - the specification is as strong as possible
  - the specification is as weak as possible
  - the implementation is allowed to ignore invalid arguments in the specification
  - the implementation is allowed to use different algorithms depending on the arguments
  - the specification is making use of the reader's knowledge of the implementation

## Part 2: Array-based List

---

- For the last few weeks, we have built our homemade list using node and pointer to another node (link-based) as our underlying data structure
  - another popular underlying data structure is *array*
- In this second part of the lecture, as a running example, before we go into more sophisticated data structure, let us once again build a homemade list, but now *using array*
  - for completeness, let us briefly review array

# Review: Arrays (1)



- Arrays are fixed-length sequences of another type
- To declare an array variable and construct an array value to assign to it:
  - `int[] a = new int[100];`
  - it includes all possible int array values, but once an array is created, we **cannot** change its length
- Operations on array types include:
  - indexing              `a[2]` ← index starts from 0
  - assignment            `a[2] = 0`
  - length                `a.length` ← this is not a method call  
array does **not** have methods

# Review: Arrays (2)

- Initialize an array

- o `arr1 = new int[5];`

each entry will get a default value  
for int, the default value is 0

- o `arr2 = new int[]{1, 2, 3, 4, 5};`

no need to specify the length

- o `int[] arr3 = {9, 10, 11, 12, 13};`

no need to use new if you  
also declare the variable

- Example:

```
int[] z = null;  
int[] x, y;  
  
x = new int[]{1, 2, 3, 4, 5};  
y = x;  
x = new int[]{6, 7, 8, 9};  
y = new int[3];  
z = new int[0];
```

try running in Java Visualizer

aliasing, y points to x's array as well

at this point, no one points to {1, 2, 3, 4, 5}  
and it's deleted from memory by garbage collector

an empty array

# Arraycopy

---

- Two ways to copy the elements of an array:
  - Element by element using a loop
  - Using arraycopy
- Arraycopy takes 5 parameters, in order:
  - Source array
  - Start position in source
  - Target array
  - Start position in target
  - Number to copy
- Example: `System.arraycopy(a, 0, b, 3, 2);`
  - it will copy 2 elements from a (starting from 0) to b (starting from 3)

# Why Array-Based List?

---

- We have build a link-based list, why bother to build another one with array?
  - Well, we want to practice using it, for later when we build more complicated data structures
  - But there is also another reason specific to list: retrieval with `get(int i)` is slow!
    - with linked-list, you have to scan from beginning/last to get to the desired position  $i$
    - one way to make this fast, is to change from link-based to array-based
- Retrieval from any position of an array is *very fast*
  - Independent of the array size

# ARList1

---

- Let us now start to build our homemade list using an array
  - start simple again, a list of *integers* ← we'll make it generic later
  - functionality includes:
    - empty constructor, size, get i-th item
    - addLast, getLast, delLast ← only add/delete from the back
- We will aim to give an alternate implementation of list using array
  - for the lab later, you will also give an alternate implementation of deque

# ARList1: Instance Variables, Empty Constructor, size

- As the underlying data structure of our list, we store the integers inside an int array

```
public class ARList1 {  
  
    private int[] items;  
    private int size;  
  
    /** Creates an empty list. */  
    public ARList1() {  
        items = new int[100];  
        size = 0;  
    }  
  
    /**  
     * Returns the number of items in the list.  
     * @return the size of the list  
     */  
    public int size() {  
        return size;  
    }  
}
```

start simple with integer-only array

using an array, we have to pick a fixed size,  
let's pick an arbitrary size, say 100

as before, we cache the size

next, we are going to **addLast** an integer  
any idea how we implement that using an array?

## ARList1: addLast

---

- One idea is to put the integer starting in index 0, and then index 1, 2, and so on, while incrementing the size

```
/**  
 * Adds item to the end of the list.  
 * @param item is an integer.  
 */  
  
public void addLast(int item) {  
    items[size] = item;  
    size++;  
}
```

notice that the next int goes into index size

next, we want to write **getLast** and **get(i)**  
how do we implement those using similar idea?

# ARList1: getLast, get(i)

- Remember that this is why we use array, we want get to be fast

```
/**  
 * Returns the last item at the back of the List.  
 * @return the last integer in the list.  
 */  
public int getLast() { return items[size - 1]; }  
  
/**  
 * Gets the item at the given index in the list.  
 * @param i is an index where 0 is the front.  
 * @return the ith item in the list.  
 */  
public int get(int i) { return items[i]; }
```

use the invariant that the last item is always in index size - 1

just return item in index i,  
in constant-time

next, we want to implement **delLast**  
how do you delete the item from the array?

# ARList1: delLast

- We don't have to actually delete the item from the array!

```
/**  
 * Deletes and returns the item at the back of the list.  
 * @return the last integer of the list to be deleted.  
 */  
  
public int delLast() {  
    int x = getLast();  
    size--;  
    return x;  
}
```

by decrementing the size and our invariant that the **last item is in size - 1**, we have effectively "delete" the item

the user of the list does **not** know that the item is actually still in the array, only we the implementer need to know this !

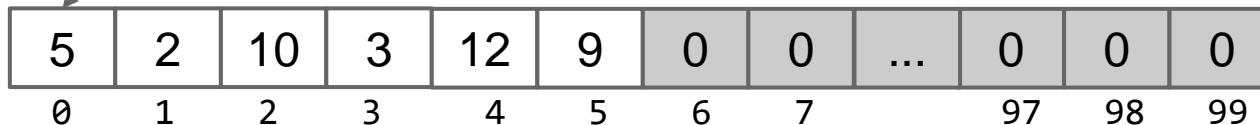
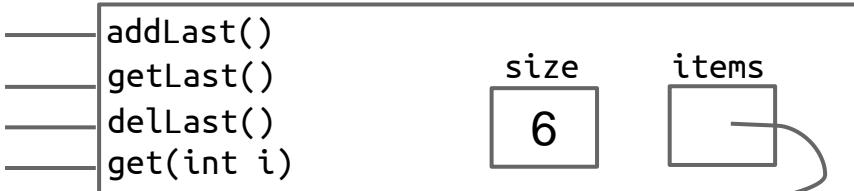
# ARList1: delLast

- We don't have to actually delete the item from the array!

```
/**  
 * Deletes and returns the item at the back of the list.  
 * @return the last integer of the list to be deleted.  
 */  
  
public int delLast() {  
    int x = getLast();  
    size--;  
    return x;  
}
```

by decrementing the size and our invariant that the **last item is in size - 1**, we have effectively "delete" the item

the user of the list does **not** know that the item is actually still in the array, only we the implementer need to know this !



# ARList1: delLast

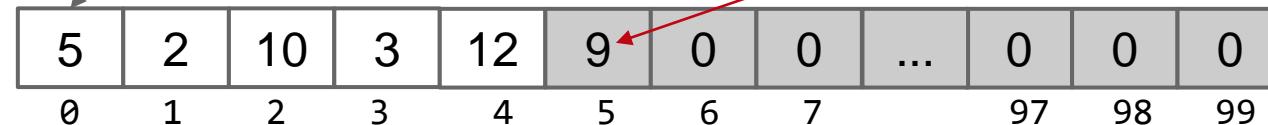
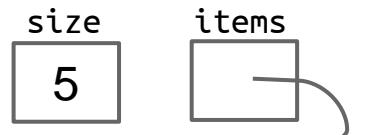
- We don't have to actually delete the item from the array!

```
/**  
 * Deletes and returns the item at the back of the list.  
 * @return the last integer of the list to be deleted.  
 */  
  
public int delLast() {  
    int x = getLast();  
    size--;  
    return x;  
}
```

by decrementing the size and our invariant that the **last item is in size - 1**, we have effectively "delete" the item

the user of the list does **not** know that the item is actually still in the array, only we the implementer need to know this !

addLast()  
getLast()  
delLast()  
get(int i)



we don't really need to change this to, for example, zero, to preserve the invariant

# A Limitation

---

- There is an obvious limitation to our ARList implementation so far:
  - it cannot hold more than 100 items
  - and remember that the length of an array is fixed

what should we do if we want to store arbitrary number of items in our list ?

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

size == items.length

- So, when the array items gets too full
  - we create a new array a with new **capacity**
  - we copy using arraycopy to a
  - we set items to be a

```
int[] a = new int[capacity];
System.arraycopy(items, srcPos: 0, a, destPos: 0, size);
items = a;
```

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

size == items.length

- So, when the array **items** gets too full
  - we create a new array **a** with new **capacity**
  - we copy using **arraycopy** to **a**
  - we set **items** to be **a**
  - how large should the new capacity be?
    - size + 1 ?

# Resizing Array

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

`size == items.length`

- So, when the array `items` gets too full
  - we create a new array `a` with new **capacity**
  - we copy using `arraycopy` to `a`
  - we set `items` to be `a`
  - how large should the new capacity be?
    - `size + 1` ?

even though there is now a place to store the new item in index `size`, `items` is full again and the next time another new item gets added we have to `arraycopy` (expensive)

# Resizing Array

---

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

`size == items.length`

- So, when the array `items` gets too full
  - we create a new array `a` with new **capacity**
  - we copy using `arraycopy` to `a`
  - we set `items` to be `a`
  - how large should the new capacity be?
    - `size + 1` ?
    - `size + 100` ?

additive factor is also turned out to be not good

# Resizing Array

- One solution is called **resizing array**
  - we *don't* actually change the size of the array (because we cannot)
  - but we create a new array of bigger size, and copy the items

`size == items.length`

- So, when the array `items` gets too full
  - we create a new array `a` with new **capacity**
  - we copy using `arraycopy` to `a`
  - we set `items` to be `a`
  - how large should the new capacity be?
    - `size + 1` ?
    - `size + 100` ?
    - `2 * size`

it can be shown\* that the best way is with multiplicative factor, say 2 times

\* interested students can look up running time analysis of "amortized analysis of table doubling"

# Implementing Resizing Array: Array Doubling

- Instead of writing all the code in addLast, it is better to create a separate method

```
/** Resizes the underlying array to the target capacity. */
private void resize(int capacity) {
    int[] a = new int[capacity];
    System.arraycopy(items, srcPos: 0, a, destPos: 0, size);
    items = a;
}

/**
 * Adds item to the end of the list.
 * @param item is an integer.
 */
public void addLast(int item) {
    if (size == items.length) {
        resize(capacity: 2 * size);
    }
    items[size] = item;
    size++;
}
```

if the array is full,  
we double the array size

so now we can add item as many as we want,  
just like what a list is supposed to do !

# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !

any idea how to fix this ?

# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !
- Opposite yet similar: once it is less full, reduce the size of the array
  - create a new array with smaller capacity
  - copy the elements to the new array

how much is less full?  
what is the new capacity?

# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !
- Opposite yet similar: once it is less full, reduce the size of the array
  - create a new array with smaller capacity
  - copy the elements to the new array
  - how much is less full? what is the new capacity?
    - if it is half full, halve the capacity ?

is this a good idea?

# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !
- Opposite yet similar: once it is less full, reduce the size of the array
  - create a new array with smaller capacity
  - copy the elements to the new array
  - how much is less full? what is the new capacity?
    - if it is half full, halve the capacity ?

this is a bad idea !  
it is full after delete and resize  
a sequence of add-deletes  
will result in copy all the time



# Array Halving

---

- Suppose you addLast and double the array until it of size 1,000,000,000
  - and then delLast and remove 990,000,000 items
  - if you are not doing anything, you waste memory,
  - and this code is not efficient in space !
- Opposite yet similar: once it is less full, reduce the size of the array
  - create a new array with smaller capacity
  - copy the elements to the new array
  - how much is less full? what is the new capacity?
    - if it is half full, halve the capacity ?
    - if it is one-quarter full, halve the capacity

it is half-full after resize

You will implement this idea in Lab !

## ARList2: Generic ARList (1)

- Finally, let us make our ARList generic, accepting any type T of items
  - as we previously did, we add <T> and change the int item into T item
  - with the exception of:

```
/** Creates an empty list. */
public ARList2() {
    items = new T[100];
    size = 0;
}
```

Type parameter 'T' cannot be instantiated directly

this is because in Java,  
generic arrays are not allowed,  
causing generic array creation error

## ARList2: Generic ARList (2)

- Finally, let us make our ARList generic, accepting any type T of items
  - as we previously did, we add <T> and change the int item into T item
  - with the exception of:

```
/** Creates an empty list. */
public ARList2() {
    items = (T[]) new Object[100];
    size = 0;
}
```

to do this, we simply create  
an array of objects, and then  
**cast** it into array of T objects

```
/** Resizes the underlying array to the target capacity. */
private void resize(int capacity) {
    T[] a = (T[]) new Object[capacity];
    System.arraycopy(items, srcPos: 0, a, destPos: 0, size);
    items = a;
}
```

However, this will trigger  
a compiler warning !

## ARList2: Generic ARList (3)

- Finally, let us make our ARList generic, accepting any type T of items
  - as we previously did, we add <T> and change the int item into T item
  - with the exception of:

```
/** Creates an empty list. */
@SuppressWarnings("unchecked")
public ARList2() {
    items = (T[]) new Object[100];
    size = 0;
}
```

one solution is to add this **annotation** before method that casts generic array

```
/** Resizes the underlying array to the target capacity. */
@SuppressWarnings("unchecked")
private void resize(int capacity) {
    T[] a = (T[]) new Object[capacity];
    System.arraycopy(items, srcPos: 0, a, destPos: 0, size);
    items = a;
}
```

we will discuss more about this in future lecture

## ARList2: delLast

---

- And on delLast:
  - unlike our integer-based ARList1, we actually want to **null out** deleted items

```
/**  
 * Deletes and returns the item at the back of the list.  
 * @return the last item of the list to be deleted.  
 */  
public T delLast() {  
    T x = getLast();  
    items[size - 1] = null; // ←  
    size--;  
    return x;  
}
```

if we don't do this,  
there still is reference to the deleted object  
and Java garbage collector won't delete it

# Thank you for your attention !

---

- In this lecture, you have learned to:
  - write declarative specs and to compare spec strength
  - write coherent, useful specifications of appropriate strength
  - use an array as an underlying dynamic data structure
  - resize an array efficiently in time and space : expand and shrink
  - create generic arrays
  - delete an object from an array completely
- Please continue to Lecture Quiz 8 and Lab 8:
  - to do Lab Exercise 8.1 - 8.4, and
  - to do Exercise 8.1 - 8.4



# Advanced Object-Oriented Programming

CPT204 – Lecture 9  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming

### Lecture 9

**Defensive Programming,  
Immutability, Iterator, Disjoint Sets**

# Welcome !

---

- Welcome to Lecture 9 !
- In this lecture we are going to
  - learn about how to avoid debugging, in particular, using defensive programming by assertion
  - continue to learn about immutability
  - learn about iterator
  - learn about a sophisticated data structure called Disjoint Sets (DS)

# Part 1: Avoiding Debugging

---

- The first part of the lecture is about how to avoid debugging entirely, or keep it easy when you have to do it
- We're going to discuss about two general ways to avoid painful debugging:
  - **Make bugs impossible by design**

We'll review some techniques that we've already talked about:  
static checking, dynamic checking, and immutability

- *we will talk more about immutability in second part*

- **Localize bugs to a small part of the program,**  
so that finding the bug becomes easier,  
with the following ideas:
  - *assertions*
  - *incremental development*
  - *scope minimization*

# First Defense: Make Bugs Impossible

- The best defense against bugs is to make them impossible by design
- **Static checking** is one way that we've already talked about
  - For example, Java prevents you from inadvertently passing an int value to a function expecting a String parameter, and it won't even allow your code to compile
  - Static checking eliminates many bugs by catching them at compile time
- We also saw some examples of **dynamic checking**
  - For example, Java makes array overflow bugs impossible by catching them dynamically
  - If you try to use an index outside the bounds of an array or a List, then Java automatically produces an error at runtime
  - Older languages like C and C++ **silently allow** the bad access, which leads to bugs and security vulnerabilities such as buffer overflow

# Make bugs impossible by Immutability (1)

- **Immutability** (immunity from change) is another design principle that prevents bugs
  - **An immutable type** is a type whose *values* can *never change* once they have been created
- Recall that String is an immutable type
  - There are no methods that you can call on a String that will change the sequence of characters that it represents — Strings can be passed around and shared without fear that they will be modified by other code
- Recall also Java gives us immutable references: variables declared with the keyword **final**, which can be assigned once but never reassigned
  - 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

# Make bugs impossible by Immutability (2)

- Consider this example:
  - `final char[] letters = new char[] {'a', 'e', 'i', 'o', 'u'};`
  - The vowels variable is declared `final`, but is it really unchanging?
- Which of the following statements will be illegal (caught statically by the compiler) and which will be allowed?
  - `letters = new char[] {'x', 'y', 'z'};`
  - `letters[0] = 'z';`
- You will have to answer in Lecture Quiz 9
- Be *careful* about what `final` means!
  - It *only* makes the reference immutable,  
*not necessarily* the object that the reference points to!

## Second Defense: Localize Bugs (1)

- If we can't prevent bugs, we can try to **localize** them to a small part of the program so that we don't have to look too hard to find the cause of a bug
- When localized to a single method or small module, bugs may be found simply by studying the program text
- We already talked about ***fail fast***: the earlier a problem is observed (the closer to its cause), the easier it is to fix
- We'll talk about a particular technique for failing fast, called **assertions**
  - Let's begin with a simple example:

```
/**  
 * @param x requires x >= 0  
 * @return approximation to square root of x  
 */  
public double sqrt(double x) { ... }
```

## Second Defense: Localize Bugs (2)

- Now suppose somebody calls `sqrt` with *a negative argument*
- What's the best behavior for `sqrt`?
  - Since the caller has failed to satisfy the requirement that  $x$  should be nonnegative, `sqrt` is *no longer* bound by the terms of its contract, so it is technically free to do whatever it wants: return an arbitrary value, enter an infinite loop, or even melt down the CPU
- Since the bad call indicates a bug in the caller, however, the most useful behavior would point out the bug as early as possible

## Second Defense: Localize Bugs (3)

- We do this by inserting a runtime check of the precondition, for example:

```
/**  
 * @param x  requires x >= 0  
 * @return approximation to square root of x  
 */  
public double sqrt(double x) {  
    if (! (x >= 0)) throw new IllegalArgumentException("required  
                           x >= 0, but was: " + x);  
    ...  
}
```

- When the precondition is not satisfied, this code terminates the program by *throwing an unchecked* `IllegalArgumentException`
  - The effects of the caller's bug are prevented from propagating

# Assertions (1)

---

- Checking preconditions is an example of **defensive programming**
- Real programs are rarely bug-free
  - Defensive programming offers a way to mitigate the effects of bugs even if you don't know where they are
- It is *common practice* to define a procedure for these kinds of defensive checks, usually called assert:  
`assert (x >= 0);`
- This approach abstracts away from what exactly happens when the assertion fails.
  - The failed assert might exit, might record an event in a log file, or might even email a report to a maintainer

## Assertions (2)

---

- Assertions have the added benefit of documenting an assumption about the state of the program at that point
  - To somebody reading your code, `assert (x >= 0)` says “at this point, it should always be true that  $x \geq 0$ .”
  - Unlike a comment, however, an assertion is executable code that enforces the assumption at runtime
- In Java, runtime assertions are a built-in feature of the language
  - The simplest form of the assert statement takes a boolean expression, and *throws an AssertionError* if the boolean expression evaluates to false:  
`assert x >= 0;`

## Assertions (3)

---

- An assert statement may also include *a description expression*, which is usually a string, but may also be a primitive type or a reference to an object
  - The description is printed in an error message when the assertion fails, so that it can be used to provide additional details to the programmer about the cause of the failure
  - The description follows the asserted expression, separated by a colon
    - For example:  
`assert (x >= 0) : "x is " + x;`
  - If  $x == -1$ , then this assertion fails with the error message: `x is -1` along with a stack trace that tells you where the assert statement was found in your code and the sequence of calls that brought the program to that point
    - This information is often enough to get started in finding the bug

## Assertions (4)

---

- A serious problem with Java assertions is that assertions are **off by default**
  - If you just run your program as usual, none of your assertions will be checked!
- Java's designers did this because checking assertions can sometimes be costly to performance
  - For example, a procedure that searches an array using binary search has a requirement that the array be sorted
  - Asserting this requirement requires scanning through the entire array, however, turning an operation that should run in logarithmic time into one that takes linear time
- You should be willing to pay this cost *during testing*, since it makes debugging much easier, but not after the program is released to users
- For most applications, however, assertions are not expensive compared to the rest of the code and the benefit they provide in bug-checking is worth that small cost in performance

## Assertions (5)

---

- So you have to enable assertions explicitly by passing **-ea** (which stands for enable assertions) to the Java virtual machine
  - In IntelliJ, enable assertions by going to Run → Edit Configurations → Configuration, putting -ea in the VM options, and click OK  
(may need to click Modify options → Add VM options first)

# Assertions and JUnit (1)

---

- It's always a good idea to have assertions turned on when you're running JUnit tests
- You can ensure that assertions are enabled using the following test case:

```
@Test(expected=AssertionError.class)
public void testAssertionsEnabled() {
    assert false;
}
```

- If assertions are turned on as desired, then assert false throws an AssertionError
  - The annotation (expected=AssertionError.class) on the test expects and requires this error to be thrown, so the test passes
  - If assertions are turned off, however, then the body of the test will do nothing, failing to throw the expected exception, and JUnit will mark the test as failing

## Assertions and JUnit (2)

---

- Note that the Java assert statement is a **different** mechanism than the JUnit methods `assertTrue()`, `assertEquals()`, etc
- They all assert a predicate about your code, but are designed for use in different contexts
- The assert statement should be used *in implementation code* for defensive checks inside the implementation
- JUnit `assert...()` methods should be used *in JUnit tests* to check the result of a test
- The assert statements don't run without `-ea`, but the JUnit `assert...()` methods always run

# What to Assert (1)

---

Here are some things you should assert:

- **Method argument requirements**, like we saw for `sqrt`
- **Method return value requirements**
  - This kind of assertion is sometimes called a self check
  - For example, the `sqrt` method might square its result to check whether it is reasonably close to `x`:

```
public double sqrt(double x) {  
    assert x >= 0;  
    double r;  
    ... // compute result r  
    assert Math.abs(r*r - x) < .0001;  
    return r;  
}
```

# What to Assert (2)

---

- **Covering all cases**

- If a conditional statement or switch does not cover all the possible cases, it is good practice to use an assertion to *block the illegal cases*:

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

- The assertion in the default clause has the effect of asserting that vowel must be one of the five vowel letters

## What to Assert (3)

---

- When should you write runtime assertions?
  - As you write the code, **not** after the fact
  - When you're writing the code, you have the **invariants** in mind
- If you postpone writing assertions, you're less likely to do it, and you're liable to omit some important invariants

## What *not* to Assert (1)

---

- Runtime assertions are not free
  - They can clutter the code, so they must be used judiciously
- Avoid trivial assertions, just as you would avoid uninformative comments
  - For example:

```
// don't do this:  
x = y + 1;  
assert x == y + 1;
```

- This assertion doesn't find bugs in your code
  - It finds bugs in the compiler or Java virtual machine, which are components that you **should trust** until you have good reason to doubt them
  - If an assertion is obvious from its local context, *leave it out*

## What *not* to Assert (2)

---

- Never use assertions to test conditions that are **external** to your program, such as the existence of files, the availability of the network, or the correctness of input typed by a human user
  - Assertions test the internal state of your program to ensure that it is within the bounds of its specification
- When an assertion fails, it indicates that the program has run off the rails, in some sense, into a state in which it was not designed to function properly
  - Assertion failures therefore indicate **bugs**
- External failures are not bugs and there is no change you can make to your program in advance that will prevent them from happening
  - External failures should be handled using **exceptions** instead

## What *not* to Assert (3)

---

- Many assertion mechanisms are designed so that assertions are executed only during testing and debugging and turned off when the program is released to users
  - Java's assert statement behaves this way
- Since assertions may be disabled, the correctness of your program should **never** depend on whether or not the assertion expressions are executed
  - In particular, asserted expressions should **not** have side-effects
  - For example, if you want to assert that an element removed from a list was actually found in the list, don't write it like this:

```
// don't do this:  
assert list.remove(x);
```

- If assertions are disabled, the entire expression is skipped, and x is never removed from the list
  - Write it like this instead:

```
boolean found = list.remove(x);  
assert found;
```

# Incremental Development

- A great way to localize bugs to a tiny part of the program is **incremental development**
  - Build only a bit of your program at a time and test that bit thoroughly before you move on
  - That way, when you discover a bug, it's more likely to be in the part that you just wrote, rather than anywhere in a huge pile of code
- Recall that our lectures on testing have talked about two techniques that help with incremental development:
  - **Unit testing:** when you test a module in isolation, you can be confident that any bug you find is in that unit — or maybe in the test cases themselves
  - **Regression testing:** when you're adding a new feature to a big system, run the regression test suite as often as possible. If a test fails, the bug is probably in the code you just changed

# Modularity

---

- **Modularity** means dividing up a system into components, or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system
  - The opposite of a modular system is a **monolithic** system — big and with all of its pieces tangled up and dependent on each other
- A program consisting of a single, very long `main()` function is monolithic — harder to understand and harder to isolate bugs in
  - By contrast, a program broken up into small functions and classes is more modular

# Encapsulation (1)

---

- **Encapsulation** means building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior and bugs in other parts of the system can't damage its integrity
- One kind of encapsulation is **access control**: using public and private to control the visibility and accessibility of your variables and methods
  - A *public* variable or method can be accessed by any code (assuming the class containing that variable or method is also public)
  - A *private* variable or method can only be accessed by code in the same class. Keeping things private as much as possible, especially for variables, provides encapsulation, since it limits the code that could inadvertently cause bugs

## Encapsulation (2)

---

- Another kind of encapsulation comes from **variable scope**
  - The scope of a variable is the portion of the program text over which that variable is defined, in the sense that expressions and statements can refer to the variable
  - A method parameter's scope is the body of the method
  - A local variable's scope (in Java) extends from its declaration to the closing curly brace of the block around the declaration
  - Keeping variable scopes ***as small as possible*** makes it much easier to reason about where a bug might be in the program

## Encapsulation (3)

---

- For example, suppose you have a loop like this:

```
for (i = 0; i < 100; i++) {  
    ...  
    doSomeThings();  
    ...  
}
```

and you've discovered that this loop keeps running forever, *i* never reaches 100

- Somewhere, somebody is changing *i*
  - But where?

## Encapsulation (4)

---

- If `i` is declared as a global variable like this:

```
public static int i;  
...  
for (i = 0; i < 100; i++) {  
    ...  
    doSomeThings();  
    ...  
}
```

then its scope is the entire program!

- It might be changed anywhere in your program: by `doSomeThings()`, by some other method that `doSomeThings()` calls, or by a concurrent thread running some completely different code

## Encapsulation (5)

---

- But if `i` is instead declared as a local variable with a narrow scope, like this:

```
for (int i = 0; i < 100; i++) {  
    ...  
    doSomeThings();  
    ...  
}
```

then the only place where `i` can be changed is within the `for` statement — in fact, only in the `...` parts that we've omitted

- You don't even have to consider `doSomeThings()` because `doSomeThings()` doesn't have access to this local variable

# Minimizing Variable Scope (1)

---

Minimizing the scope of variables is a powerful practice for bug localization

Here are a few rules that are good for Java:

- Always declare a loop variable in the for-loop initializer

- Don't declare it before the loop:

```
int i; // don't do this  
for (i = 0; i < 100; i++) {
```

This makes the scope of the variable the entire rest of the outer curly-brace block containing this code

- Do this instead:

```
for (int i = 0; i < 100; i++) {
```

which makes the scope of i limited just to the for loop

## Minimizing Variable Scope (2)

---

- Declare a variable only when you first need it and in the innermost curly-brace block that you can
  - Variable scopes in Java are **curly-brace blocks**, so put your variable declaration in the **innermost** one that contains all the expressions that need to use the variable
  - Don't declare all your variables at the start of the function — it makes their scopes unnecessarily large
  - But note that in languages without static type declarations, like Python and Javascript, the scope of a variable is normally the entire function anyway, so you **cannot** restrict the scope of a variable with curly braces

# Minimizing Variable Scope (3)

---

- Avoid global variables
  - Using global variables is very bad idea, especially as programs get large
  - Global variables are often unwisely used as a shortcut to provide a parameter to several parts of your program, but that's a **terrible** idea
  - It's better to just pass the parameter into the code that needs it, rather than putting it in global space where it can inadvertently reassigned

## Part 2: Mutability, Immutability, Iterator

---

- In the second part of the lecture, we are going to discuss more about the dangers of mutable data types, the advantages of immutability, and how to create an iterator
- Recall from when we discussed snapshot diagrams that some objects are immutable: once created, they always represent the same value
- Other objects are mutable: they have methods that change the value of the object
- String is an example of an immutable type
  - a String object always represents the same string
- StringBuilder is an example of a mutable type
  - it has methods to delete parts of the string, insert or replace characters, etc

# Mutability (1)

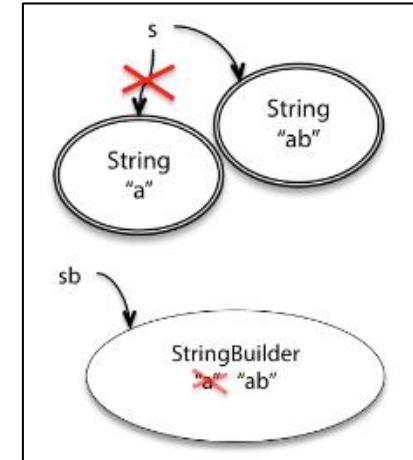
- Since String is immutable, once created, a String object always has the same value
  - To add something to the end of a String, you have to create a new String object:

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

- By contrast, StringBuilder objects are mutable
  - 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");
```

- StringBuilder has other methods as well, for deleting parts of the string, inserting in the middle, or changing individual characters



# Mutability (2)

---

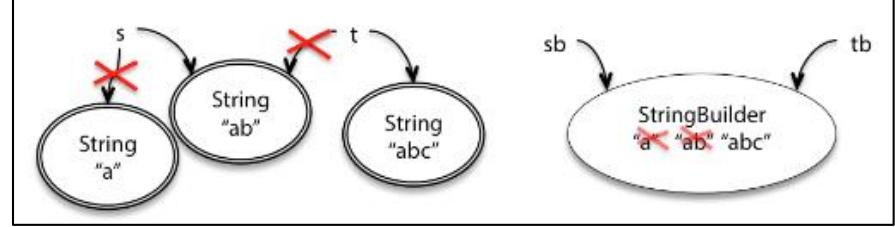
- *So what?*
  - In both cases, you end up with s and sb referring to the string of characters "ab"
- The difference between mutability and immutability doesn't matter much when there's only one reference to the object
  - But there are big differences in how they behave when *there are other references to the object*

# Mutability (3)

- 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");
```



- This shows that changing `t` had ***no effect*** on `s`,  
but changing `tb` ***affected*** `sb` ***too*** — possibly to the surprise of the programmer
- That's the essence of the problem we're going to look at in this lecture

# StringBuilder (1)

- Since we have the immutable String class already, why do we even need the mutable StringBuilder in programming?
  - A common use for it is to concatenate a large number of strings together
  - Consider this code:

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

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

## StringBuilder (2)

- 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:

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

- Getting good performance is one reason why we use mutable objects
- Another reason is convenient sharing: two parts of your program can communicate more conveniently by sharing a common mutable data structure

# Risks of Mutation

---

- Mutable types seem much more powerful than immutable types
  - If you were shopping in the Datatype Supermarket and you had to choose between a boring immutable String and a super-powerful-do-anything mutable StringBuilder, *why 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
- Let us look at the following two examples that illustrate why

# Risky Example 1: Passing Mutable Values (1)

---

- Let's start with a simple method that sums the integers in a list:

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

# Risky Example 1: Passing Mutable Values (2)

- Suppose we also need a method that sums the absolute values
- Following good DRY practice (Don't Repeat Yourself), the implementer writes a method that uses `sum()`:

```
/** @return the sum of the absolute values of the numbers in the list */
public static int sumOfAbsoluteValues(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);
}
```

- Notice that this method does its job by *mutating the list directly*
  - It seemed sensible to the implementer because it's more efficient to reuse the existing list

# In-Class Quiz 1

---

- But the resulting behavior will be very surprising to anybody who uses it!
  - Here is an example:

```
// meanwhile, somewhere else in the code
public static void main(String[] args) {
    // ...
    List<Integer> myData = Arrays.asList(-5, -2, -10);
    System.out.println(sumOfAbsoluteValues(myData));
    System.out.println(sum(myData));
}
```

- What will this code print?

# Risky Example 1: Passing Mutable Values (4)

---

- Let's think about the key lessons of the risky example 1 here:
  - *Is it 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.
  - *Is it easy to understand?*
    - When reading main(), what would you assume about sum() and sumOfAbsoluteValues()?
    - It is not visible to the reader that myData gets changed by one of them

## Risky Example 2: Returning Mutable Values (1)

- We just saw an example where passing a mutable object to a function caused problems : what about returning a mutable object?
- Let's consider Date, one of the built-in Java classes
  - Date happens to be a **mutable** type
  - Suppose we write a method that determines the first day of spring:

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

Simply think that this method somehow returns a Date object

- Here we're using the Groundhog algorithm  
(watch the movie Groundhog Day!)  
for determining when spring starts



## Risky Example 2: Returning Mutable Values (2)

---

- Clients start using this method, for example to plan their big parties:

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

- All the code works and people are happy
- Now, independently, two things happen

## Risky Example 2: Returning Mutable Values (3)

---

- First, the implementer of `startOfSpring()` realizes that the groundhog is starting to get annoyed from being constantly asked when spring will start
- So the code is rewritten to ask the groundhog at most once and then cache the groundhog's answer for future calls:

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

(Aside: note the use of a private static variable for the cached answer. Would you consider this a global variable or not?)

## Risky Example 2: Returning Mutable Values (4)

---

- Second, one of the clients of `startOfSpring()` decides that the actual first day of spring is too cold for the party, so the party will be exactly a month later instead:

```
// 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?
}
```

(Aside: this code also has a latent bug in the way it adds a month.  
Why? What does it implicitly assume about when spring starts?)

## Risky Example 2: Returning Mutable Values (5)

---

- What happens when these two decisions interact?
  - Even worse, think about who will first discover this bug — will it be `startOfSpring()`? Will it be `partyPlanning()`?
- The key lessons from this second example:
  - Is it safe from bugs? No — again we had a latent bug that reared its ugly head
  - Is it ready for change? Obviously the mutation of the date object is a change, but that's not the kind of change we're talking about when we say “ready for change”
    - Instead, the question is whether the code of the program can be easily changed without rewriting a lot of it or introducing bugs
    - Here we had two apparently independent changes by different programmers that interacted to produce a bad bug

# Avoiding Bug by Immutability (1)

---

- In both of these examples — the `List<Integer>` example and the `Date` example — 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
- In fact, 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**
- This example also illustrates why using mutable objects can actually be bad for performance
  - The simplest solution to this bug, which avoids changing any of the specifications or method signatures, is for `startOfSpring()` to always return a **deep copy** of the groundhog's answer:

```
return new Date(groundhogAnswer.getTime());
```

## Avoiding Bug by Immutability (2)

---

- This pattern is **defensive copying**
  - we'll see much more of it when we talk about abstract data types (ADT)
- The defensive copy means `partyPlanning()` can freely edit the returned date without affecting `startOfSpring()`'s cached date
- But defensive copying forces `startOfSpring()` to do extra work and 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 (1)

- Actually, 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 sumOfAbsoluteValues) and myData (in main)
  - One programmer (sumOfAbsoluteValues'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

# Aliasing is what makes mutable types risky (2)

- 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
- Draw snapshot diagrams on paper first, but your real goal should be to develop the snapshot diagram in your head, so you can visualize what's happening in the code

# Specifications for Mutating Methods (1)

- At this point it should be clear that when a method performs mutation, it is crucial to include that mutation in the method's spec
  - Now we've seen that even when a particular method doesn't mutate an object, that object's mutability can still be a source of bugs
- Here's an example of a *mutating* method:

```
static void sort(List<String> list)
requires: nothing
effects: puts list in sorted order, i.e. list[i] <= list[j]
           for all 0 <= i < j < list.size()
```

# Specifications for Mutating Methods (2)

- And an example of a method that ***does not mutate*** its argument:

```
static List<String> toLowerCase(List<String> list)
  requires: nothing
  effects:   returns a new list t where t[i] == list[i].toLowerCase()
```

- If the effects do **not** explicitly say that an input can be mutated, then let us assume mutation of the input is **implicitly disallowed**
  - Virtually all programmers would assume the same thing
  - Surprise mutations lead to terrible bugs

# Iterator (1)

---

- The next mutable object we're going to look at is **an iterator** — 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 *an enhanced for (... : ...) loop* to step through a List or array
- This code:

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

is rewritten by the compiler into something like this:

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

## Iterator (2)

---

- An iterator has two methods:
  - `next()` returns the next element in the collection
  - `hasNext()` tests whether the iterator has reached the end of the collection
- In addition, the `next()` method is a **mutator** method that ***not only*** returns an element ***but also advances*** the iterator so that the subsequent call to `next()` will return a different element

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



the iterator starts somewhere,  
usually the beginning of the data structure

list

|  |       |         |       |         |
|--|-------|---------|-------|---------|
|  | "You" | "shall" | "not" | "pass!" |
|--|-------|---------|-------|---------|

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



hasNext() asks, is there an item  
in where you are currently?

`iter.hasNext()`

list

|       |         |       |         |
|-------|---------|-------|---------|
| "You" | "shall" | "not" | "pass!" |
|-------|---------|-------|---------|

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



1) first, next() asks the iterator to return the item where it is currently

`iter.hasNext()`  
`iter.next()`

list

|       |         |       |         |
|-------|---------|-------|---------|
| "You" | "shall" | "not" | "pass!" |
|-------|---------|-------|---------|

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



2) then, the iterator moves  
(mutates) itself to the next place

`iter.hasNext()`  
`iter.next()`

list

|  |       |         |       |         |
|--|-------|---------|-------|---------|
|  | "You" | "shall" | "not" | "pass!" |
|--|-------|---------|-------|---------|

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



`iter.hasNext()`  
`iter.next()`  
`iter.hasNext()`

list    "You"    "shall"    "not"    "pass!"

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()
```

list

|  |       |         |       |         |
|--|-------|---------|-------|---------|
|  | "You" | "shall" | "not" | "pass!" |
|--|-------|---------|-------|---------|

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()
```

list

|  |       |         |       |         |
|--|-------|---------|-------|---------|
|  | "You" | "shall" | "not" | "pass!" |
|--|-------|---------|-------|---------|

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



list

|  |       |         |       |         |
|--|-------|---------|-------|---------|
|  | "You" | "shall" | "not" | "pass!" |
|--|-------|---------|-------|---------|

```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()
```

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



list

|  |       |         |       |         |
|--|-------|---------|-------|---------|
|  | "You" | "shall" | "not" | "pass!" |
|--|-------|---------|-------|---------|

```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()
```

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



list

|  |       |         |       |         |
|--|-------|---------|-------|---------|
|  | "You" | "shall" | "not" | "pass!" |
|--|-------|---------|-------|---------|

```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()
```

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



list

|  |       |         |       |         |
|--|-------|---------|-------|---------|
|  | "You" | "shall" | "not" | "pass!" |
|--|-------|---------|-------|---------|

```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()
```

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure



list

|  |       |         |       |         |
|--|-------|---------|-------|---------|
|  | "You" | "shall" | "not" | "pass!" |
|--|-------|---------|-------|---------|

```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()
```

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure

list

|       |         |       |         |
|-------|---------|-------|---------|
| "You" | "shall" | "not" | "pass!" |
|-------|---------|-------|---------|



```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()
```

# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure

list

|       |         |       |         |
|-------|---------|-------|---------|
| "You" | "shall" | "not" | "pass!" |
|-------|---------|-------|---------|



false

```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()
```

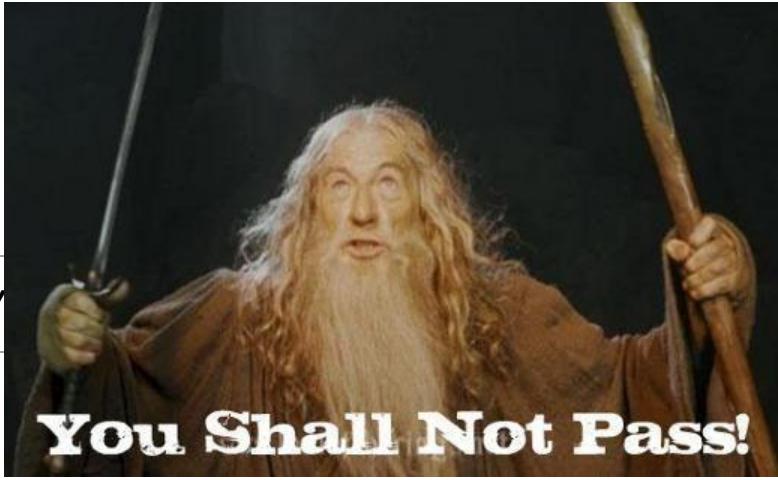
# How an Iterator Work

---

- Think of an iterator as a powerful wizard that can look inside your data structure

list

"Y



!"

```
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()  
iter.next()  
iter.hasNext()
```

# MyIterator (1)

- To better understand how an iterator works, here's a simple implementation of **an iterator for ArrayList<String>**
- Class MyIterator declaration and instance variables:

```
]/**
* A MyIterator is a mutable object that iterates over
* the elements of an ArrayList<String>, from first to last.
* This is just an example to show how an iterator works.
* In practice, you should use the ArrayList's own iterator
* object, returned by its iterator() method.
*/
public class MyIterator {
    private final ArrayList<String> list;
    private int index;
```

list[index] is the next element  
that will be returned by next()

index == list.size() means  
no more elements to return

# MyIterator (2)

- The constructor:

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

start iterating from the beginning

- The hasNext() method:

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

# MyIterator (3)

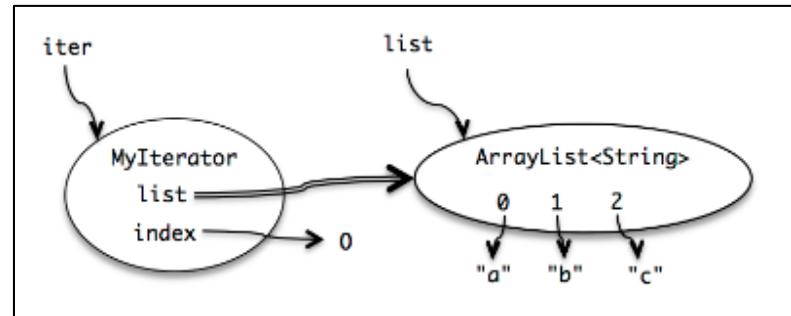
---

- The next() 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;  
}
```

## MyIterator (4)

- Here's a snapshot diagram showing a typical state for a MyIterator object in action:
- Recall that we draw the arrow from list with a double line to indicate it's final
- That means the arrow can't change once it's drawn
- But the ArrayList object it points to is mutable — elements can be changed within it — and declaring list as final has no effect on that



# Why Iterator?

---

- Why do iterators exist?
- There are many kinds of collection data structures (linked lists, maps, hash tables) with different kinds of internal representations
- The iterator concept allows a single uniform way to access them all, so that client code is simpler and the collection implementation can change without changing the client code that iterates over it
- Most modern languages (including Python, C#, and Ruby) use the notion of an iterator
- It's an effective **design pattern** (a well-tested solution to a common design problem)
  - We'll see many other design patterns as we move through the lecture

# Mutation Undermines an Iterator (1)

---

- Let's try using our iterator for a simple job
- Suppose we have a list of subjects represented as strings, like ["a100", "b100", "c200"]  
We want a method dropCourseA that will delete the "a" subjects/courses from the list, leaving the other subjects behind
- Following good practices, we first write the spec:

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

Note that dropCourseA has a clause in its contract that warns the client that its list argument will be mutated

## Mutation Undermines an Iterator (2)

- Next, following test-first programming, we devise a testing strategy that partitions the input space and choose test cases to cover that partition:

```
// Testing strategy:  
//   subjects.size: 0, 1, n  
//   contents: no axxx, one axxx, all axxx  
//   position: axxx at start, axxx in middle, axxx at end  
  
// Test cases:  
//   [] => []  
//   ["b100"] => ["b100"]  
//   ["c200", "d200", "e100"] => ["c200", "d200", "e100"]  
//   ["b100", "a100", "c100"] => ["b100", "c100"]  
//   ["a100", "a200", "a300"] => []
```

# Mutation Undermines an Iterator (3)

---

- Finally, we implement it:

```
public static void dropCourseA(ArrayList<String> subjects) {  
    MyIterator iter = new MyIterator(subjects);  
    while (iter.hasNext()) {  
        String subject = iter.next();  
        if (subject.startsWith("a")) {  
            subjects.remove(subject);  
        }  
    }  
}
```

## In-Class Quiz 2

---

- What is the output of dropCourseA on input ["a100", "a200", "a300"] ?

```
public static void dropCourseA(ArrayList<String> subjects) {  
    MyIterator iter = new MyIterator(subjects);  
    while (iter.hasNext()) {  
        String subject = iter.next();  
        if (subject.startsWith("a")) {  
            subjects.remove(subject);  
        }  
    }  
}
```

# Mutation Undermines an Iterator (4)

---

- The last test case fails:

```
// dropCourseA(["a100", "a200", "a300"])
//   expected [], actual ["a200"]
```

- We got the wrong answer: dropCourseA left a course behind in the list!
- Why?
  - Trace through what happens!
  - It will help to use a snapshot diagram showing the MyIterator object and the ArrayList object and update the diagram while you work through the code

# Mutation Undermines an Iterator (5)

---

- So we've seen that MyIterator can be broken by mutating the underlying list of strings while you're iterating over it
- But this isn't just a bug in our MyIterator!
  - The **built-in iterator** in ArrayList **suffers from the same problem** and so does the for loop that's syntactic sugar for it
  - The problem just has a different symptom — if you used this code instead:

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

then you'll get a ConcurrentModificationException

- The built-in iterator detects that you're changing the list under its feet and cries foul (How do you think it does that?)

# Mutation Undermines an Iterator (6)

- How can you fix this problem? One way is to use the **remove()** method of **Iterator** so that the iterator *adjusts its index* appropriately:

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

instead of `subjects.remove(subject)`

- This is actually more efficient as well, it turns out, because `iter.remove()` already knows where the element it should remove is, while `subjects.remove()` had to search for it again
- But this doesn't fix the whole problem
  - What if there are *other Iterators* currently active over the same list?
  - They won't all be informed!

# Mutable objects can make simple contracts very complex (1)

---

- This is a fundamental issue with mutable data structures
  - Multiple references to the same mutable object (also called **aliases for the object**) may mean that multiple places in your program — possibly widely separated — are relying on that object to remain consistent
- To put it in terms of specifications, contracts can't be enforced in just one place anymore, e.g. between the client of a class and the implementer of a class
- Contracts involving mutable objects now depend on the good behavior of everyone who has a reference to the mutable object

## Mutable objects can make simple contracts very complex (2)

---

- As a symptom of this non-local contract phenomenon, consider the Java collections classes, which are normally documented with very clear contracts on the client and implementer of a class
  - Try to find where it documents the crucial requirement on the client that we've just discovered — that you **can't** modify a collection while you're iterating over it
  - Who takes responsibility for it? Iterator? List? Collection? Can you find it?
- The need to reason about global properties like this make it much harder to understand, and be confident in the correctness of, programs with mutable data structures
  - We still have to do it — for performance and convenience — but we pay a big cost in bug safety for doing so

# Mutable objects reduce changeability (1)

- Mutable objects make the contract between a client and an implementer more complicated and reduce the freedom of the client and implementer to change
- In other words, using objects that are allowed to change makes the code harder to change
- Here's an example to illustrate the point, where the crux of our example will be the specification for this method, which looks up a username in a university database and returns the user's 9-digit identifier:

```
/**  
 * @param username username of person to look up.  
 * @return the 9-digit university identifier for username.  
 * @throws NoSuchElementException if nobody with username is in university database  
 */  
public static char[] getUnivId(String username) throws NoSuchElementException {  
    // ... look up username in university's database and return the 9-digit ID  
}
```

## Mutable objects reduce changeability (2)

---

- Suppose we have a client using this method to print out a user's identifier:

```
char[] id = getUnivId("abcde");
System.out.println(id);
```

- Now, **both** the client and the implementer **separately decide to make a change**
- The client is worried about the user's privacy and decides to obscure the first 5 digits of the id:

```
char[] id = getUnivId("abcde");
for (int i = 0; i < 5; ++i) {
    id[i] = '*';
}
System.out.println(id);
```

## Mutable objects reduce changeability (3)

---

- The implementer is worried about the speed and load on the database, so the implementer introduces a cache that remembers usernames that have been looked up:

```
private static Map<String, char[]> cache = new HashMap<String, char[]>();  
  
public static char[] getUnivId(String username) throws NoSuchUserException {  
    // see if it's in the cache already  
    if (cache.containsKey(username)) {  
        return cache.get(username);  
    }  
    // ... look up username in university database ...  
    // store it in the cache for future lookups  
    cache.put(username, id);  
    return id;  
}
```

## Mutable objects reduce changeability (4)

---

- These two changes have created a subtle bug
- When the client looks up "abcde" and gets back a char array, now both the client and the implementer's cache are pointing to the same char array
- The array is **aliased**
  - That means that the client's obscuring code is actually overwriting the identifier in the cache
  - so future calls to `getUnivId("abcde")` will not return the full 9-digit number, like "928432033", but instead the obscured version "\*\*\*\*\*2033"

# Sharing a mutable object complicates a contract (1)

---

- If this contract failure went to software engineering court, it would be contentious.  
Who's to blame here? Was the client obliged not to modify the object it got back?  
Was the implementer obliged not to hold on to the object that it returned?
- Here's one (bad) way we could have clarified the spec:

```
public static char[] getUnivId(String username) throws NoSuchUserException  
requires: nothing  
effects: returns an array containing the 9-digit university identifier of username,  
or throws NoSuchUserException if nobody with username is in university  
database. Caller may never modify the returned array.
```

## Sharing a mutable object complicates a contract (2)

---

- That is a bad way to do it
- The problem with this approach is that it means the contract has to be in force for the entire rest of the program — it's a lifetime contract!
- The other contracts we wrote were much narrower in scope; you could think about the precondition just before the call was made, and the postcondition just after, and you didn't have to reason about what would happen for the rest of time
- Here's a spec with a similar problem:

```
public static char[] getUnivId(String username) throws NoSuchUserException
  requires: nothing
  effects: returns a new array containing the 9-digit university identifier of username,
           or throws NoSuchUserException if nobody with username is in university
           database
```

## Sharing a mutable object complicates a contract (3)

---

- That doesn't entirely fix the problem either
- That spec at least says that the array has to be fresh
- But does it keep the implementer from holding an alias to that new array?  
Does it keep the implementer from changing that array or reusing it in the future for something else?

## Sharing a mutable object complicates a contract (4)

- Here's a much better spec:

```
public static String getUnivId(String username) throws NoSuchUserException  
    requires: nothing  
    effects: returns the 9-digit university identifier of username, or throws  
            NoSuchUserException if nobody with username is in university database
```

- The immutable String return value provides a guarantee that the client and the implementer will never step on each other the way they could with char arrays
- It doesn't depend on a programmer reading the spec comment carefully
  - String is immutable
  - Not only that, but this approach (unlike the previous one) gives the implementer the freedom to introduce a cache — a performance improvement

# Useful Immutable Types (1)

---

Since immutable types avoid so many pitfalls, let us enumerate some commonly-used immutable types in the Java API:

- 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 Dates but instead use the appropriate immutable type from **java.time** based on the granularity of timekeeping you need

## Useful Immutable Types (2)

---

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`
  - You can think of the unmodifiable view as *a wrapper* around the underlying list/set/map
  - A client who has a reference to the wrapper and tries to perform mutations — add, remove, put, etc. — will trigger an `UnsupportedOperationException`

## Useful Immutable Types (3)

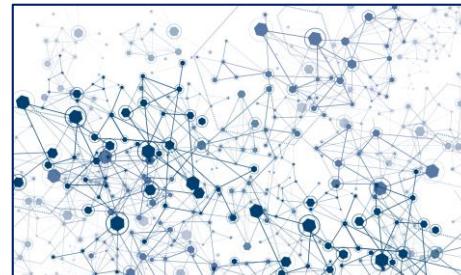
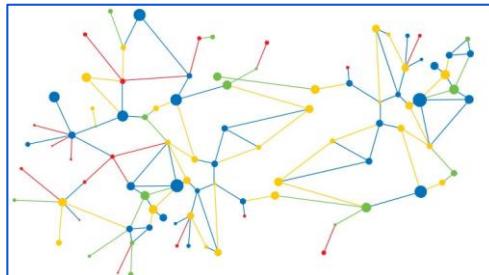
---

- Before we pass a mutable collection to another part of our program, we can wrap it in an unmodifiable wrapper
- We should be careful at that point to *forget our reference to the mutable collection*, lest we accidentally mutate it (one way to do that is to let it go out of scope)
- Just as a mutable object behind a final reference can be mutated, the mutable collection inside an unmodifiable wrapper **can still be modified** by someone with a reference to it, defeating the wrapper
- Collections also provides methods for obtaining immutable empty collections:  
`Collections.emptyList`, etc.
  - Nothing's worse than discovering your definitely very empty list is suddenly definitely not empty!

## Part 3: Disjoint Sets and Dynamic Connectivity

---

- For the third part of the lecture, let us take a break from the *boring* example of List, and explore this new data structure
- We will build a data structure called the **Disjoint Sets**, also known as **Union Find**
  - to solve the **Dynamic Connectivity** problem
- Furthermore, we will see:
  - How a data structure design can evolve from *basic* to *sophisticated*
  - How our choice of *underlying abstraction* can affect the running time



# Disjoint Sets

---

- Disjoint Sets data structure has two operations:
    - **connect(x, y)** : Connects x and y
    - **isConnected(x, y)** : Returns **true** if x and y are connected
- Note that connections can be transitive, i.e. they *don't need* to be direct

# Disjoint Sets

---

- Disjoint Sets data structure has two operations:
    - **connect(x, y)** : Connects x and y
    - **isConnected(x, y)** : Returns **true** if x and y are connected
- Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- connect(China, Vietnam)



# Disjoint Sets

---

- Disjoint Sets data structure has two operations:
    - **connect(x, y)** : Connects x and y
    - **isConnected(x, y)** : Returns **true** if x and y are connected
- Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- `connect(China, Vietnam)`
- `connect(China, Mongolia)`



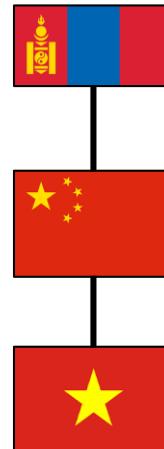
# Disjoint Sets

---

- Disjoint Sets data structure has two operations:
    - **connect(x, y)** : Connects x and y
    - **isConnected(x, y)** : Returns **true** if x and y are connected
- Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- `connect(China, Vietnam)`
- `connect(China, Mongolia)`
- `isConnected(Vietnam, Mongolia) → ?`



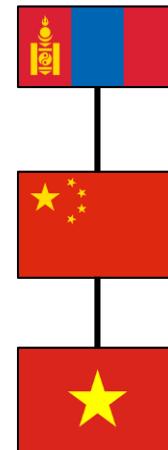
# Disjoint Sets

---

- Disjoint Sets data structure has two operations:
    - **connect(x, y)** : Connects x and y
    - **isConnected(x, y)** : Returns **true** if x and y are connected
- Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- `connect(China, Vietnam)`
- `connect(China, Mongolia)`
- `isConnected(Vietnam, Mongolia) → true`
- `connect(USA, Canada)`



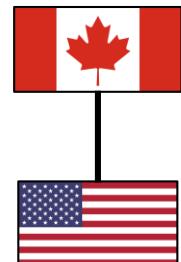
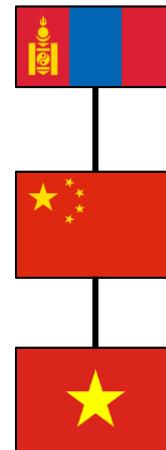
# Disjoint Sets

---

- Disjoint Sets data structure has two operations:
    - **connect(x, y)** : Connects x and y
    - **isConnected(x, y)** : Returns **true** if x and y are connected
- Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- `connect(China, Vietnam)`
- `connect(China, Mongolia)`
- `isConnected(Vietnam, Mongolia) → true`
- `connect(USA, Canada)`
- `isConnected(Canada, Mongolia) → ?`



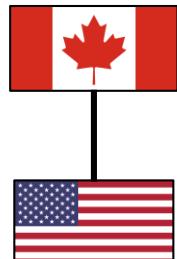
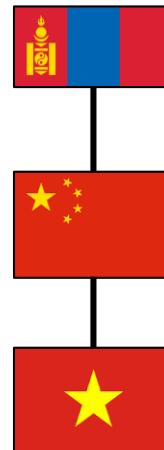
# Disjoint Sets

---

- Disjoint Sets data structure has two operations:
    - **connect(x, y)** : Connects x and y
    - **isConnected(x, y)** : Returns **true** if x and y are connected
- Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- `connect(China, Vietnam)`
- `connect(China, Mongolia)`
- `isConnected(Vietnam, Mongolia) → true`
- `connect(USA, Canada)`
- `isConnected(Canada, Mongolia) → false`
- `connect(China, USA)`



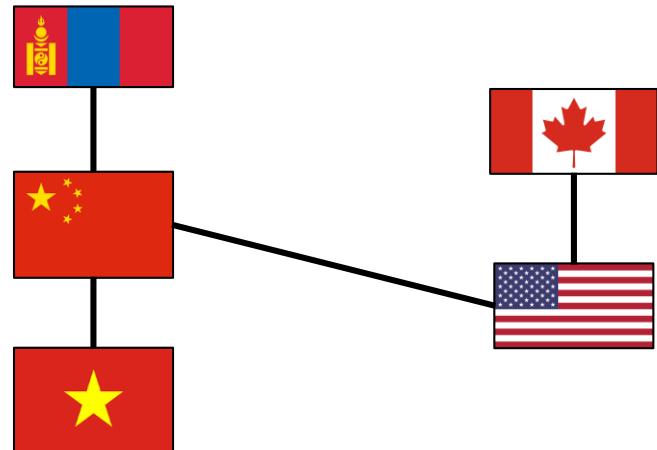
# Disjoint Sets

---

- Disjoint Sets data structure has two operations:
    - **connect(x, y)** : Connects x and y
    - **isConnected(x, y)** : Returns **true** if x and y are connected
- Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- `connect(China, Vietnam)`
- `connect(China, Mongolia)`
- `isConnected(Vietnam, Mongolia) → true`
- `connect(USA, Canada)`
- `isConnected(Canada, Mongolia) → false`
- `connect(China, USA)`
- `isConnected(Canada, Mongolia) → ?`



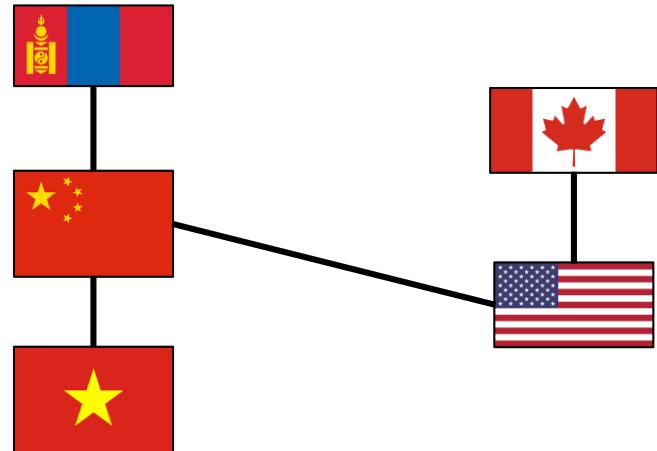
# Disjoint Sets

- Disjoint Sets data structure has two operations:
    - **connect(x, y)** : Connects x and y
    - **isConnected(x, y)** : Returns **true** if x and y are connected
- Note that connections can be transitive, i.e. they *don't need* to be direct

Example:

- `connect(China, Vietnam)`
- `connect(China, Mongolia)`
- `isConnected(Vietnam, Mongolia) → true`
- `connect(USA, Canada)`
- `isConnected(Canada, Mongolia) → false`
- `connect(China, USA)`
- `isConnected(Canada, Mongolia) → true`

the answer can change!



# Disjoint Sets

---

- Disjoint Sets data structure has two operations:
  - **connect(x, y)** : Connects x and y
  - **isConnected(x, y)** : Returns **true** if x and y are connected

Note that connections can be transitive, i.e. they *don't need* to be direct
- This data structure is useful for, for example:
  - Percolation theory in Computational Chemistry
  - Kruskal Algorithm in computing MST
  - and many more

# Disjoint Sets on Integers

---

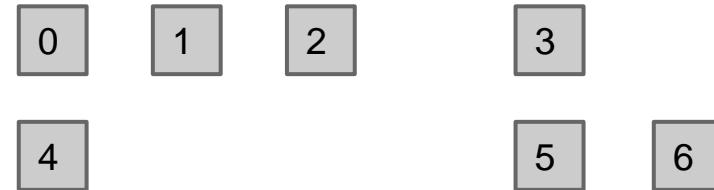
- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is *disconnected*

# Disjoint Sets on Integers

---

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is *disconnected*

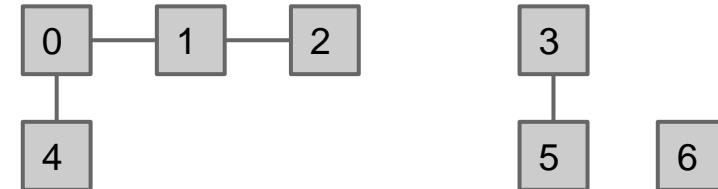
```
ds = DisjointSets(7)
```



# Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4) → true
ds.isConnected(3, 0) → false
ds.connect(4, 2)
```



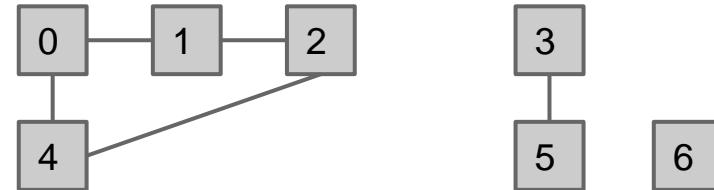
these are called  
**connected components**  
there are 3 connected  
components currently

# Disjoint Sets on Integers

---

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4) → true
ds.isConnected(3, 0) → false
ds.connect(4, 2)
ds.connect(4, 6)
```

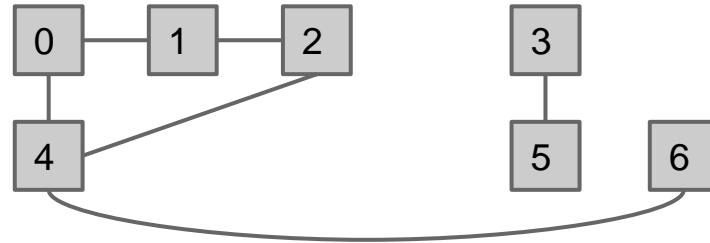


# Disjoint Sets on Integers

---

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4) → true
ds.isConnected(3, 0) → false
ds.connect(4, 2)
ds.connect(4, 6)
ds.connect(3, 6)
```

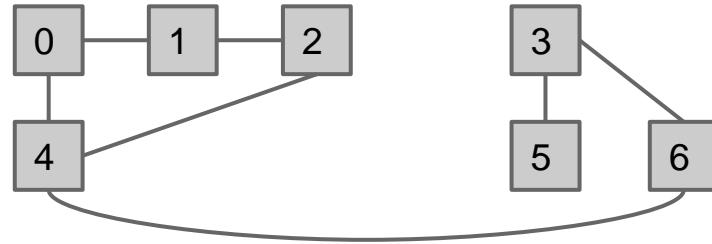


# Disjoint Sets on Integers

---

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4) → true
ds.isConnected(3, 0) → false
ds.connect(4, 2)
ds.connect(4, 6)
ds.connect(3, 6)
ds.isConnected(3, 0) → ?
```

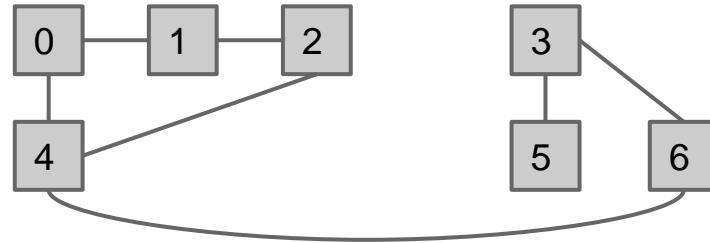


# Disjoint Sets on Integers

---

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
ds.isConnected(2, 4) → true
ds.isConnected(3, 0) → false
ds.connect(4, 2)
ds.connect(4, 6)
ds.connect(3, 6)
ds.isConnected(3, 0) → true
```



# Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected
- We want to implement two methods: (and a constructor)

```
/** Connects two items p and q. */
void connect(int p, int q) {
    ...
}

/** Decides if two items p and q are connected. */
boolean isConnected(int p, int q) {
    ...
}
```

also called **union**

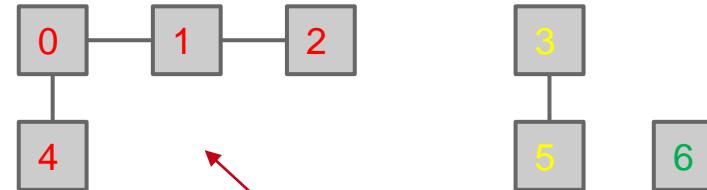
also called **find**

- where the **number of items N** and the **number of operations M** can be *very large*
- and the two methods are called in *arbitrary* order

# Disjoint Sets on Integers

- Let's start simple, we let:
  - the items are integers instead of arbitrary data, and we
  - declare the number of items in advance, where everything is disconnected

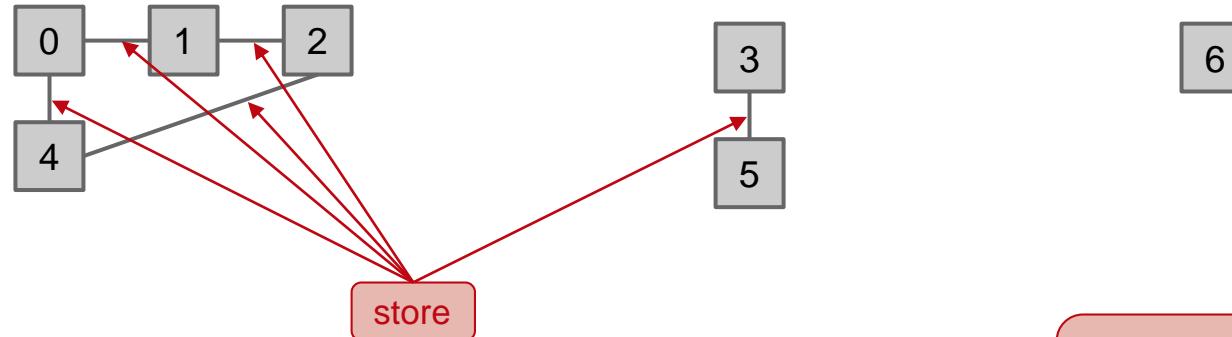
```
ds = DisjointSets(7)
ds.connect(0, 1)
ds.connect(1, 2)
ds.connect(0, 4)
ds.connect(3, 5)
```



the question is: how do we store these connected components?

# Naive Approach

- Let us start to think how we store the connectivity information
- A naive approach:
  - Connect: Record every single connecting line in some data structure
  - isConnected: Iterate over the lines to see if one item can be reached from the other



since connectivity is transitive,  
this idea is redundant  
what should we store instead?

# Connected Components / Sets

---

- A better idea would be to record **the sets** that each item belongs to
  - each set represents a connected component

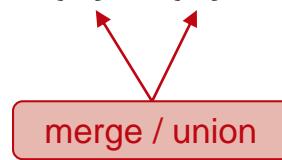
```
ds = DisjointSets(7) {0}, {1}, {2}, {3}, {4}, {5}, {6}
```

# Connected Components / Sets

---

- A better idea would be to record **the sets** that each item belongs to
  - each set represents a connected component

```
ds = DisjointSets(7)      {0}, {1}, {2}, {3}, {4}, {5}, {6}  
ds.connect(0, 1)
```



# Connected Components / Sets

---

- A better idea would be to record **the sets** that each item belongs to
  - each set represents a connected component

```
ds = DisjointSets(7)      {0}, {1}, {2}, {3}, {4}, {5}, {6}  
ds.connect(0, 1)          {0, 1}, {2}, {3}, {4}, {5}, {6}  
ds.connect(1, 2)
```

merge / union

# Connected Components / Sets

---

- A better idea would be to record **the sets** that each item belongs to
  - each set represents a connected component

```
ds = DisjointSets(7)      {0}, {1}, {2}, {3}, {4}, {5}, {6}
ds.connect(0, 1)          {0, 1}, {2}, {3}, {4}, {5}, {6}
ds.connect(1, 2)          {0, 1, 2}, {3}, {4}, {5}, {6}
ds.connect(0, 4)          {0, 1, 2, 4}, {3}, {5}, {6}
ds.connect(3, 5)          {0, 1, 2, 4}, {3, 5}, {6}
ds.isConnected(2, 4)      ←
ds.isConnected(3, 0)      ←
ds.connect(4, 2)
```

no change

# Connected Components / Sets

---

- A better idea would be to record **the sets** that each item belongs to
  - each set represents a connected component

```
ds = DisjointSets(7)      {0}, {1}, {2}, {3}, {4}, {5}, {6}
ds.connect(0, 1)          {0, 1}, {2}, {3}, {4}, {5}, {6}
ds.connect(1, 2)          {0, 1, 2}, {3}, {4}, {5}, {6}
ds.connect(0, 4)          {0, 1, 2, 4}, {3}, {5}, {6}
ds.connect(3, 5)          {0, 1, 2, 4}, {3, 5}, {6}
ds.isConnected(2, 4)
ds.isConnected(3, 0)
ds.connect(4, 2)          {0, 1, 2, 4}, {3, 5}, {6}
ds.connect(4, 6)          {0, 1, 2, 4, 6}, {3, 5}
ds.connect(3, 6)
```

merge / union

# Connected Components / Sets

---

- A better idea would be to record the sets that each item belongs to
  - each set represents a connected component
  - if two items are **in the same set**, they are connected

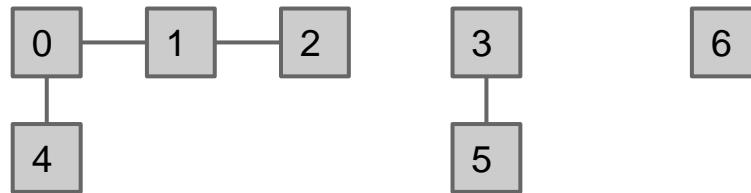
```
ds = DisjointSets(7)      {0}, {1}, {2}, {3}, {4}, {5}, {6}
ds.connect(0, 1)          {0, 1}, {2}, {3}, {4}, {5}, {6}
ds.connect(1, 2)          {0, 1, 2}, {3}, {4}, {5}, {6}
ds.connect(0, 4)          {0, 1, 2, 4}, {3}, {5}, {6}
ds.connect(3, 5)          {0, 1, 2, 4}, {3, 5}, {6}
ds.isConnected(2, 4)
ds.isConnected(3, 0)
ds.connect(4, 2)          {0, 1, 2, 4}, {3, 5}, {6}
ds.connect(4, 6)          {0, 1, 2, 4, 6}, {3, 5}
ds.connect(3, 6)          {0, 1, 2, 3, 4, 5, 6}
ds.isConnected(3, 0)
```

asking: are 3 and 6 in the same set ?

# Selecting Underlying Data Structure

- So, what data structure should we use to track the set membership?
  - in other words, how do we keep track which item in which set?

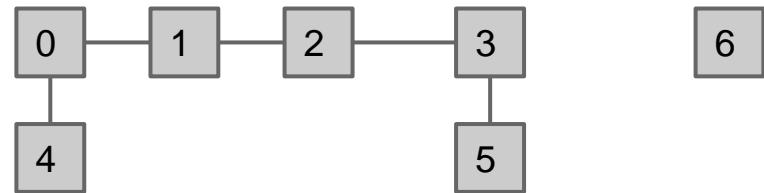
Before connect(2, 3) operation:



{0, 1, 2, 4}, {3, 5}, {6}

5 in here before

After connect(2, 3) operation:



{0, 1, 2, 4, 3, 5}, {6}

5 in here after

# Idea 1: List of Sets

---

- Using `List<Set<Integer>>` from Java Library
  - A very intuitive idea
  - However, imagine in the beginning, we have this List of Sets of integers:  
`[{0}, {1}, {2}, {3}, {4}, {5}, {6}]`
  - We have to iterate through all the sets to find an item
    - complicated and slow!
  - Worst case: if nothing is connected, then `isConnected(5, 6)` requires iterating through  $N-1$  sets to find 5, then  $N$  sets to find 6
    - overall runtime is linear or  $O(N)$
    - similarly for connect

# Performance Summary

---

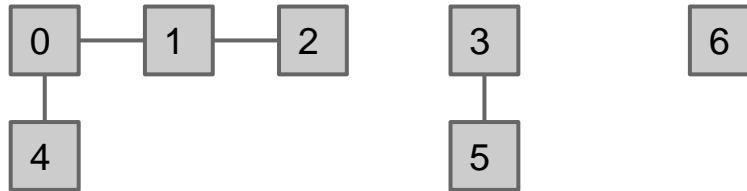
- Let's keep track our progress so far

| Implementation | constructor | connect | isConnected |
|----------------|-------------|---------|-------------|
| ListOfSetsDS   | $O(N)$      | $O(N)$  | $O(N)$      |

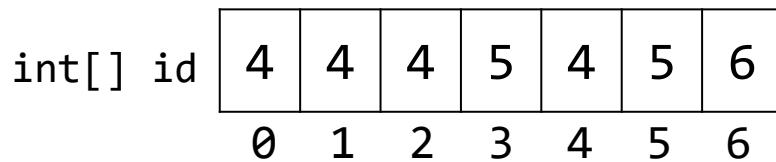
The diagram illustrates the components of the ListOfSetsDS implementation. Three red rounded rectangles contain the following text: 'List of Sets Disjoint Sets' (underlined), 'Initialize N items', and 'need to find items'. Red arrows point from each of these boxes to the corresponding table cells: the first arrow points to the 'Implementation' cell, the second to the 'constructor' cell, and the third to the 'isConnected' cell.

## Idea 2: Array of Integers (1)

- Using an array of integers, where the  $i$ th entry gives *the set number/id* of item  $i$



{0, 1, 2, 4}, {3, 5}, {6}

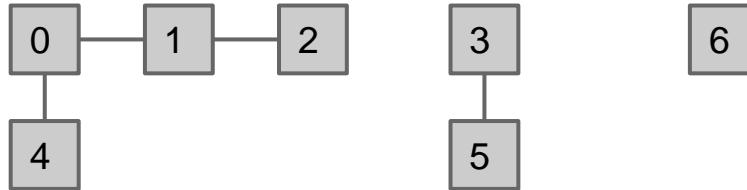


item 0, 1, 2, and 4 belong to set number 4

the set number 4 does not matter, could be any int

## Idea 2: Array of Integers (2)

- Using an array of integers, where the  $i$ th entry gives the set number/id of item  $i$ 
  - `isConnected(p, q)` simply checks whether  $\text{id}[p]$  and  $\text{id}[q]$  are *the same*



{0, 1, 2, 4}, {3, 5}, {6}

int[] id

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 4 | 4 | 4 | 5 | 4 | 5 | 6 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

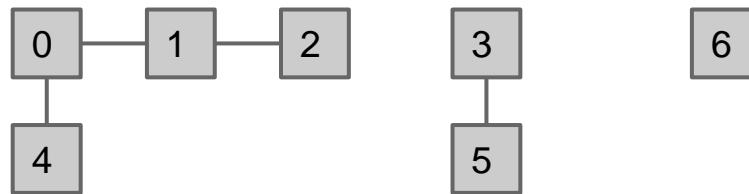
items 2 and 4 are connected

items 5 and 6 are not connected

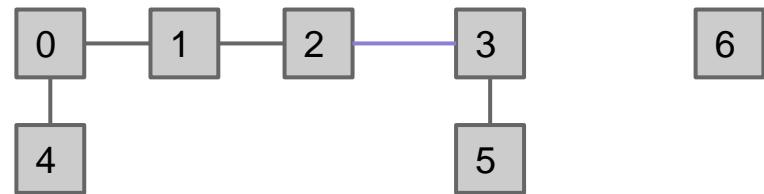
## Idea 2: Array of Integers (3)

- Using an array of integers, where the  $i$ th entry gives the set number/id of item  $i$ 
  - `connect(p, q)` simply changes **all** entries that equal  $\text{id}[p]$  to  $\text{id}[q]$

Before `connect(2, 3)` operation:



After `connect(2, 3)` operation:



{0, 1, 2, 4}, {3, 5}, {6}

| int[] id | 4 | 4 | 4 | 5 | 4 | 5 | 6 |
|----------|---|---|---|---|---|---|---|
|          | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

{0, 1, 2, 4, 3, 5}, {6}

| int[] id | 5 | 5 | 5 | 5 | 5 | 5 | 6 |
|----------|---|---|---|---|---|---|---|
|          | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

the set number of 0, 1, 2, 4 changes from 4 to 5, which is the set number of 3

# Idea 2: Array of Integers (4)

- This implementation of Disjoint Sets data structure is called **QuickFindDS**

```
public class QuickFindDS {  
  
    private int[] id;  
  
    /** Construct a new disjoint sets data structure of N items. */  
    public QuickFindDS(int N){  
        id = new int[N];  
        for (int i = 0; i < N; i++){  
            id[i] = i;  
        }  
    }  
  
    /** Connects two items p and q. */  
    public void connect(int p, int q){  
        int pid = id[p];  
        int qid = id[q];  
        for (int i = 0; i < id.length; i++){  
            if (id[i] == pid){  
                id[i] = qid;  
            }  
        }  
    }  
  
    /** Decides if two items p and q are connected. */  
    public boolean isConnected(int p, int q){  
        return (id[p] == id[q]);  
    }  
}
```

because the **isConnected/find** operation  
is **very fast**, just two array accesses

initially, each item belong to each set

however, as seen in the for loop here,  
**connect/union** is **slow**, taking O(N) time

# Performance Summary

---

- Our second attempt made some progress with `isConnected`,
  - but connecting two items taking linear time each time is unacceptable!

| Implementation            | constructor | connect | <code>isConnected</code> |
|---------------------------|-------------|---------|--------------------------|
| <code>ListOfSetsDS</code> | $O(N)$      | $O(N)$  | $O(N)$                   |
| <code>QuickFindDS</code>  | $O(N)$      | $O(N)$  | $O(1)$                   |

## Next Idea

---

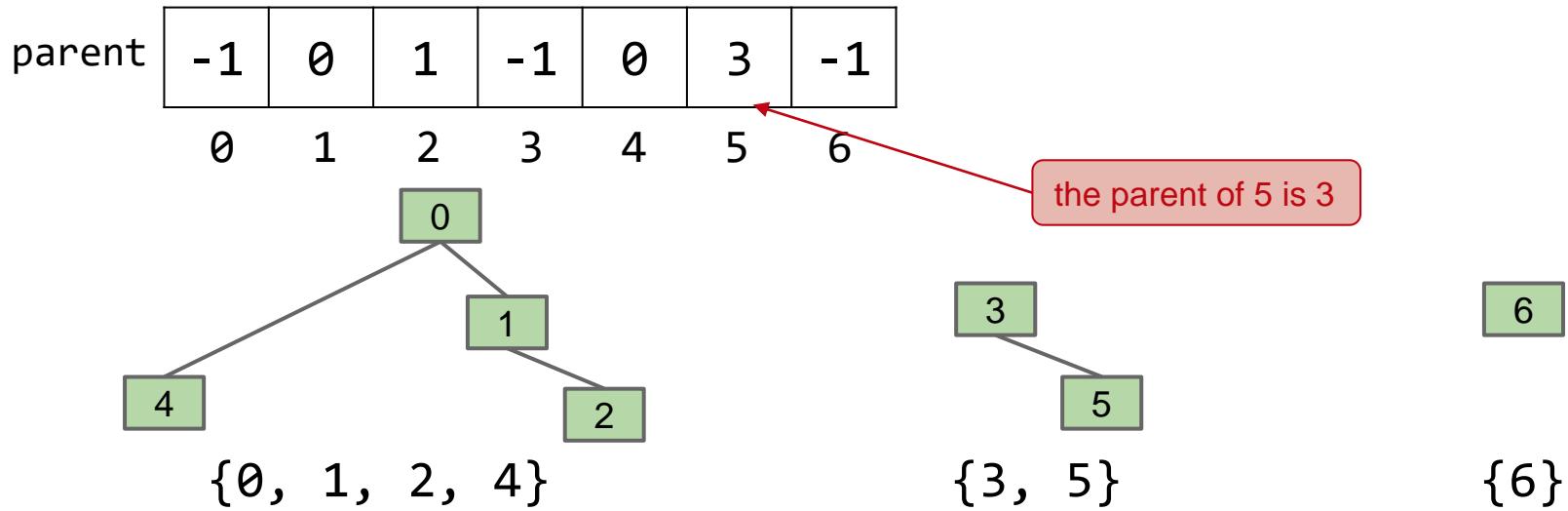
- Because we may need to change *many values*, connect of QuickFindDS is slow
  - we still want to represent the connected components as an array of integers
  - however, values will be chosen so that connect is fast
- How could we change our connected components representation so that combining two sets requires changing only **one value?**

can you solve this? any ideas?



## Idea 3: Array of Parents

- How could we change our set representation so that combining two sets requires changing only **one value**?
  - Assign each item a parent (instead of an id), and set the roots to have value -1
  - Resulting in a forest (a set of trees)

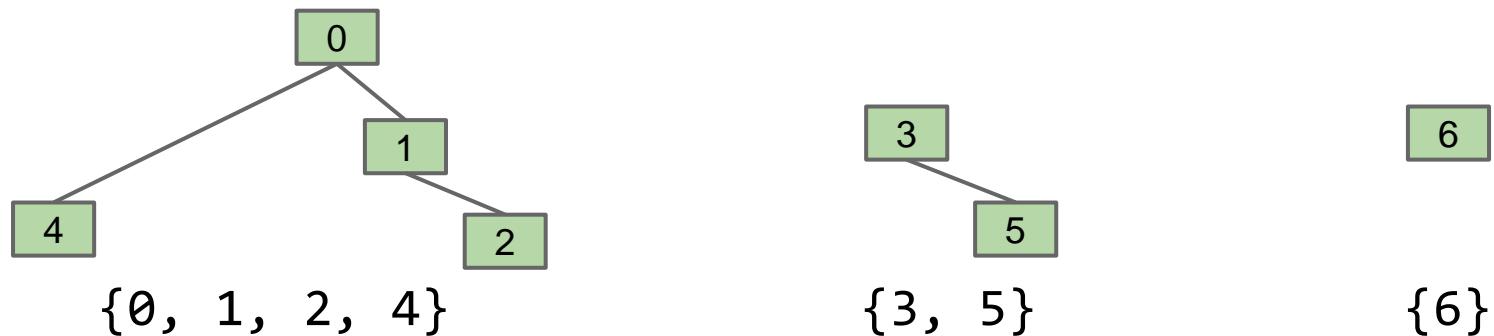


# Improving Connect (1)

- How should we change the parent array if we call **connect(5, 2)**?
  - Can we just set  $\text{parent}[5]$  to 2?

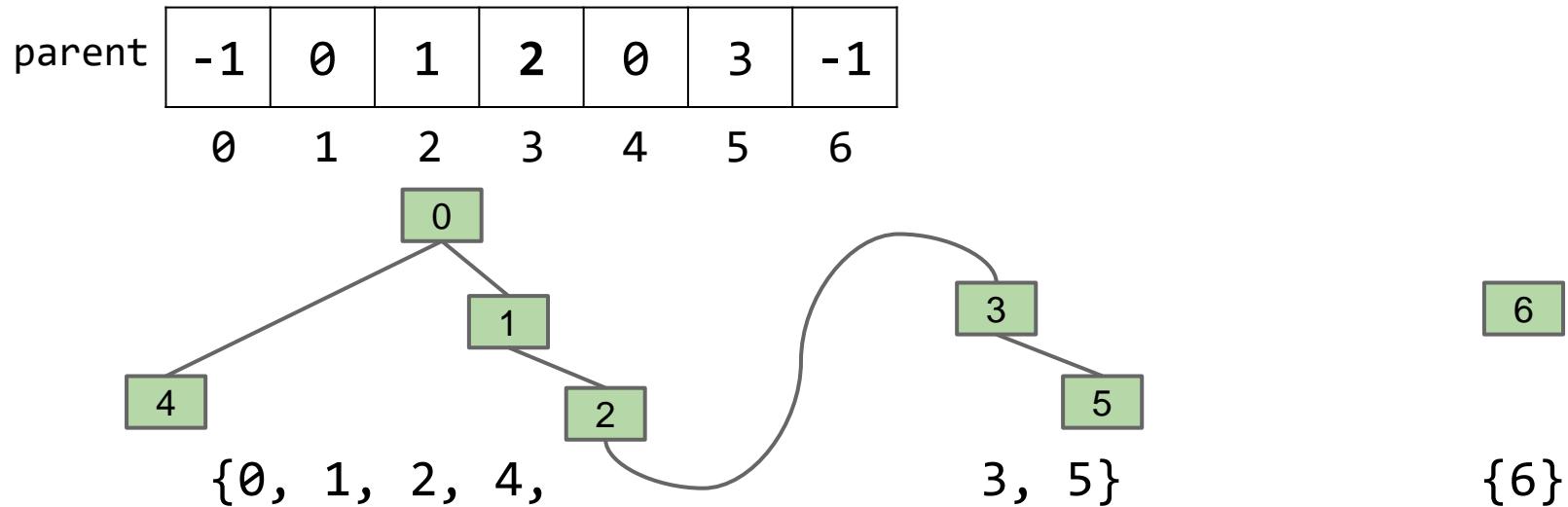
parent

|    |   |   |    |   |   |    |
|----|---|---|----|---|---|----|
| -1 | 0 | 1 | -1 | 0 | 3 | -1 |
| 0  | 1 | 2 | 3  | 4 | 5 | 6  |



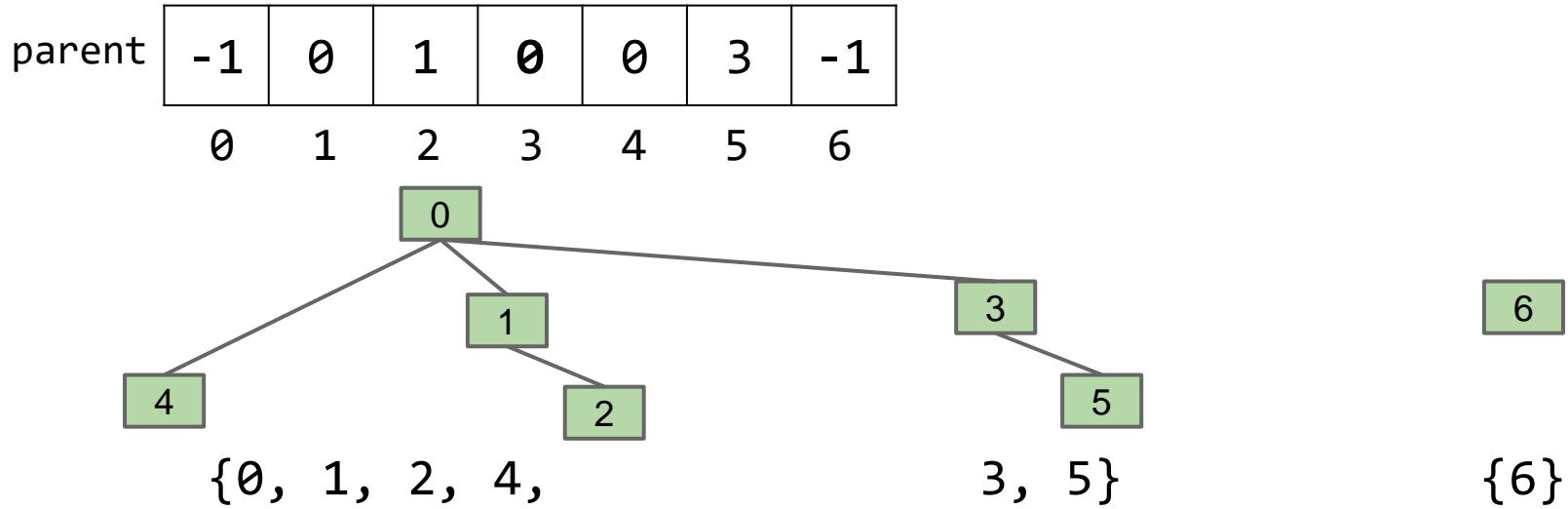
# Improving Connect (2)

- How should we change the parent array if we call **connect(5, 2)**?
  - Can we just set parent[5] to 2? No, 3, the root of 5, will be disconnected
  - One possible idea is to set parent[3] to 2, but this results in taller tree, longer path to root



# Improving Connect (3)

- How should we change the parent array if we call **connect(5, 2)**?
  - Another idea is to do the following:  
find the root 5, find the root of 2,  
and then set the value of the root of 5 to be the root of 2



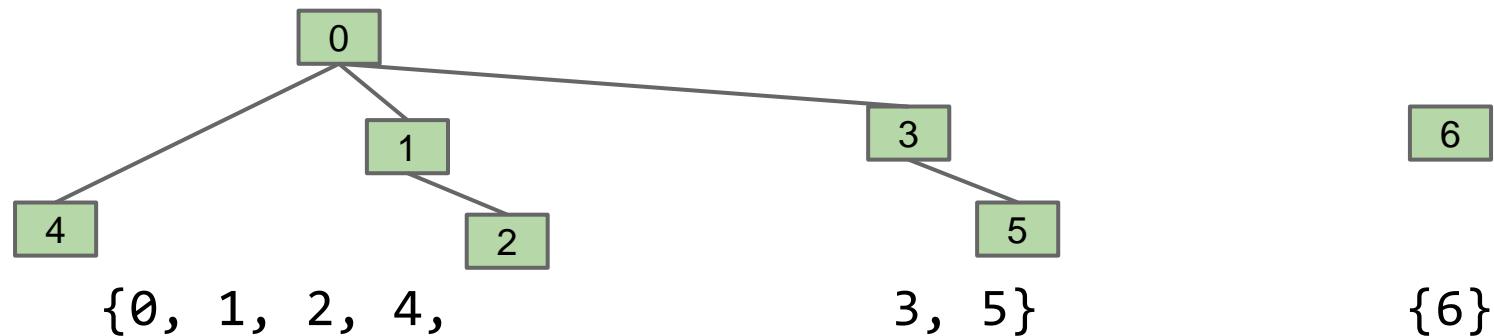
# Improving Connect (3)

- How should we change the parent array if we call **connect(5, 2)**?
  - Another idea is to do the following:  
find the root 5, find the root of 2,  
and then set the value of the root of 5 to be the root of 2

parent

|    |   |   |   |   |   |    |
|----|---|---|---|---|---|----|
| -1 | 0 | 1 | 0 | 0 | 3 | -1 |
| 0  | 1 | 2 | 3 | 4 | 5 | 6  |

seems like a good idea...  
can things go wrong?



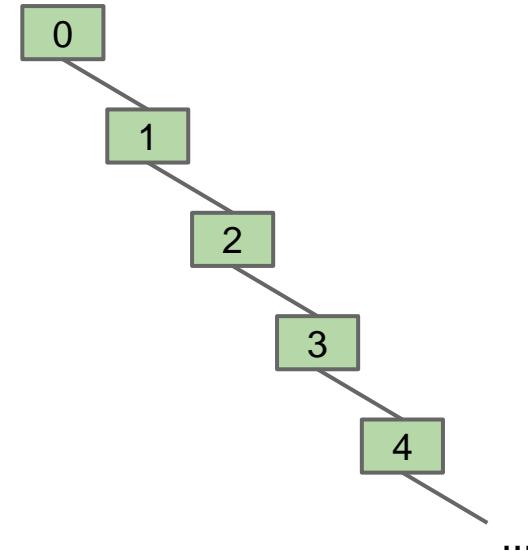
# The Worst-Case

---

- Consider the following series of operations:

- ...
- connect(4, 3)
- connect(3, 2)
- connect(2, 1)
- connect(1, 0)

- For N items, what is the worst case runtime of:
  - connect( $p, q$ )?
  - isConnected( $p, q$ )?



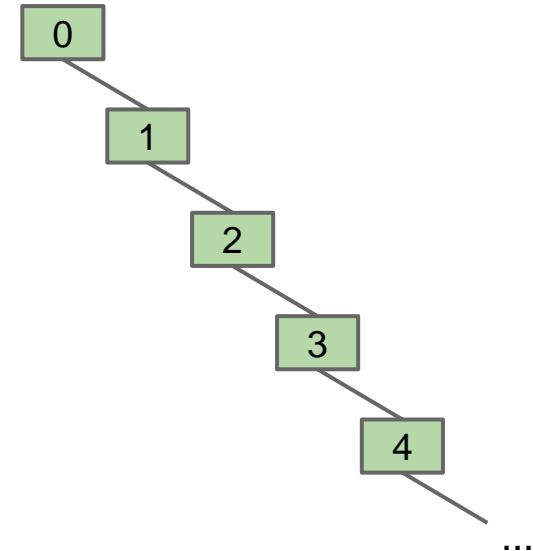
# The Worst-Case

---

- Consider the following series of operations:

- ...
- connect(4, 3)
- connect(3, 2)
- connect(2, 1)
- connect(1, 0)

- For N items, what is the worst case runtime of:
  - connect(p, q)?  $O(N)$
  - isConnected(p, q)?  $O(N)$



# Idea 3: Array of Parents

- This implementation of Disjoint Sets data structure is called **QuickUnionDS**

```
public class QuickUnionDS {  
  
    private int[] parent;  
  
    /** Construct a new disjoint sets data structure of N elements. */  
    public QuickUnionDS(int N) {  
        parent = new int[N];  
        for (int i = 0; i < N; i++) {  
            parent[i] = -1;  
        }  
    }  
  
    // Finding the root.  
    private int find(int p) {  
        while (parent[p] >= 0) {  
            p = parent[p];  
        }  
        return p;  
    }  
  
    /** Connects two elements p and q. */  
    public void connect(int p, int q) {  
        int i = find(p);  
        int j = find(q);  
        parent[i] = j;  
    }  
  
    /** Decides if two elements p and q are connected. */  
    public boolean isConnected(int p, int q) {  
        return find(p) == find(q);  
    }  
}
```

finding the root costs  $O(N)$  in worst-case

causing connect and isConnected to cost  $O(N)$  worst-case running time as well

# Performance Summary

---

- Our third attempt may perform well in average,  
but in the worst case, we actually get a worse performance!

| Implementation | Constructor | connect | isConnected |
|----------------|-------------|---------|-------------|
| ListOfSetsDS   | $O(N)$      | $O(N)$  | $O(N)$      |
| QuickFindDS    | $O(N)$      | $O(N)$  | $O(1)$      |
| QuickUnionDS   | $O(N)$      | $O(N)$  | $O(N)$      |

## Next Idea

---

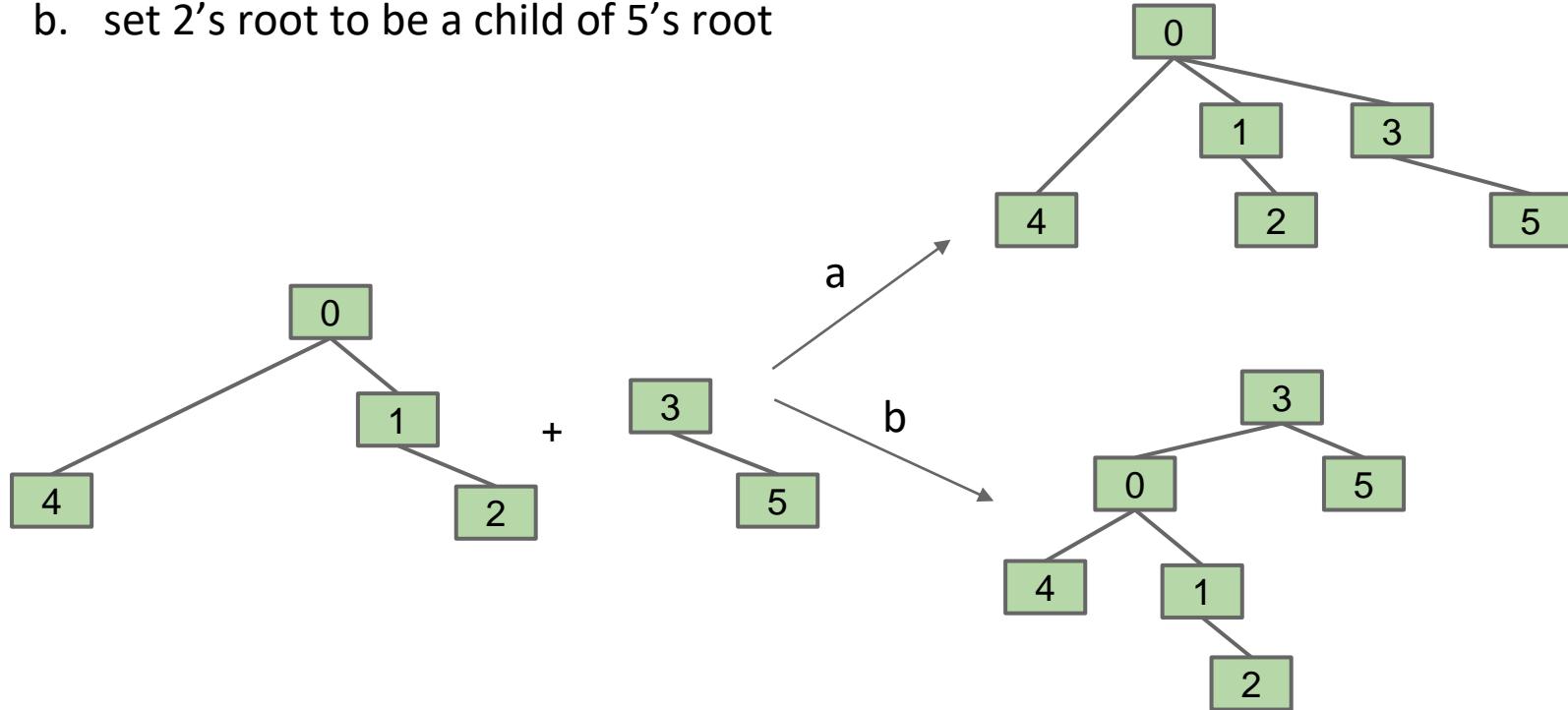
- The defect of QuickUnionDS is that the trees can get tall
  - resulting in potentially even worse performance than QuickFindDS if tree is imbalanced
- What should we do to keep our trees balanced?

can you solve this? any ideas?



# In-Class Quiz 3

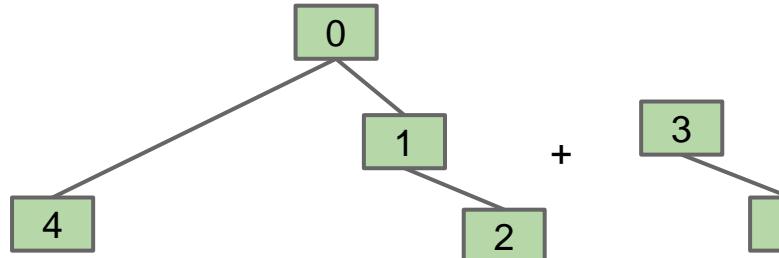
- Suppose you are trying to connect(5, 2) — which one is better :
  - set 5's root to be a child of 2's root
  - set 2's root to be a child of 5's root



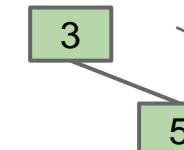
# An Observation

- Suppose you are trying to connect(5, 2) — which one is better :
  - set 5's root to be a child of 2's root
  - set 2's root to be a child of 5's root

which option gives you a less tall tree?

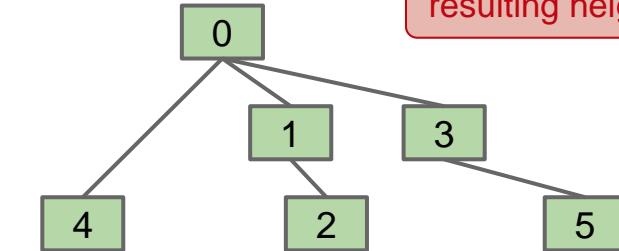


+

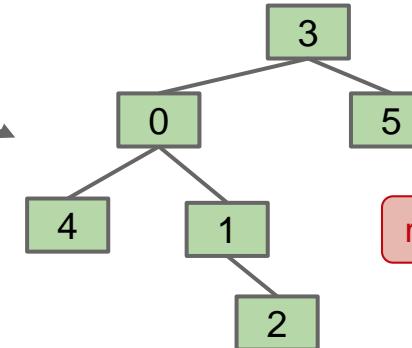


a

b



resulting height is 2



resulting height is 3

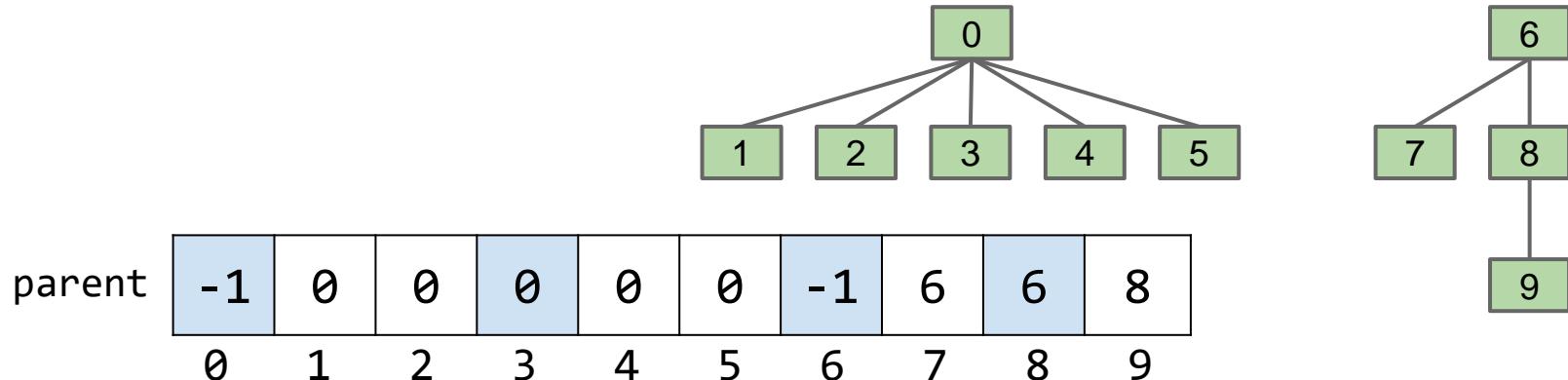
## Idea 4: Link Smaller to Larger

---

- Rather than always links the root of *first* tree to the root of *second* tree during connect as in QuickUnionDS:
  - keep track of the **tree size** (its number of elements)
  - link the the root of *smaller* tree to the root of *larger* tree

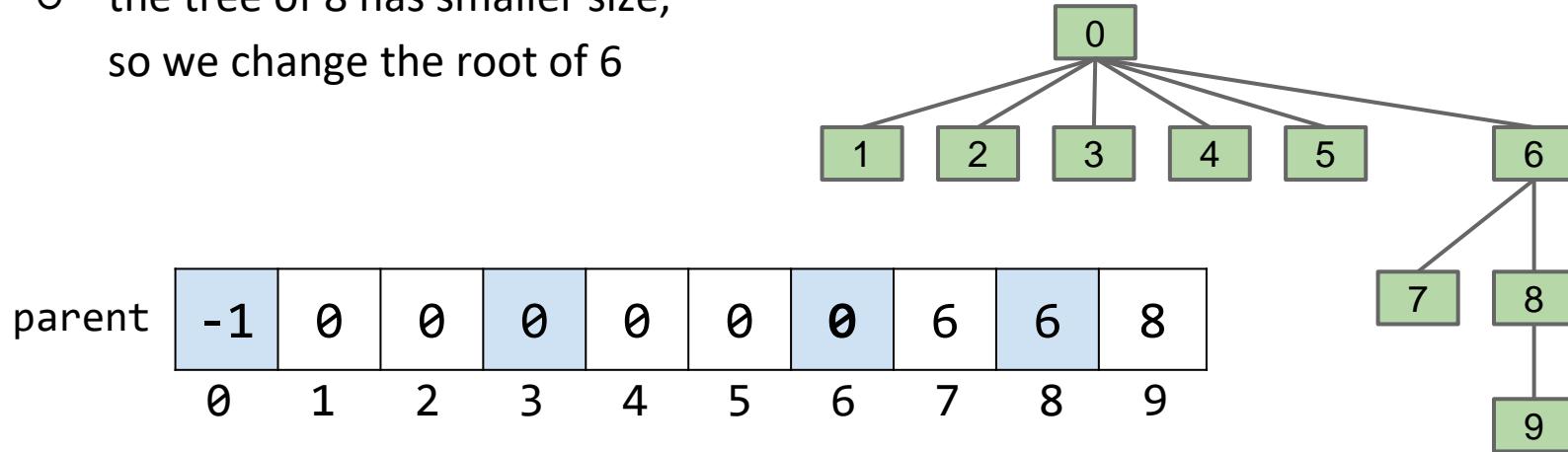
# Example Linking Smaller to Larger (1)

- Rather than always links the root of *first* tree to the root of *second* tree during connect as in QuickUnionDS:
  - keep track of the **tree size** (its number of elements)
  - link the the root of *smaller* tree to the root of *larger* tree
- Example: what will connect(3, 8) change ?



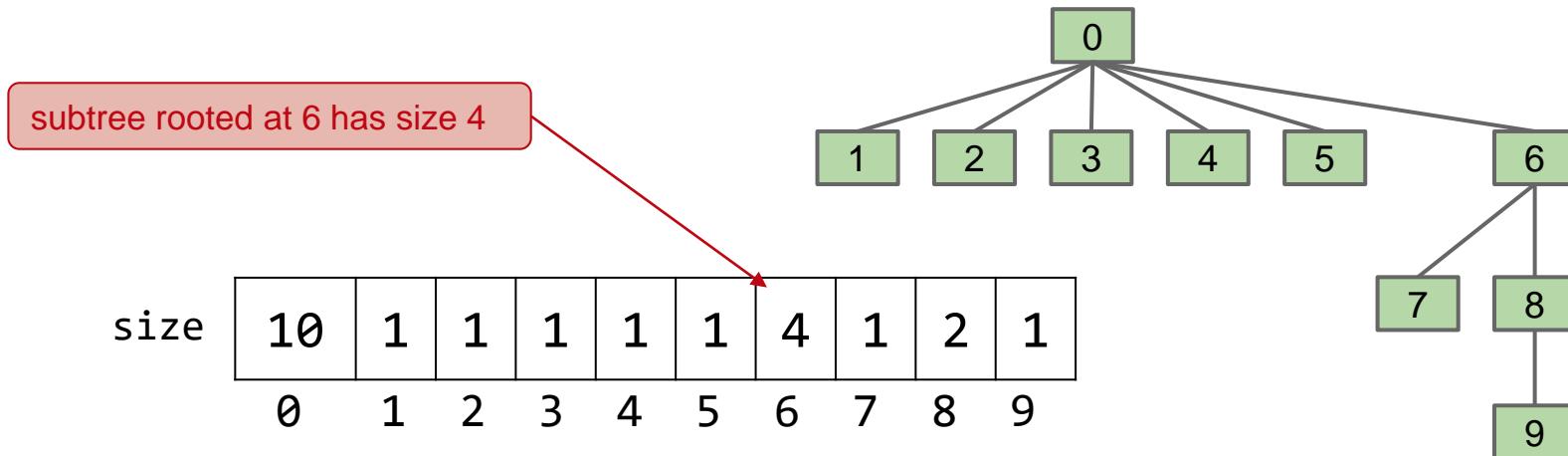
## Example Linking Smaller to Larger (2)

- Rather than always links the root of *first* tree to the root of *second* tree during connect as in QuickUnionDS:
  - keep track of the **tree size** (its number of elements)
  - link the the root of *smaller* tree to the root of *larger* tree
- Example: what will connect(3, 8) change ?
  - the tree of 8 has smaller size,  
so we change the root of 6



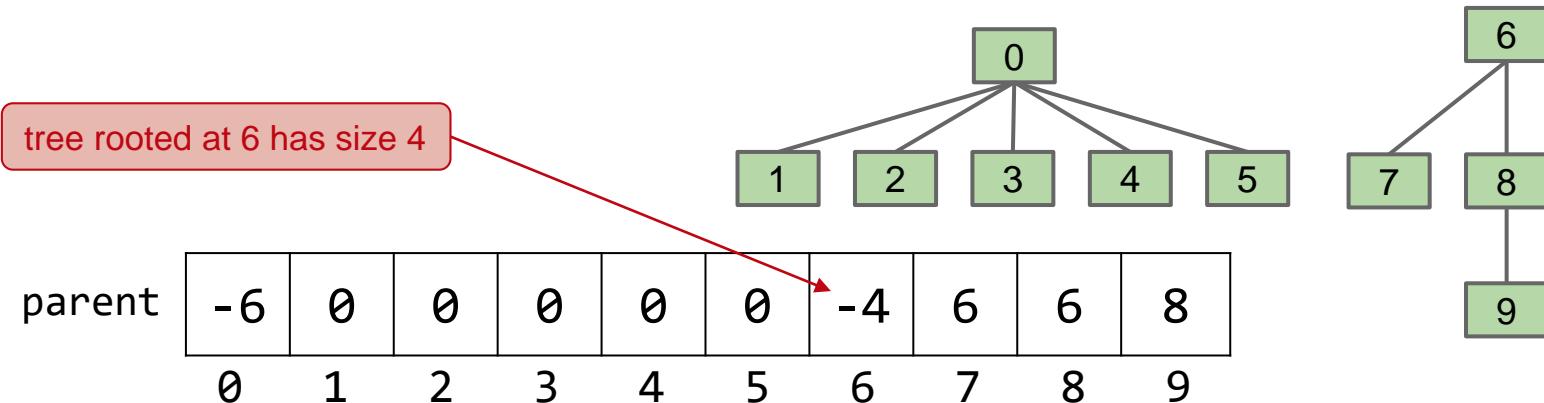
## Idea 4: Weighted Quick Union (1)

- How do we implement this? What is the underlying data structure(s)?
  - we still use array of parents as in QuickUnionDS
  - to keep track the sizes:
    1. we could have another array size[] to store ***the size of the subtree rooted at index i***



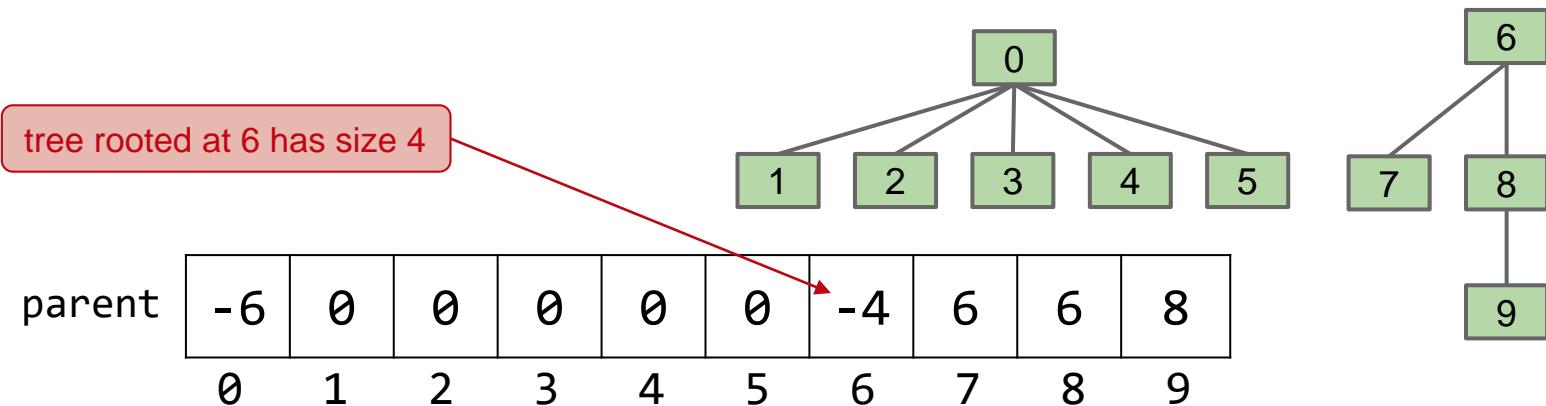
## Idea 4: Weighted Quick Union (2)

- How do we implement this? What is the underlying data structure(s)?
  - we still use array of parents as in QuickUnionDS
  - to keep track the sizes:
    1. we could have another array size[] to store the size of the subtree rooted at index i
    2. instead of -1, we store ***negative size*** of the trees in the root



## Idea 4: Weighted Quick Union (2)

- How do we implement this? What is the underlying data structure(s)?
  - we still use array of parents as in QuickUnionDS
  - to keep track the sizes:
    1. we could have another array size[] to store the size of the subtree rooted at index i
    2. instead of -1, we store ***negative size*** of the trees in the root



let's use Idea 4.2. in our Lab 9 !

# Relationship between Weight and Height (1)

---

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )

| $N$ | $H$ |
|-----|-----|
| 1   | 0   |

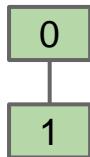
0

when  $N = 1$ , the tallest possible tree has height  $H = 0$

## Relationship between Weight and Height (2)

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )

| $N$ | $H$ |
|-----|-----|
| 1   | 0   |
| 2   | 1   |

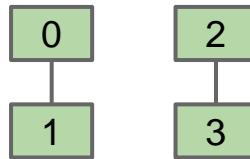


when  $N = 2$ , the tallest possible tree has height  $H = 1$

# Relationship between Weight and Height (3)

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )

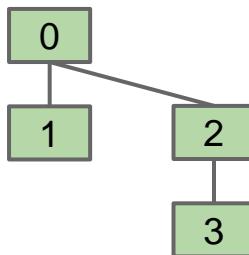
| $N$ | $H$ |
|-----|-----|
| 1   | 0   |
| 2   | 1   |



we will connect/union  
those two trees next

# Relationship between Weight and Height (4)

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )



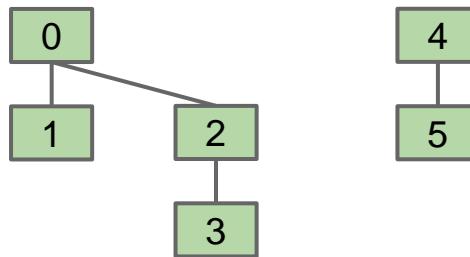
| $N$ | $H$ |
|-----|-----|
| 1   | 0   |
| 2   | 1   |
| 4   | 2   |

when  $N = 4$ , the tallest possible tree has height  $H = 2$

# Relationship between Weight and Height (5)

---

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )

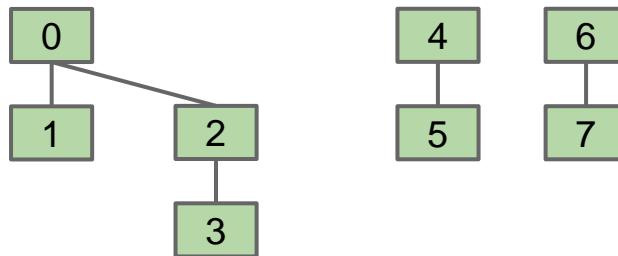


| N | H |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 4 | 2 |

# Relationship between Weight and Height (6)

---

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )

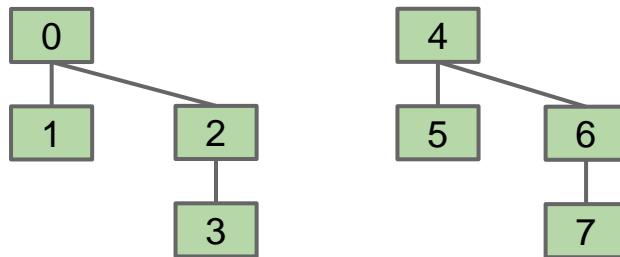


| N | H |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 4 | 2 |

# Relationship between Weight and Height (7)

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )

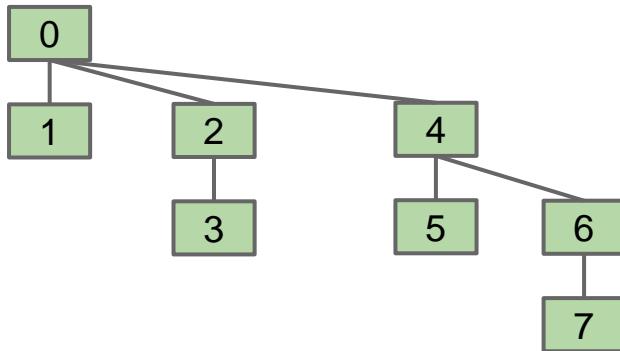
| $N$ | $H$ |
|-----|-----|
| 1   | 0   |
| 2   | 1   |
| 4   | 2   |



connect those two to  
get a height-3 tree

# Relationship between Weight and Height (8)

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )



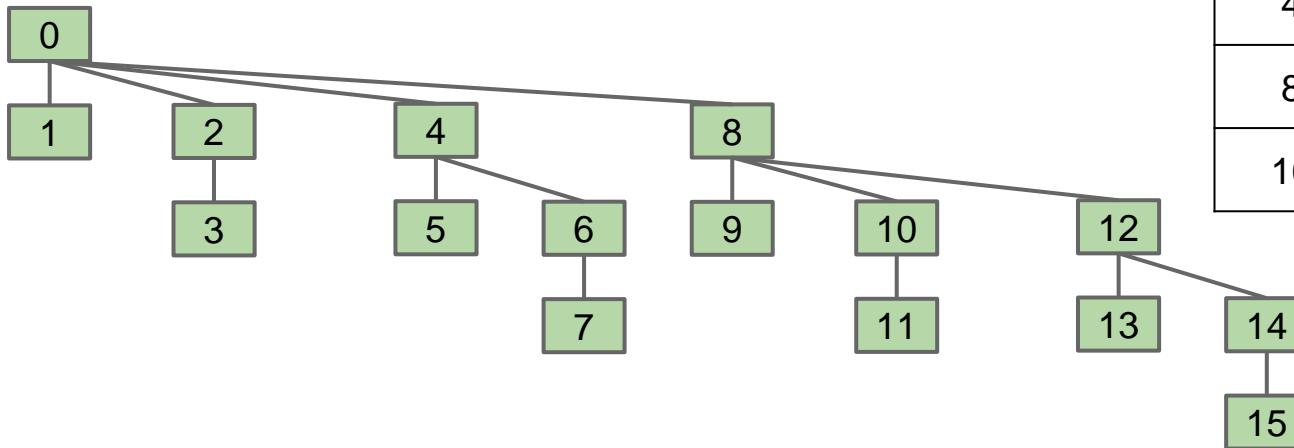
| $N$ | $H$ |
|-----|-----|
| 1   | 0   |
| 2   | 1   |
| 4   | 2   |
| 8   | 3   |

when  $N = 8$ , the tallest possible tree has height  $H = 3$

# Relationship between Weight and Height (9)

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )

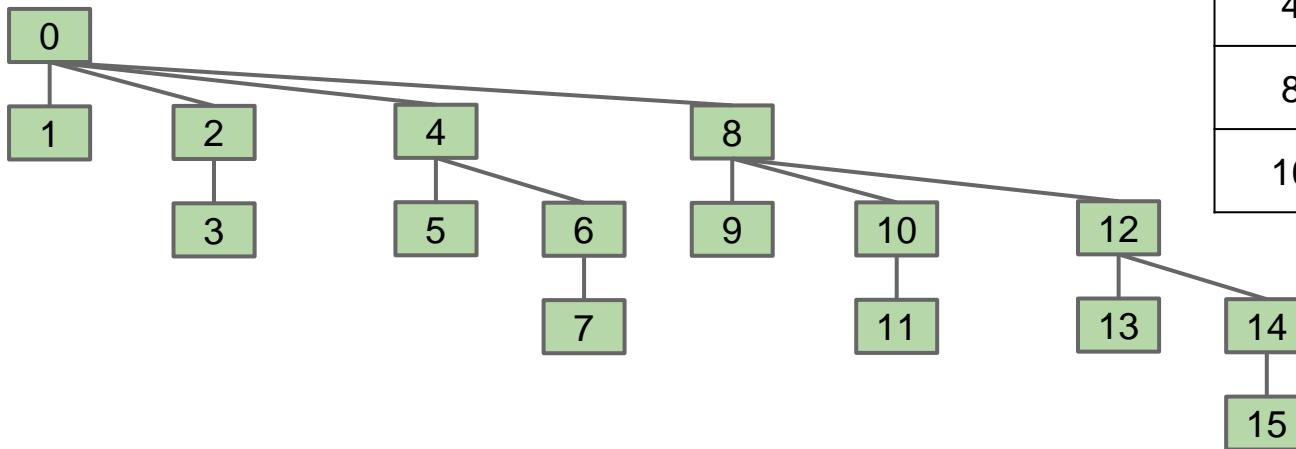
when  $N = 16$ , the tallest possible tree has height  $H = 4$



| $N$ | $H$ |
|-----|-----|
| 1   | 0   |
| 2   | 1   |
| 4   | 2   |
| 8   | 3   |
| 16  | 4   |

# Relationship between Weight and Height (10)

- We now illustrate the relationship between the weight ( $N$ , number of items) and the worst possible height ( $H$ )
  - the worst-case tree height  $H = O(\log N)$
  - since connect and isConnected both depends on find, and find depends on the height, they are also  $O(\log N)$



| N  | H |
|----|---|
| 1  | 0 |
| 2  | 1 |
| 4  | 2 |
| 8  | 3 |
| 16 | 4 |

# Performance Summary

---

- Our fourth attempt has finally achieved a good performance!
  - logarithmic time is fast enough for practical problems

| Implementation       | constructor | connect     | isConnected |
|----------------------|-------------|-------------|-------------|
| ListOfSetsDS         | $O(N)$      | $O(N)$      | $O(N)$      |
| QuickFindDS          | $O(N)$      | $O(N)$      | $O(1)$      |
| QuickUnionDS         | $O(N)$      | $O(N)$      | $O(N)$      |
| WeightedQuickUnionDS | $O(N)$      | $O(\log N)$ | $O(\log N)$ |

# Thank you for your attention !

---

- In this lecture, you have learned:
  - about assertions, incremental development, and scope minimization
  - about encapsulation
  - to identify aliasing and understand the dangers of mutability
  - to use immutability to improve correctness, clarity, and changeability
  - to create an iterator that iterate through your data structures
  - to create an efficient disjoint sets data structure
- Please continue to Lecture Quiz 9 and Lab 9:
  - to do Lab Exercise 9.1 - 9.4, and
  - to do Exercise 9.1 - 9.3



# Advanced Object-Oriented Programming

CPT204 – Lecture 10  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming

### Lecture 10

**ADT, Rep Independence,  
Interface, Inheritance, Iterable**

# Welcome !

---

- Welcome to Lecture 10 !
- In this lecture we are going to
  - learn about Abstract Data Types (ADT) and Representation Independence
  - learn about Interface, Inheritance, and Iterability

# Part 1 : ADT and Rep Independence

---

- The first part of the lecture introduces two ideas:
  - Abstract data types
  - Representation independence
- We look at a powerful idea called **abstract data types**, which enable us to *separate* how we use a data structure in a program from the particular form of the data structure itself
- Abstract data types address a particularly dangerous problem: clients making assumptions about the type's *internal representation*
  - we'll see why this is dangerous and how it can be avoided
  - we'll also discuss the classification of operations and some principles of good design for abstract data types

# What Abstraction Means

---

Abstract data types are an instance of a general principle in software engineering, which goes by many names:

- **Abstraction.** Omitting or hiding low-level details with a simpler, higher-level idea
- **Modularity.** Dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system
- **Encapsulation.** Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity
- **Information hiding.** Hiding details of a module's implementation from the rest of the system, so that those details can be changed later without changing the rest of the system
- **Separation of concerns.** Making a feature (or "concern") the responsibility of a single module, rather than spreading it across multiple modules

# User-Defined Types (1)

---

- In the early days of computing, a programming language came with built-in types (such as integers, booleans, strings, etc.) and built-in procedures, e.g., for input and output
  - users could define their own procedures: that's how large programs were built
- A major advance in software development was the idea of abstract types: *that one could design a programming language to allow user-defined types, too*
- This idea came out of the work of many researchers, notably **Dahl** (the inventor of the Simula language), **Hoare** (who developed many of the techniques we now use to reason about abstract types), **Parnas** (who coined the term information hiding and first articulated the idea of organizing program modules around the secrets they encapsulated), and **Barbara Liskov** and **John Guttag**, who did seminal work in the specification of abstract types, and in programming language support for them
  - **Barbara Liskov** earned the **Turing Award**, computer science's equivalent of the Nobel Prize, for her work on abstract types !!

## User-Defined Types (2)

---

- The key idea of data abstraction is that a type is *characterized by the operations you can perform on it*
  - a number is something you can add and multiply; a string is something you can concatenate and take substrings of; a boolean is something you can negate, and so on
- In a sense, users could already define their own types in early programming languages: you could create a record type date, for example, with integer fields for day, month, and year
  - but what made abstract types new and different was the focus on operations: the user of the type would not need to worry about how its values were actually stored, in the same way that a programmer can ignore how the compiler actually stores integers

## User-Defined Types (3)

---

- In Java, as in many modern programming languages, the separation between built-in types and user-defined types is a bit blurry
  - the classes in `java.lang`, such as `Integer` and `Boolean` are built-in; whether you regard all the collections of `java.util` as built-in is less clear (and not very important anyway)
  - Java complicates the issue by having primitive types that are *not* objects
  - the set of these types, such as `int` and `boolean`, cannot be extended by the user

# Classifying Types and Operations

---

- Types, whether built-in or user-defined, can be classified as mutable or immutable
- The objects of a mutable type can be changed: that is, they provide operations which when executed cause the results of other operations on the same object to give different results
  - so Date is mutable, because you can call `setMonth` and observe the change with the `getMonth` operation
  - but String is immutable, because its operations create new String objects rather than changing existing ones
  - sometimes a type will be provided in two forms, a mutable and an immutable form: `StringBuilder`, for example, is a mutable version of `String` (although the two are certainly not the same Java type, and are not interchangeable)

# Operations of Abstract Type (1)

---

The operations of an abstract type are classified as follows:

- **Creators** create new objects of the type;  
A creator may take an object as an argument, but *not* an object of the type being constructed
- **Producers** create new objects from old objects of the type;  
The concat method of String, for example, is a producer: it takes two strings and produces a new one representing their concatenation
- **Observers** take objects of the abstract type and return objects of a different type;  
The size method of List, for example, returns an int
- **Mutators** change objects;  
The add method of List, for example, mutates a list by adding an element to the end

## Operations of Abstract Type (2)

---

- We can summarize these distinctions schematically like this:
  - creator :  $t^* \rightarrow T$
  - producer :  $T+, t^* \rightarrow T$
  - observer :  $T+, t^* \rightarrow t$
  - mutator :  $T+, t^* \rightarrow \text{void} \mid t \mid T$
- These show informally the shape of the signatures of operations in the various classes
  - each  $T$  is the abstract type itself; each  $t$  is some other type
  - the  $+$  marker indicates that the type may occur *one or more* times in that part of the signature
  - and the  $*$  marker indicates that it occurs *zero or more* times
  - $\mid$  indicates or

## Operations of Abstract Type (3)

---

- For example, a producer may take two values of the abstract type T, like `String.concat()` does:
  - `concat : String × String → String`
- Some observers take zero arguments of other types t, such as:
  - `size : List → int`
- And other observers take several, such as:
  - `regionMatches : String × boolean × int × String × int × int → boolean`

## Operations of Abstract Type (4)

---

- A creator operation is often implemented as a constructor, like `new ArrayList()`
  - But a creator can simply be a static method instead, like `Arrays.asList()`
- A creator *implemented as a static method* is often called **a factory method**
- The various `String.valueOf` methods in Java are other examples of creators implemented as factory methods

## Operations of Abstract Type (5)

---

- Mutators are *often* signaled by a void return type
  - a method that returns void must be called for some kind of *side-effect*, since otherwise it doesn't return anything
- But **not all** mutators return void
  - for example, `Set.add()` returns a boolean that indicates whether the set was actually changed
  - also, in Java's graphical user interface toolkit, `Component.add()` returns the object itself, so that multiple `add()` calls can be chained together

# Abstract Data Type Examples (1)

---

Here are some examples of abstract data types, along with some of their operations, grouped by kind

- `int` is Java's primitive integer type
  - int is immutable, so it has *no* mutators
    - creators: the numeric literals 0, 1, 2, ...
    - producers: arithmetic operators +, -, ×, ÷
    - observers: comparison operators ==, !=, <, >
    - mutators: none (it's immutable)

## Abstract Data Type Examples (2)

---

- `List` is Java's list type

`List` is mutable

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

These classes include `ArrayList` and `LinkedList`

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

## Abstract Data Type Examples (3)

---

- **String** is Java's string type

String is immutable

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

# In-Class Quiz 1

---

- Map.keySet() is a method from the Java library
  - below is the method's Javadoc documentation, look at its signature
- What kind of operation of an abstract data type is it?

**keySet**

```
Set<K> keySet()
```

Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

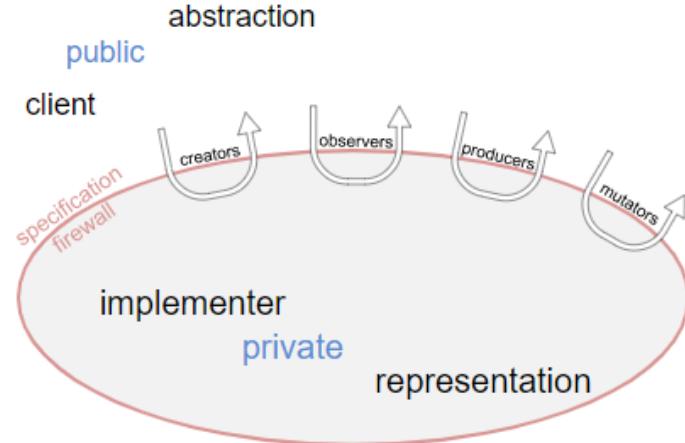
**Returns:**

a set view of the keys contained in this map

- creator
- producer
- observer
- mutator

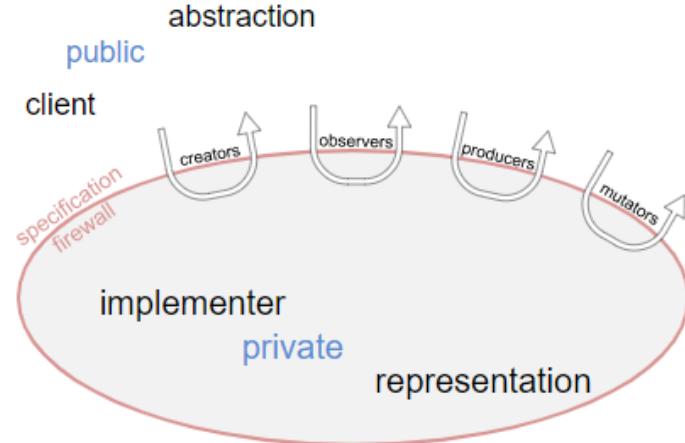
# An Abstract Type is Defined by Its Operations

- The essential idea here is that **an abstract data type is defined by its operations**
  - the set of operations for a type T, along with their specifications, fully characterize what we mean by T
- So, for example, when we talk about the List type, what we mean is **not** a linked list or an array or any other specific data structure for representing a list
- Instead, the List type is a set of opaque values — the possible objects that can have List type — that satisfy the specifications of all the operations of List: get(), size(), etc
  - the values of an abstract type are opaque in the sense that a client can't examine the data stored inside them, except as permitted by operations



# Abstraction vs Representation

- Expanding our metaphor of a specification firewall, you might picture values of an abstract type as *hard shells, hiding* not just the implementation of an individual function, but of a set of related functions (the operations of the type) and the data they share (the private fields stored inside values of the type)
- The operations of the type constitute its **abstraction**
  - this is the **public** part, visible to clients who use the type
- The fields of the class that implements the type, as well as related classes that help implement a complex data structure, constitute a particular **representation**
  - this part is **private**, visible only to the implementer of the type



# Designing an Abstract Type (1)

---

Designing an abstract data type involves choosing good operations and determining how they should behave;

Here are a few rules of thumb:

- It's better to have **a few, simple operations** that can be combined in powerful ways, rather than lots of complex operations
- Each operation should have a well-defined purpose and should have a **coherent** behavior rather than a multitude of special cases
  - we probably *shouldn't* add a sum operation to List, for example
  - it might help clients who work with lists of integers, but what about lists of strings? or nested lists? all these special cases would make sum a hard operation to understand and use

## Designing an Abstract Type (2)

---

- The set of operations should be **adequate** in the sense that there must be enough to do the kinds of computations clients are likely to want to do
  - a good test is to check that *every property* of an object of the type can be *extracted*
  - for example, if there were no get operation, we would not be able to find out what the elements of a list are
  - basic information should not be inordinately difficult to obtain
  - for example, the size method is not strictly necessary for List, because we could apply get on increasing indices until we get a failure, but this is inefficient and inconvenient

# Designing an Abstract Type (3)

---

- The type may be *generic*: a list or a set, or a graph, for example
  - or it may be *domain-specific*: a street map, an employee database, a phone book, etc
- But it **should not mix generic and domain-specific features**
  - A Deck type intended to represent a sequence of playing cards shouldn't have a generic add method that accepts arbitrary objects like integers or strings
  - Conversely, it wouldn't make sense to put a domain-specific method like dealCards into the generic type List
- Next, it should be **representation independent**

# Representation Independence

---

- Critically, a good abstract data type (ADT) should be **representation independent**
- This means that the use of an abstract type is independent of its *representation* : the **actual data structure** or **data fields** or **instance variable** used to implement it, so that changes in representation have no effect on code outside the abstract type itself
  - for example, the *operations offered by List* are *independent* of whether the list is represented as a *linked list* or as an *array*
- You won't be able to change the representation of an ADT at all unless its operations are fully specified with preconditions and postconditions, so that clients know what to depend on, and you know what you can safely change

# Example: Different Representations for Strings (1)

---

- Let's look at a simple abstract data type to see what representation independence means and why it's useful
- The **MyString type** below has far fewer operations than the real Java String, and their specs are a little different, but it's still illustrative
- Here are the specs for the ADT:

```
/** MyString represents an immutable sequence of characters. */
public class MyString {

    ///////////////////// Example of a creator operation ///////////////////
    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) { ... }
```

## Example: Different Representations for Strings (2)

---

```
////////// Examples of observer operations //////////
/** @return number of characters in this string */
public int length() { ... }

/** @param i character position (requires 0 <= i < string length)
 *  @return character at position i */
public char charAt(int i) { ... }

////////// Example of a producer operation //////////
/** Get the substring between start (inclusive) and end (exclusive).
 *  @param start starting index
 *  @param end ending index. Requires 0 <= start <= end <= string length.
 *  @return string consisting of charAt(start)...charAt(end-1) */
public MyString substring(int start, int end) { ... }

}
```

no mutator operation. why not?

# Testing MyString (1)

---

- These public operations and their specifications are the only information that a client of this data type is allowed to know
- Following the test-first programming paradigm, in fact, the first client we should create is a test suite that exercises these operations according to their specs
- At the moment, however, writing test cases that use `assertEquals` directly on `MyString` objects wouldn't work, because we don't have an equality operation defined on `MyString`
  - We'll talk about how to implement *equality* carefully in next lecture
  - For now, the only operations we can perform with `MyStrings` are the ones we've defined above: `valueOf`, `length`, `charAt`, and `substring`
  - Our tests have to limit themselves to those operations

## Testing MyString (2)

---

- For example, here's one test for the `valueOf` operation:

```
MyString s = MyString.valueOf(true);
assertEquals(4, s.length());
assertEquals('t', s.charAt(0));
assertEquals('r', s.charAt(1));
assertEquals('u', s.charAt(2));
assertEquals('e', s.charAt(3));
```

- Next, let's look at a simple representation for `MyString`:  
just an array of characters, which is exactly the length of the string with no extra room at the end

# Representation of MyString: Array of Characters (1)

- Here's how that internal representation would be declared, as an instance variable within the class:

```
private char[] a;
```

- With that choice of representation, the operations would be implemented in a straightforward

```
public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
             : new char[] { 'f', 'a', 'l', 's', 'e' };
    return s;
}

public int length() {
    return a.length;
}

public char charAt(int i) {
    return a[i];
}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = new char[end - start];
    System.arraycopy(this.a, start, that.a, 0, end - start);
    return that;
}
```

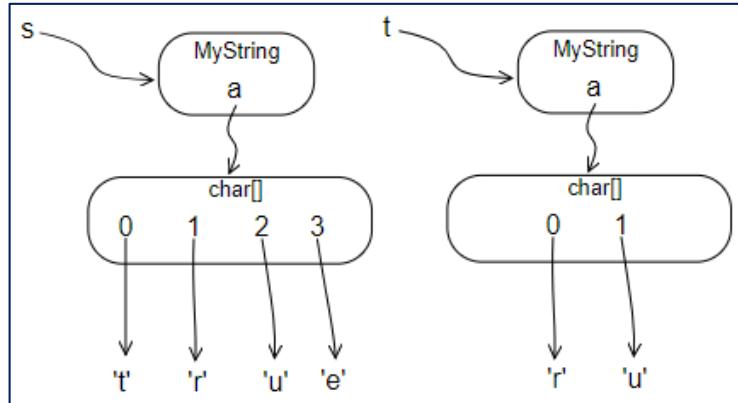
The `? ... : ...` syntax is called the ternary conditional operator and it's a shorthand if-else statement

Question to ponder: Why don't `charAt` and `substring` have to check whether their parameters are within the valid range? What do you think will happen if the client calls these implementations with illegal inputs?

# Representation of MyString: Array of Characters (2)

- Here's a snapshot diagram showing what this representation looks like for a couple of typical client operations:

```
MyString s = MyString.valueOf(true);  
MyString t = s.substring(1,3);
```



## Representation of MyString: Array of Characters (3)

---

- One problem with this implementation is that it's passing up an opportunity for performance improvement
  - Because this data type is immutable, the substring operation *doesn't really have to copy characters out into a fresh array*
  - It could just point to the original MyString object's character array and keep track of the start and end that the new substring object represents
    - The String implementation in some versions of Java do this!

# Representation of MyString: Array of Characters (4)

---

- To implement this optimization, we could change the *internal representation* and *operations* to:

```
private char[] a;
private int start;
private int end;

public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
             : new char[] { 'f', 'a', 'l', 's', 'e' };
    s.start = 0;
    s.end = s.a.length;
    return s;
}

public int length() {
    return end - start;
}

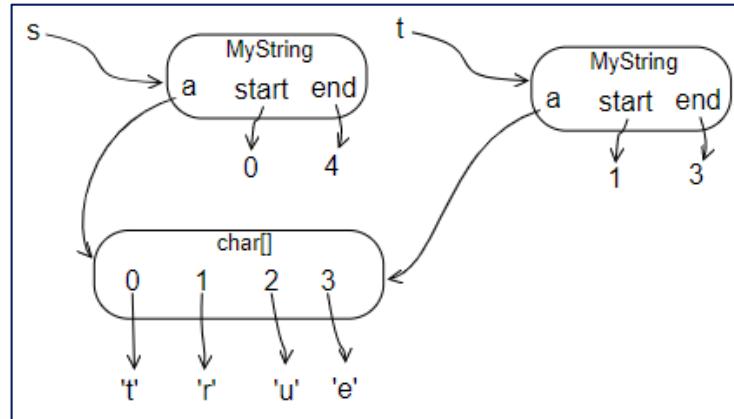
public char charAt(int i) {
    return a[start + i];
}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = this.a;
    that.start = this.start + start;
    that.end = this.start + end;
    return that;
}
```

# Representation of MyString: Array of Characters (5)

- Now the same client code produces a very different internal structure:

```
MyString s = MyString.valueOf(true);  
MyString t = s.substring(1,3);
```



## Representation of MyString: Array of Characters (6)

---

- Because MyString's existing clients ***depend only*** on the specs of its public methods, not on its private fields
  - we can make this change without having to inspect and change all that client code
  - that's the power of *representation independence* !

# Realizing ADT Concepts in Java (1)

---

- Let's summarize some of the general ideas we've discussed in this first part of the lecture, which are applicable in general to programming in any language, and their specific realization using Java language features
- The point is that there are several ways to do it, and it's important to both understand the big idea, like a creator operation, and different ways to achieve that idea in practice
- We'll also include three items that haven't yet been discussed so far, with notes about them in the next slide...

# Realizing ADT Concepts in Java (2)

| ADT concept        | Ways to do it in Java       | Examples                                     |
|--------------------|-----------------------------|----------------------------------------------|
| Abstract data type | Class                       | <code>String</code>                          |
|                    | Interface + class(es)       | <code>List</code> and <code>ArrayList</code> |
|                    | Enum                        | <code>DayOfWeek</code>                       |
| Creator operation  | Constructor                 | <code>ArrayList()</code>                     |
|                    | Static (factory) method     | <code>List.of()</code>                       |
|                    | Constant                    | <code>BigInteger.ZERO</code>                 |
| Observer operation | Instance method             | <code>List.get()</code>                      |
|                    | Static method               | <code>Collections.max()</code>               |
| Producer operation | Instance method             | <code>String.trim()</code>                   |
|                    | Static method               | <code>Collections.unmodifiableList()</code>  |
| Mutator operation  | Instance method             | <code>List.add()</code>                      |
|                    | Static method               | <code>Collections.copy()</code>              |
| Representation     | <code>private</code> fields |                                              |

# Realizing ADT Concepts in Java (3)

| ADT concept        | Ways to do it in Java                              | Examples                                                                                      |
|--------------------|----------------------------------------------------|-----------------------------------------------------------------------------------------------|
| Abstract data type | Class<br>Interface + class(es)<br>Enum             | <code>String</code><br><code>List</code> and <code>ArrayList</code><br><code>DayOfWeek</code> |
| Creator operation  | Constructor<br>Static (factory) method<br>Constant | <code>ArrayList()</code><br><code>List.of()</code><br><code>BigInteger.ZERO</code>            |
| Observer operation | Instance method<br>Static method                   | <code>List.get()</code><br><code>Collections.max()</code>                                     |
| Producer operation | Instance method<br>Static method                   | <code>String.trim()</code><br><code>Collections.unmodifiableList()</code>                     |
| Mutator operation  | Instance method<br>Static method                   | <code>List.add()</code><br><code>Collections.copy()</code>                                    |
| Representation     | <code>private</code> fields                        |                                                                                               |

Defining an abstract data type using an interface + class(es). We've seen `List` and `ArrayList` as an example, and we'll discuss **interface** in **Lecture Part 2** next

Defining an abstract data type using an enumeration (enum). Enums are ideal for ADTs that have a small fixed set of values, like the days of the week Monday, Tuesday, etc

Using a constant object as a creator operation. This pattern is commonly seen in immutable types, where the *simplest* or *emptiest* value of the type is simply a public constant, and producers are used to build up more complex values from it

# Testing an Abstract Data Type (1)

---

- We build a test suite for an abstract data type by creating tests for each of its operations
- These tests inevitably interact with each other
- The only way to test creators, producers, and mutators is by calling observers on the objects that result, and likewise, the only way to test observers is by creating objects for them to observe

# Testing an Abstract Data Type (2)

---

- Here's how we might partition the input spaces of the four operations in our MyString type:

```
// testing strategy for each operation of MyString:  
//  
// valueOf():  
//   true, false  
// length():  
//   string len = 0, 1, n  
//   string = produced by valueOf(), produced by substring()  
// charAt():  
//   string len = 1, n  
//   i = 0, middle, len-1  
//   string = produced by valueOf(), produced by substring()  
// substring():  
//   string len = 0, 1, n  
//   start = 0, middle, len  
//   end = 0, middle, len  
//   end-start = 0, n  
//   string = produced by valueOf(), produced by substring()
```

# Testing an Abstract Data Type (3)

---

- Then a compact test suite that covers all these partitions might look like:

```
@Test public void testValueOfTrue() {  
    MyString s = MyString.valueOf(true);  
    assertEquals(4, s.length());  
    assertEquals('t', s.charAt(0));  
    assertEquals('r', s.charAt(1));  
    assertEquals('u', s.charAt(2));  
    assertEquals('e', s.charAt(3));  
}  
  
@Test public void testValueOfFalse() {  
    MyString s = MyString.valueOf(false);  
    assertEquals(5, s.length());  
    assertEquals('f', s.charAt(0));  
    assertEquals('a', s.charAt(1));  
    assertEquals('l', s.charAt(2));  
    assertEquals('s', s.charAt(3));  
    assertEquals('e', s.charAt(4));  
}
```

# Testing an Abstract Data Type (4)

---

```
@Test public void testEndSubstring() {
    MyString s = MyString.valueOf(true).substring(2, 4);
    assertEquals(2, s.length());
    assertEquals('u', s.charAt(0));
    assertEquals('e', s.charAt(1));
}

@Test public void testMiddleSubstring() {
    MyString s = MyString.valueOf(false).substring(1, 2);
    assertEquals(1, s.length());
    assertEquals('a', s.charAt(0));
}

@Test public void testSubstringIsWholeString() {
    MyString s = MyString.valueOf(false).substring(0, 5);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testSubstringOfEmptySubstring() {
    MyString s = MyString.valueOf(false).substring(1, 1).substring(0, 0);
    assertEquals(0, s.length());
}
```

## Testing an Abstract Data Type (5)

---

- Notice that each test case typically calls a few operations that make or modify objects of the type (creators, producers, mutators) and some operations that inspect objects of the type (observers)
- As a result, each test case covers parts of several operations

## Part 2 : Inheritance

---

- In the second part of the lecture, we discuss Inheritance, Interface, and Iterable
- You have previously learned about Inheritance in CSE105
  - Please review those CSE105 materials and labs
  - We are also going to very quickly review it ...

# SLLList and ARLList

---

- Recall our two list implementations

```
public class ARLList<T>{  
    public ARLList()  
    public void addLast(T item)  
    public T getLast()  
    public T get(int i)  
    public int size()  
    public T delLast()  
}
```

```
public class SLLList<T>{  
    public SLLList()  
    public SLLList(T item)  
    public void addFirst(T item)  
    public void addLast(T item)  
    public T getFirst()  
    public T getLast()  
    public T get(int i)  
    public int size()  
    public T delLast()  
}
```

# SLLList and ARLList

---

- Recall our two list implementations
  - notice there are same method signatures that appear on both classes

```
public class ARLList<T>{  
    public ARLList()  
    public void addLast(T item)  
    public T getLast()  
    public T get(int i)  
    public int size()  
    public T delLast()  
}
```

```
public class SLLList<T>{  
    public SLLList()  
    public SLLList(T item)  
    public void addFirst(T item)  
    public void addLast(T item)  
    public T getFirst()  
    public T getLast()  
    public T get(int i)  
    public int size()  
    public T delLast()  
}
```

# A Method Taking SLList

---

- Suppose you wrote a method taking SLList of String in a class StrLib as follows
  - Now, you also want a similar method that takes ARList of String as well...

```
public static String longestStr(SLList<String> list) {  
    int max = 0;  
    for (int i = 0; i < list.size(); i++) {  
        String longestString = list.get(max);  
        String thisString = list.get(i);  
        if (thisString.length() > longestString.length()) {  
            max = i;  
        }  
    }  
    return list.get(max);  
}
```

# A Method Taking ARList

---

- You could create another copy of it, and make a little modification
  - You just did **method overloading** : same name, different parameter/signature!

```
public static String longestStr(ARList<String> list) {  
    int max = 0;  
    for (int i = 0; i < list.size(); i++) {  
        String longestString = list.get(max);  
        String thisString = list.get(i);  
        if (thisString.length() > longestString.length()) {  
            max = i;  
        }  
    }  
    return list.get(max);  
}
```

# Method Overloading

---

- **Method overloading** : multiple methods with same name, but different parameters
  - Java will figure out which method is going to get called

```
public static String longestStr(SLList<String> list) {  
    ...  
}
```

```
public static String longestStr(ARLlist<String> list) {  
    ...  
}
```

# Not a Good Idea

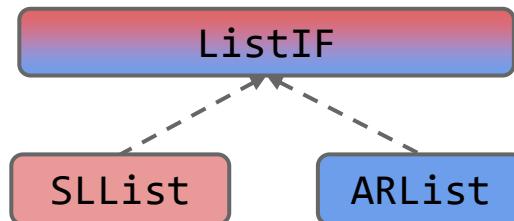
---

- While overloading works, it is a *bad* idea in this specific case of `longestStr`, for the following reasons:
  - Code is identical
  - Won't work for future lists
    - If you create a `NewList` class, you will have to make a third method
  - Harder to maintain
    - For example, suppose you find a bug in one of the methods
    - You fix it in the `SLList` version, but may forget to do it in the `ARList` version

# Is-A Relationships in Java

---

- SLList and ARList are both clearly some kind of a list
  - SLList is-a list
  - ARList is-a list
- Expressing this in Java is a two-step process:
  - Step 1: Define a *reference type* called **interface**, let's call this ListIF
  - Step 2: Specify that SLList and ARList are **implementation** of that type



# Step 1: Defining ListIF Interface

---

- We use the new keyword **interface** instead of class to define a ListIF interface
  - Interface is a specification of **what** a ListIF is able to do, **not how** to do it

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

end method signature with semicolon

every ListIF should be able to do get(i)

every ListIF should be able to do size()

## Step 2: Implementing ListIF Interface

---

- Use the `implements` keyword to tell the Java compiler that SLList and ARList are implementations of ListIF

```
public class ARList<T> implements ListIF<T> {  
    ...  
    public void addLast(T item) {  
        ...  
    }  
}
```

# One Method to rule them all

---



- We can now modify our longestStr method to work on either kind of list

```
public static String longestStr(ListIF<String> list) {  
    int max = 0;  
    for (int i = 0; i < list.size(); i++) {  
        String longestString = list.get(max);  
        String thisString = list.get(i);  
        if (thisString.length() > longestString.length()) {  
            max = i;  
        }  
    }  
    return list.get(max);  
}
```

# Method Overriding

- If a subclass has a method with the ***exact same signature*** as in the superclass, we say the subclass **overrides** the method

```
public interface ListIF<T> {  
    ...  
    public void addLast(T item);  
    ...
```

we say ARList overrides addLast

```
public class ARList<T> implements ListIF<T> {  
    ...  
    public void addLast(T item) {  
        ...
```

# Method Overriding vs Overloading

- Overriding: same signature
- Overloading: same method name, different signature
- Another example:

```
public interface Animal {  
    public void makeNoise();  
}
```

```
public class Dog implements Animal {  
    public void makeNoise(Dog d) {  
        ...  
    }  
}
```

```
public class Pig implements Animal {  
    public void makeNoise() {  
        System.out.print("oink");  
    }  
}
```

makeNoise is **overloaded**,  
it behaves differently when  
taking a Dog

Pig **overrides** makeNoise

# @Override Annotation (1)

---

- Tag every overriding method with the @Override annotation

```
/**  
 * Adds item to the end of the list.  
 * @param item is the new item.  
 */  
@Override  
public void addLast(T item) {  
    if (size == items.length) {  
        resize(capacity: 2 * size);  
    }  
    items[size] = item;  
    size++;  
}
```

## @Override Annotation (2)

- Tag every overriding method with the @Override annotation
  - The only effect of this tag is that the code **won't** compile if it is **not** actually an overriding method

```
/** Resizes the underlying array to the target capacity. */
@SuppressWarnings("unchecked")
@Override
private void resize(T[] a) {
    System.out.println("Extract interface Alt+Shift+Enter More actions... Alt+Enter");
    items = a;
}
```

for example, adding it to resize will generate compile error

# Why @Override ?

---

- Why use @Override?
  - protects against typos
    - if you tag a method with @Override, but it isn't actually overriding anything, you'll get a compile error  
e.g. `public void addLats(T item)`
    - otherwise, when you want to use the (not) overridden method of the subclass, the one that is actually used is the method of the superclass  
e.g. `addLats` is a whole new method
  - reminds programmer/code maintainer that method definition came from somewhere higher up in the inheritance hierarchy

# Interface Inheritance

---

- What we have seen so far in this lecture is Interface Inheritance, where we specify the capabilities of a subclass using the `implements` keyword
  - Interface: The list of all method signatures
  - Inheritance: The subclass *inherits* the interface from a superclass
  - The interface specifies what the subclass can do, but not how
  - Subclasses **must override all** of these methods!
    - it will fail to compile otherwise
- Interface inheritance is a powerful tool for generalizing code
  - `StrLib.longestStr` works on `SLList`, `ARList`, and even lists that have not yet been invented!

# Instantiation Example

---

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

- An SLLList object is created and its address is stored in the list1 variable of type ListIF
  - this is accepted by the compiler because SLLList implements ListIF
- Then, the string "a" is added into the SLLList object by addLast of SLLList
  - because addLast of ListIF is overridden by addLast of SLLList

# Implementation Inheritance

---

- Another type of inheritance is called **Implementation Inheritance**
- Interface inheritance: subclass inherits signatures, but NOT implementation
  - Implementation inheritance: subclass inherits signatures AND implementation
- Inside an interface, we use the **default** keyword to specify a method that subclasses should inherit from that interface
  - For example, let's add **a default print() method** to ListIF interface

# Default Method

---

- Default method example:

```
public interface ListIF<T> {  
  
    public void addLast(T item);  
    public T getLast();  
    public T delLast();  
    public T get(int i);  
    public int size();  
  
    default public void print() {  
        for (int i = 0; i < size(); i++) {  
            System.out.print(get(i) + " ");  
        }  
    }  
}
```

we can use the methods defined above in default print method

this print method will be inherited by both SLList and ARList. Is this an efficient implementation for them?

# Default Method

---

- Default method example:

```
public interface ListIF<T> {  
  
    public void addLast(T item);  
    public T getLast();  
    public T delLast();  
    public T get(int i);  
    public int size();  
  
    default public void print() {  
        for (int i = 0; i < size(); i++) {  
            System.out.print(get(i) + " ");  
        }  
    }  
}
```

we can use the methods defined above in default print method

because of this `get(i)`, this print implementation is efficient for ARList, but **not** for SLList!

let's override it in SLList!

# Overriding Default Method

---

- This time by overriding, we are not only implement the method, but really **re-implement** the superclass's method
  - In SLLList:

```
@Override  
public void print() {  
    Node p = sentinel.next;  
    while (p != null) {  
        System.out.print(p.item + " ");  
        p = p.next;  
    }  
}
```

overriding default print  
with a more efficient code

# Recalling the use of @Override annotation

---

- Suppose you don't annotate print with @Override, and you misspell print

```
public void prnit() {  
    Node p = sentinel.next;  
    while (p != null) {  
        System.out.print(p.item + " ");  
        p = p.next;  
    }  
}
```

the code still compiles, but when you call print on objects of SLList, the default print will be called !

# Which Method Selected

---

- Recall that if X is a superclass of Y, then an X variable can hold a reference to a Y

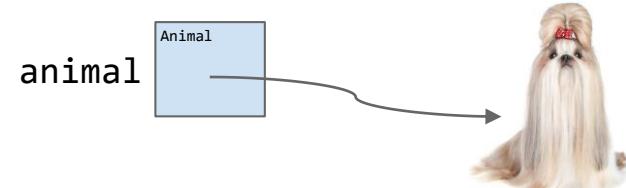
```
public static void main(String[] args) {  
    ListIF<String> list2 = new SLLList<String>();  
    list2.addLast("abc");  
    list2.print();  
}
```

- Which print method will run when the code above executes?
  - SLLList.print(), and not ListIF.print()
  - How does it work?
    - Before we can answer, we need to understand static and dynamic type

# Static Type vs Dynamic Type

- Every variable in Java has a ***compile-time*** type = **static** type
  - This is the type specified at declaration and it *never* changes!
- Variables also have a ***run-time*** type = **dynamic** type
  - This is the type specified *at instantiation* (e.g. when using new)
  - Equal to the type of the object *being* pointed at

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

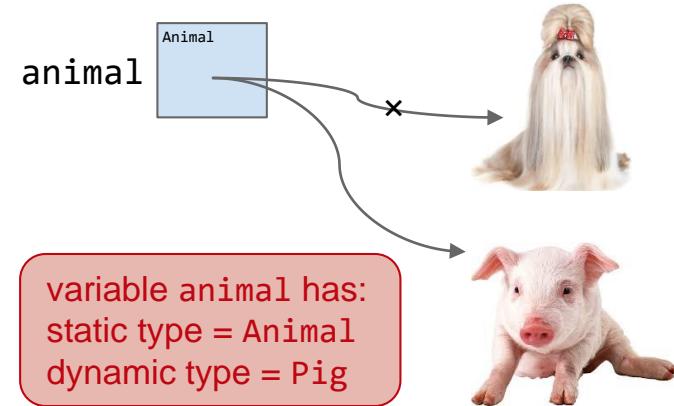


variable animal has:  
static type = Animal  
dynamic type = Dog

# Static Type vs Dynamic Type

- Every variable in Java has a ***compile-time*** type = **static** type
  - This is the type specified at declaration and it *never* changes!
- Variables also have a ***run-time*** type = **dynamic** type
  - This is the type specified *at instantiation* (e.g. when using new)
  - Equal to the type of the object *being* pointed at

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



# Dynamic Method Selection for Overridden Methods

---

- Suppose we call a method `m()` of an object using a variable with:
  - static type X and dynamic type Y
- First, the compiler records the X's method `m()`, to be used in *run-time*
- At *run-time*, if Y overrides the method `m()`, then Y's method `m()` is used *instead*
  - this is known as **dynamic method selection**

# Dynamic Method Selection for Print()

- Suppose we call a method `m()` of an object using a variable with:
  - static type X and dynamic type Y
- First, the compiler records the X's method `m()`, to be used in *run-time*
- At *run-time*, if Y overrides the method `m()`, then Y's method `m()` is used *instead*
  - this is known as **dynamic method selection**
- Therefore, `print()` that belongs to `SLList` is used:

```
public static void main(String[] args) {  
    ListIF<String> list2 = new SLList<String>();  
    list2.addLast("abc");  
    list2.print();  
}
```

# A Puzzle

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

**stop!**

try to guess and write your answers  
in a piece of paper!

# In-Class Quiz 2.1

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

What is the output of this line ?

## In-Class Quiz 2.2

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

What is the output of this line ?

# In-Class Quiz 2.3

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

What is the output of this line ?

# In-Class Quiz 2.4

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

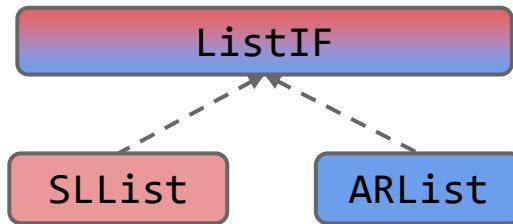
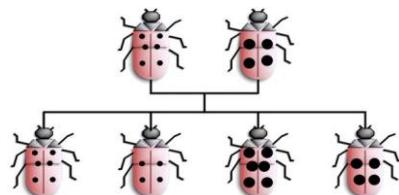
```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

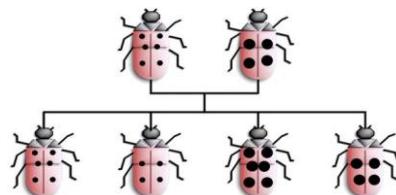
What is the output of this line ?

# Implementation Inheritance 2

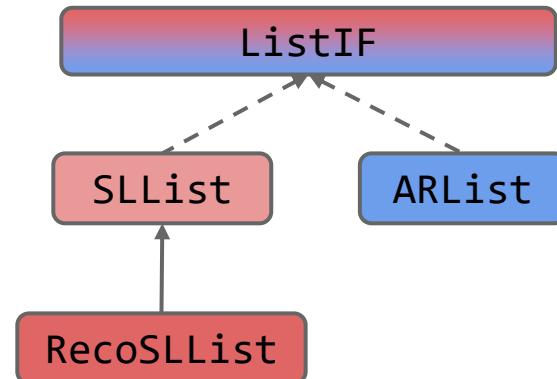
- If you want one class to be a subclass of another class (instead of an interface), you use the keyword **extends**



# Implementation Inheritance 2



- If you want one class to be a subclass of another class (instead of an interface), you use the keyword **extends**
- For example, you want to build an **SLList** that
  - Recollects all items that have been deleted by `delLast`
  - Has an additional method `printDelItems()`, which prints all deleted items
- Let's call it **RecoSLList**, and
  - make it to be a subclass of **SLList**



# Extends

- Recall that because of extends, RecoSLLList inherits all members of SLLList:
  - All instance and static variables
  - All methods
  - All nested classes
- Constructors are **not** inherited

even though inherited, the private members will not be accessible !

```
public class RecoSLLList<T> extends SLLList<T> {  
    . . .  
  
    public void printDelItems() {  
        . . .  
    }  
}
```

next, we will implement it  
you may want to **try it on your own** first!

# Storing Deleted Items

---

- To recollect and print the deleted items, we need store them

```
SLList<T> deletedItems;  
  
/** Prints deleted items. */  
public void printDelItems() {  
    deletedItems.print();  
}
```

let us choose to store them in an SLList!  
yes, we can do that!

we can then use the print method  
we implement last time

next, we want that deleted items get  
automatically added into deletedItems.  
how do we do that?

# Overriding delLast

- You want to add another step to delLast, so that it stores the deleted item as well

```
@Override  
public T delLast() {  
    T x = super.delLast(); ←  
    deletedItems.addLast(x); ←  
    return x;  
}
```

first, you want to do "normal" deletion, however if you copy paste delLast code from SLList, you cannot access the private member variables

so, you can use **super** to call the delLast that belongs to SLList, the *superclass*

and then, store the deleted item

we still need to do one more thing to our class  
can you guess what?

# Constructor for RecoSLLList 1

---

- We use the constructor to initialize our new member variable: deletedItems

```
public RecoSLLList() {  
    deletedItems = new SLLList<T>();  
}
```

# Constructor for RecoSLLList 2

---

- We use the constructor to initialize our new member variable: deletedItems

```
public RecoSLLList() {  
    deletedItems = new SLLList<T>();  
}
```

- In Java, every subclass constructor must call one of the superclass's constructor
  - if we don't, like we did above, Java will add the superclass's default constructor, so what we have is actually:

```
public RecoSLLList() {  
    super(); ←  
    deletedItems = new SLLList<T>();  
}
```

SLLList default constructor will initialize sentinel node and size which RecoSLLList also inherited !

## Constructor for RecoSLLList 3

---

- Alternatively, you may want to overload the constructor with another constructor taking 1 element
  - in this case, you have to specifically call the constructor of the superclass that takes 1 element
  - otherwise, super() will be added and called instead, and that item is lost forever!

```
public RecoSLLList(T x) {  
    super(x);  
    deletedItems = new SLLList<T>();  
}
```

call SLLList(T x) instead of SLLList()

## Last week on Iterator

---

- Last week, we have learned a design pattern for iterating elements using **another class** separated from the one (the data structure) we are iterating
  - we pass the data structure into the class using the constructor
  - we use method `hasNext` to check whether there are still elements to iterate
  - we use method `Next` to
    - retrieve the element
    - move to the next element in the data structure

# This week on Iterator

---

- Last week, we have learned a design pattern for iterating elements using **another class** separated from the one (the data structure) we are iterating
  - we pass the data structure into the class using the constructor
  - we use method `hasNext` to check whether there are still elements to iterate
  - we use method `Next` to
    - retrieve the element
    - move to the next element in the data structure
- Now, we want use use the **same class** to give us an iterator that can iterate itself!
  - we also want to make that data structure class to be *iterable* using the enhanced for loop (also known as for-each loop)

# Making your data structure iterable

---

- Three steps to make your data structure class DS can be iterated using the enhanced for loop:
  1. The DS must implement an `iterator()` method that return an `Iterator` object
  2. The `Iterator` must implement the methods `next()` and `hasNext()`
  3. You have to tell the world that DS has implemented 1

# Making your data structure iterable

---

- Three steps to make your data structure class DS can be iterated using the enhanced for loop:
  1. The DS must implement an `iterator()` method that return an `Iterator` object
    - so, since we only have one public class, we have to use **private inner class `DSIterator`** to implement the **Iterator interface**, and then `iterator()` can instantiate it and return its object
  2. The Iterator must implement the methods `next()` and `hasNext()`
    - you have done this last week!
  3. You have to tell the world that DS has implemented 1
    - we use standard interface for this: **Iterable interface**

let's make our `ARList` iterable as an example!

# Step 1

---

- The DS must implement an `iterator()` method that return an `Iterator` object

```
import java.util.Iterator;
```

Iterator is defined here, need to import it

```
/**  
 * Make an iterator  
 */  
  
public Iterator<T> iterator() { return new ARLListIterator(); }
```

the method `iterator()` needs to return an `Iterator`

we will have to define this inner class,  
and it has to implement the `Iterator` interface,  
so that its object is indeed an `Iterator`

## Step 2

---

- The Iterator must implement the methods `hasNext()` and `next()`
  - ARListIterator is-an Iterator, implement this interface

```
private class ARListIterator implements Iterator<T> {  
    private int index;  
  
    public ARListIterator() { index = 0; }  
  
    @Override  
    public boolean hasNext() { return index < size; }  
  
    @Override  
    public T next() {  
        T nextItem = items[index];  
        index++;  
        return nextItem;  
    }  
}
```

we have learned about this last week

being an inner class inside ARList,  
it has direct access to ARList  
member variables

being an Iterator, ARListIterator  
must have `hasNext()` and `next()`,  
override them!

## Step 3

---

- You have to tell the world that DS has implemented 1
  - without this step 3, you can already use the "naked" enhanced for loop

```
// naked enhanced for Loop
Iterator<Integer> iter = intList.iterator();
while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

## Step 3

---

- You have to tell the world that DS has implemented 1
  - without this step 3, you can already use the "naked" enhanced for loop

```
// naked enhanced for Loop
Iterator<Integer> iter = intList.iterator();
while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

- but to be able to **use enhanced for loop** you need *another interface*:

```
public class ARList<T> implements ListIF<T>, Iterable<T> {
```

now you can do:

```
// enhanced for Loop / for-each Loop
for(int item : intList) {
    System.out.println(item);
}
```

to be an Iterable, must implement iterator()

we tell Java that we have implemented it,  
so Java can use it to do enhanced for loop

# Thank you for your attention !

---

- In this lecture, you have learned:
  - about ADTs: their operations, specs, good ones, and to test ones
  - to use representation independence so that the implementation of an ADT can change without requiring changes from its clients
  - to use interface to define data structure operations
  - to differentiate overriding versus overloading a method
  - to reuse codes from parent class
  - to make user-defined data structure iterable
- Please continue to Lecture Quiz 10 and Lab 10:
  - to do Lab Exercise 10.1 - 10.3, and
  - to do Exercise 10.1 - 10.3

|          |  |
|----------|--|
| The Best |  |
| Amazing  |  |
| Great    |  |
| Fine     |  |
| Bad      |  |
| Terrible |  |

# Advanced Object-Oriented Programming

CPT204 – Lecture 11  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming

### Lecture 11

**Invariant, Abstraction Function,  
Comparability, Equality**

# Welcome !

---

- Welcome to Lecture 11 !
- In this lecture we are going to
  - learn to implement an ADT using *abstraction functions* and *representation invariants*
  - learn about *casting*: what it is and what is it not
  - learn how to make an instance of your data structure can be *compared* to another
  - learn to define the *equality* operation for an ADT

# Invariant and Abstraction Function

---

- In the first part of the lecture, we discuss about Invariant, that we have used before
- We will also going to introduce several ideas:
  - Representation exposure
  - Abstraction functions
  - Representation invariants
- We study a more formal idea of what it means for a class to implement an ADT, via the notions of abstraction functions and rep invariants
  - The abstraction function will give us a way to cleanly define the equality operation on an abstract data type
  - The rep invariant will make it easier to catch bugs caused by a corrupted data structure

# Invariant (1)

---

- Resuming our discussion of what makes a good abstract data type, the final, and perhaps most important, property of a good abstract data type is that **it preserves its own invariants**
- An invariant is a property of a program that is *always true*, for every possible runtime state of the program
- Immutability is one crucial invariant that we've already encountered: once created, an immutable object should always represent the same value, for its entire lifetime
- Saying that the ADT preserves its own invariants means that the ADT is *responsible* for ensuring that its own invariants hold
  - it *doesn't* depend on good behavior from its clients

## Invariant (2)

---

- When an ADT preserves its own invariants, reasoning about the code becomes much easier
  - If you can count on the fact that Strings never change, you can rule out that possibility when you're debugging code that uses Strings — or when you're trying to establish an invariant for another ADT that uses Strings
  - Contrast that with a string type that guarantees that it will be immutable only if its clients promise not to change it
    - then you'd have to check all the places in the code where the string might be used

# Immutability (1)

---

- In this example, we want to have immutability as an invariant:

```
/**  
 * This immutable data type represents a tweet from Twitter.  
 */  
public class Tweet {  
  
    public String author;  
    public String text;  
    public Date timestamp;  
  
    /**  
     * Make a Tweet.  
     * @param author    Twitter user who wrote the tweet  
     * @param text      text of the tweet  
     * @param timestamp date/time when the tweet was sent  
     */  
    public Tweet(String author, String text, Date timestamp) {  
        this.author = author;  
        this.text = text;  
        this.timestamp = timestamp;  
    }  
}
```

## Immutability (2)

---

- How do we guarantee that these Tweet objects are immutable — that, once a tweet is created, its author, message, and date can never be changed?
- The first threat to immutability comes from the fact that clients can — in fact must — directly access its fields
  - So nothing's stopping us from writing code like this:

```
Tweet t = new Tweet("Donald Trump", "Despite the constant negative  
press covfefe", new Date());  
t.author = "abcde";
```

## Immutability (3)

---

- This is a trivial example of **representation exposure**, meaning that code outside the class can modify the representation directly
- Rep exposure like this threatens not only invariants, but also *representation independence*
  - we can't change the implementation of Tweet without affecting all the clients who are directly accessing those fields

# Immutability (4)

---

- Fortunately, Java gives us language mechanisms to deal with this kind of rep exposure:

```
public class Tweet {  
  
    private final String author;  
    private final String text;  
    private final Date timestamp;  
  
    public Tweet(String author, String text, Date timestamp) {  
        this.author = author;  
        this.text = text;  
        this.timestamp = timestamp;  
    }  
  
    /** @return Twitter user who wrote the tweet */  
    public String getAuthor() {  
        return author;  
    }  
  
    /** @return text of the tweet */  
    public String getText() {  
        return text;  
    }  
  
    /** @return date/time when the tweet was sent */  
    public Date getTimestamp() {  
        return timestamp;  
    }  
}
```

## Immutability (5)

---

- The `private` and `public` keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class
- The `final` keyword also helps by guaranteeing that the fields of this immutable type *won't be reassigned* after the object is constructed

## Immutability (6)

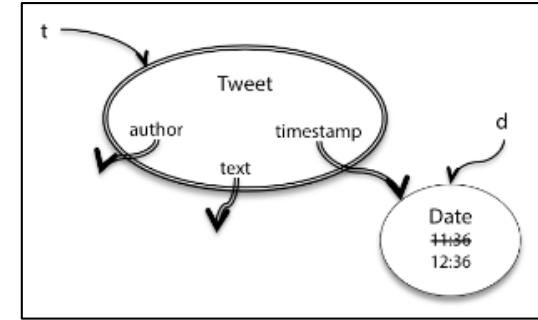
---

- But that's not the end of the story: the rep is still exposed!
- Consider this perfectly reasonable client code that uses Tweet

```
/**  
 * @return a tweet that retweets t, one hour later  
 */  
public static Tweet retweetLater(Tweet t) {  
    Date d = t.getTimestamp();  
    d.setHours(d.getHours() + 1);  
    return new Tweet("abcde", t.getText(), d);  
}
```

## Immutability (7)

- `retweetLater` takes a tweet and should return another tweet with the same message (called a retweet) but sent an hour later
- The `retweetLater` method might be part of a system that automatically echoes funny things that Twitter celebrities say
- What's the problem here?
- The `getTimestamp` call returns *a reference* to the same `Date` object referenced by tweet `t`
  - `t.timestamp` and `d` are aliases to the same mutable object
  - So when that `Date` object is mutated by `d.setHours()`, this affects the date in `t` as well, as shown in the snapshot diagram



## Immutability (8)

---

- Tweet's immutability invariant has been broken
  - the problem is that Tweet *leaked out* a reference to a mutable object that its immutability depended on
- We exposed the rep, in such a way that Tweet can no longer guarantee that its objects are immutable
  - perfectly reasonable client code created a subtle bug
- We can patch this kind of rep exposure by using **defensive copying**: making a copy of a mutable object to avoid leaking out references to the rep

## Immutability (9)

---

- Here's the code using defensive copying:

```
public Date getTimestamp() {  
    return new Date(timestamp.getTime());  
}
```

- Mutable types often have a **copy constructor** that allows you to make a new instance that duplicates the value of an existing instance
  - In this case, Date's copy constructor uses the timestamp value, measured in milliseconds since January 1, 1970
- As another example, StringBuilder's copy constructor takes a String
- Another way to copy a mutable object is `clone()`, which is supported by some types but not all

# Immutability (10)

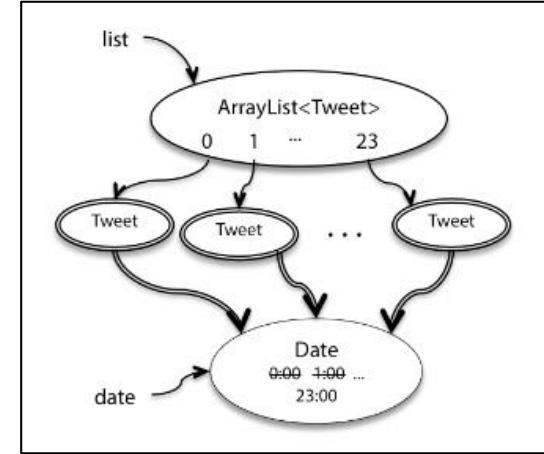
---

- So we've done some defensive copying in the return value of `getTimestamp`
- But we're not done yet! There's still rep exposure; consider this client code:

```
/**  
 * @return a list of 24 inspiring tweets, one per hour today  
 */  
public static List<Tweet> tweetEveryHourToday() {  
    List<Tweet> list = new ArrayList<Tweet>();  
    Date date = new Date();  
    for (int i = 0; i < 24; i++) {  
        date.setHours(i);  
        list.add(new Tweet("abcde", "jiāyóu! you can do it!", date));  
    }  
    return list;  
}
```

# Immutability (11)

- The code intends to advance a single Date object through the 24 hours of a day, creating a tweet for every hour
- But notice that the constructor of Tweet saves the reference that was passed in, so all 24 Tweet objects end up with the same time, as shown in this snapshot diagram
  - Again, the immutability of Tweet has been violated
- We can fix this problem too by using **defensive copying**, this time in the constructor:



```
public Tweet(String author, String text, Date timestamp) {  
    this.author = author;  
    this.text = text;  
    this.timestamp = new Date(timestamp.getTime());  
}
```

## Immutability (12)

---

- In general, you should carefully inspect the argument types and return types of all your ADT operations
  - If any of the types are mutable, make sure your implementation doesn't return direct references to its representation
  - Doing that creates rep exposure!
- You may object that this seems wasteful
  - Why make all these copies of dates?
- Why can't we just solve this problem by a carefully written specification?

# Immutability (13)

---

- Why can't we just solve this problem by a carefully written specification, like this:

```
/**  
 * Make a Tweet.  
 * @param author    Twitter user who wrote the tweet  
 * @param text      text of the tweet  
 * @param timestamp date/time when the tweet was sent.  
 *                  Caller must never mutate this Date object again!  
 */  
public Tweet(String author, String text, Date timestamp) {
```

# Immutability (14)

---

- This approach is sometimes taken when there isn't any other reasonable alternative — for example, when the mutable object is too large to copy efficiently
  - But the cost in your ability to reason about the program, and your ability to avoid bugs, is enormous
  - In the absence of compelling arguments to the contrary, it's almost always worth it for an abstract data type to guarantee its own invariants, and preventing rep exposure is essential to that
- An even better solution is to prefer immutable types
  - If — as recommended in Mutability and Immutability Groundhog Day example in Lecture 9 — we had used an immutable date object, like `java.time.ZonedDateTime`, instead of the mutable `java.util.Date`, then we would have ended this part of lecture after talking about `public` and `private`
  - **No** further rep exposure would have been possible

# Immutable Wrappers Around Mutable Data Types

---

- The Java collections classes offer an interesting compromise: **immutable wrappers**
  - we discussed about these before in the lecture on Mutability and Immutability
- `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()` 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 here is that *you get immutability at runtime*, but **not** at compile time
  - 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

# Abstract Space vs Rep Space (1)

---

- We now take a deeper look at the theory underlying abstract data types
  - This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types
  - If you understand the theory deeply, you'll be able to build better abstract types, and will be less likely to fall into subtle traps
- In thinking about an abstract type, it helps to consider the relationship between two spaces of values
- **The space of abstract values** consists of the values that the type is *designed to support*, from the client's point of view
  - for example, an abstract type for unbounded integers, like Java's BigInteger, would have the mathematical integers as its abstract value space

## Abstract Space vs Rep Space (2)

---

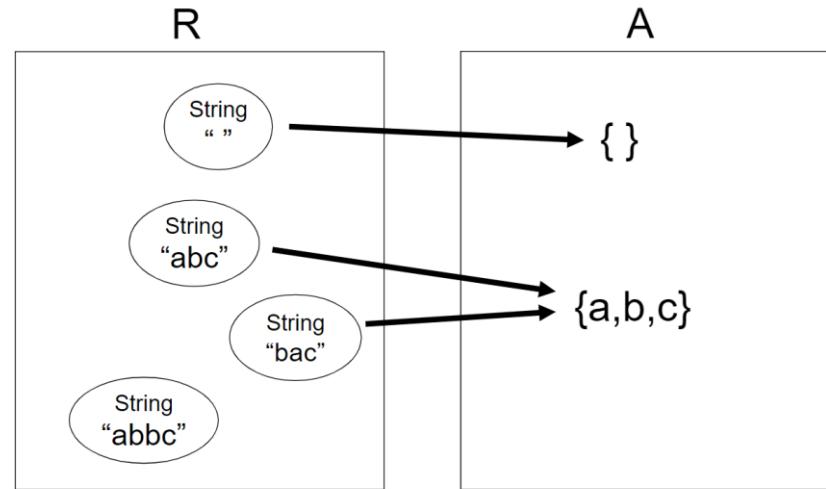
- The space of representation values (or rep values for short) consists of *the Java objects that actually implement* the abstract values
  - for example, a BigInteger value might be implemented using *an array of digits*, represented as primitive int values
  - The rep space would then be *the set of all such arrays*
- In simple cases, an abstract type will be implemented as *a single Java object*, but more commonly a small network of objects is needed
  - for example, a rep value for List might be a linked list, a group of objects linked together by next and previous pointers; so a rep value is not necessarily a single object, but often something rather complicated
- Now of course the implementer of the abstract type must be interested in the representation values, since it is the implementer's job to *achieve the illusion* of the abstract value space using the rep value space

# Abstract Space vs Rep Space (3)

- Suppose, for example, that we choose to use a *string* to represent *a set of characters*:

```
public class CharSet {  
    private String s;  
    ...  
}
```

- Then the **rep space R** contains *Strings*, and the **abstract space A** is *mathematical sets of characters*
- We can show the two value spaces graphically, with an arc from a rep value to the abstract value it represents



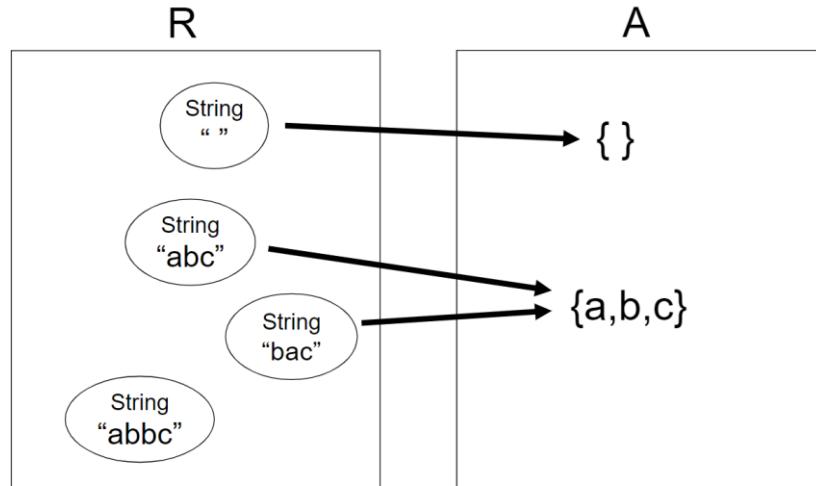
# Abstract Space vs Rep Space (4)

There are several things to note about this picture:

- **Every abstract value is mapped to by some rep value**

The purpose of implementing the abstract type is to support operations on abstract values;

Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable

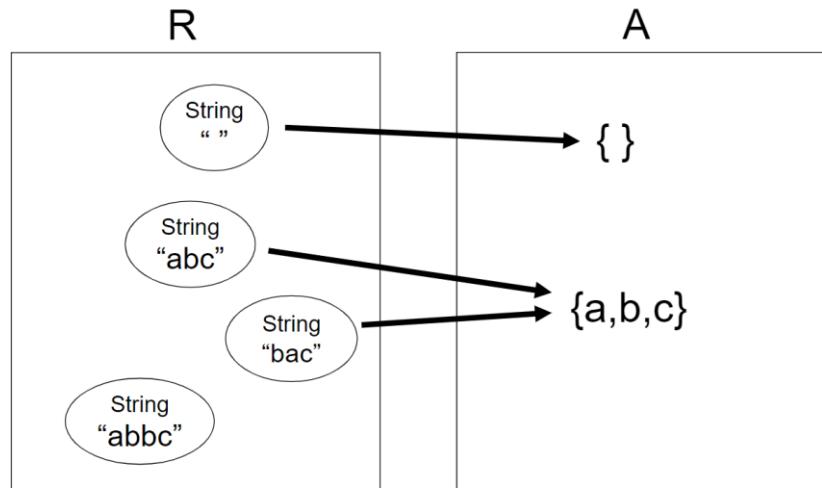


# Abstract Space vs Rep Space (5)

---

- Some abstract values are mapped to by more than one rep value

This happens because the representation isn't a tight encoding; There's more than one way to represent an unordered set of characters as a string

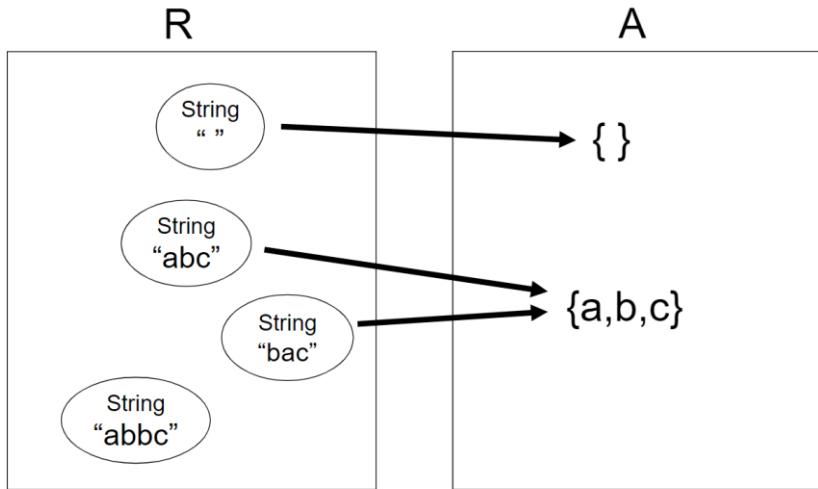


# Abstract Space vs Rep Space (6)

---

- **Not all rep values are mapped**

Notice the string “abbc” is not mapped;  
In this case, we have decided that the string should not contain duplicates;  
This will allow us to terminate the remove method when we hit the first instance of a particular character, since we know there can be at most one



# Rep Invariant and Abstraction Function (1)

---

In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite

So we describe it by giving two things:

- An **abstraction function** that maps rep values to the abstract values they represent:

$$AF : R \rightarrow A$$

The *arcs* in the diagram show the abstraction function

In the terminology of functions, the properties we discussed above can be expressed by saying that the function is *surjective* (also called *onto*), *not necessarily injective* (also called *one-to-one*), therefore *not necessarily bijective*; and often *partial*

how to interpret  
rep values as  
abstract values



## Rep Invariant and Abstraction Function (2)

---

- A **rep invariant** that maps rep values to booleans:

$RI : R \rightarrow \text{boolean}$

For a rep value  $r$ ,  $RI(r)$  is true if and only if  $r$  is mapped by AF

In other words,  $RI$  tells us whether a given rep value is well-formed

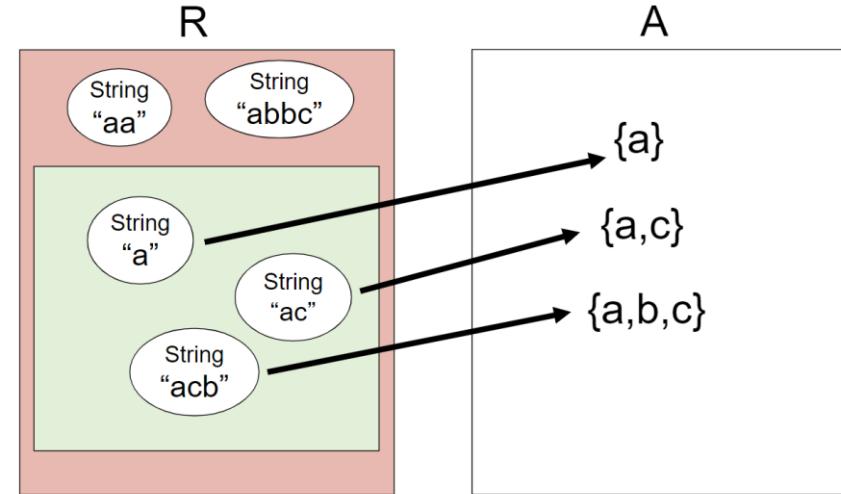
Alternatively, you can think of  $RI$  as a set: it's the subset of rep values on which AF is defined

which rep values  
are legal



# Rep Invariant and Abstraction Function Example 1 (1)

- For example, the diagram at the right showing a rep for CharSet that forbids repeated characters
  - $RI("a") = \text{true}$ ,  $RI("ac") = \text{true}$  and  $RI("acb") = \text{true}$
  - but  $RI("aa") = \text{false}$  and  $RI("abbc") = \text{false}$

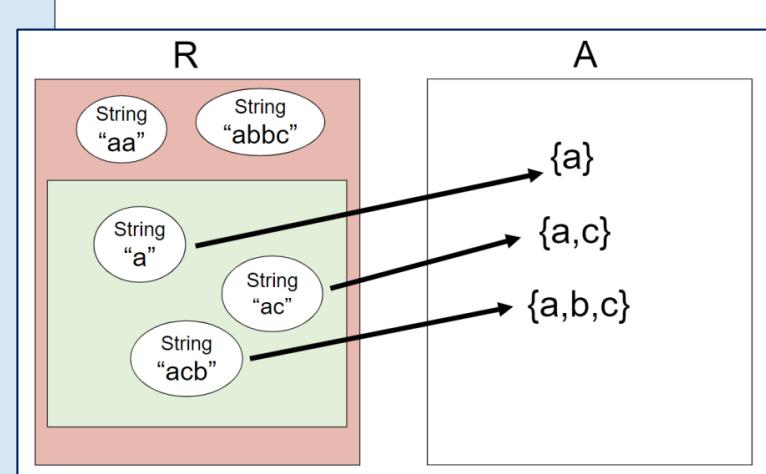


- Rep values that *obey* the rep invariant are shown in the green part of the R space, and must map to an abstract value in the A space
- Rep values that *violate* the rep invariant are shown in the red zone, and have no equivalent abstract value in the A space

# Rep Invariant and Abstraction Function Example 1 (2)

- Both the rep invariant and the abstraction function should be documented in the code, right next to the declaration of the rep itself:

```
public class CharSet {  
    private String s;  
  
    // Rep invariant:  
    // s contains no repeated characters  
  
    // Abstraction function:  
    // AF(s) = {s[i] | 0 <= i < s.length()}  
  
    ...  
}
```



# Rep Value Spaces, Rep Invariant, Abstraction Function

---

- A *common confusion* about abstraction functions and rep invariants is that they are determined by the choice of rep and abstract value spaces, or even by the abstract value space alone
  - if this were the case, they would be of little use, since they would be saying something redundant that's already available elsewhere
- The abstract value space *alone* **doesn't determine** AF or RI: there can be **several** representations for the **same** abstract type
  - A set of characters could equally be represented as *a string*, as above, or as *a bit vector*, with one bit for each possible character
  - Clearly we need two *different* abstraction functions to map these two different rep value spaces

## Rep Invariant and Abstraction Function Example 2 (1)

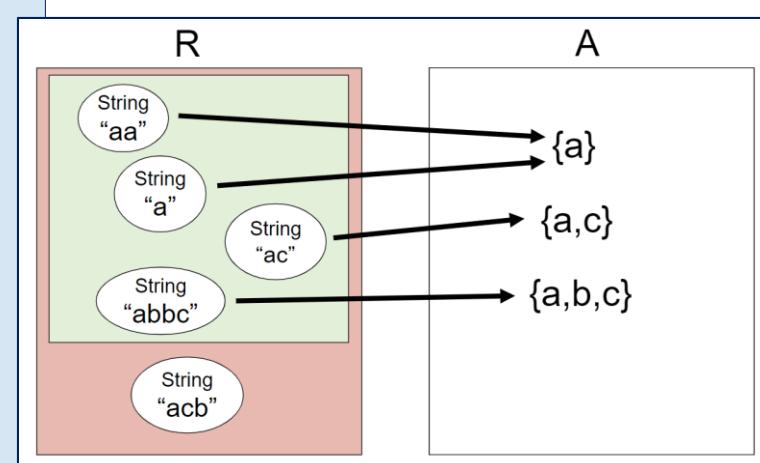
---

- It's less obvious why the choice of both spaces doesn't determine AF and RI
  - The key point is that defining a type for the rep, and thus choosing the values for the space of rep values, does *not* determine *which of the rep values will be deemed to be legal*, and of those that are legal, *how they will be interpreted*
  - Rather than deciding, as we did above, that the strings have no duplicates, we could instead *allow duplicates*, but at the same time require that the characters be *sorted*, appearing in nondecreasing order
    - This would allow us to perform a binary search on the string and thus check membership in logarithmic rather than linear time

## Rep Invariant and Abstraction Function Example 2 (2)

- Same rep value space — different rep invariant:

```
public class CharSet {  
    private String s;  
    // Rep invariant:  
    //   s[0] <= s[1] <= ... <= s[s.length()-1]  
    // Abstraction function:  
    //   AF(s) = {s[i] | 0 <= i < s.length()}  
    ...  
}
```



## Rep Invariant and Abstraction Function Example 3 (1)

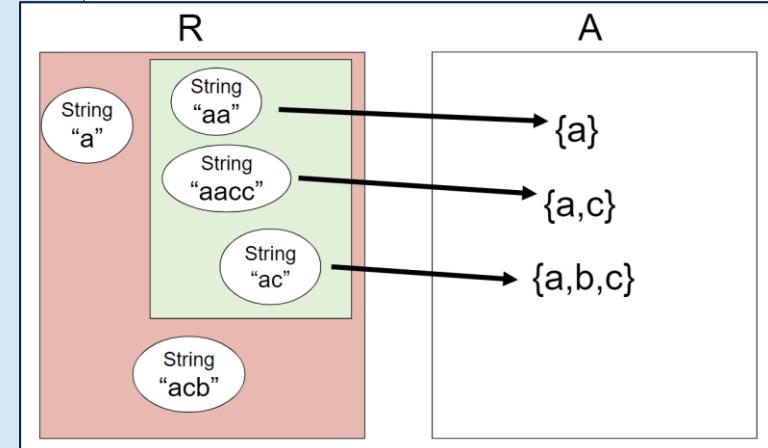
---

- Even with the *same type* for the rep value space and the same rep invariant RI, we might still interpret the rep differently, with *different* abstraction functions AF
- Suppose RI admits any string of characters
- Then we could define AF, as above, to interpret the array's elements as the elements of the set
- But there's no a priori reason to let the rep decide the interpretation
  - Perhaps we'll interpret consecutive pairs of characters as subranges, so that the string rep “acgg” is interpreted as two range pairs, [a-c] and [g-g], and therefore represents the set {a, b, c, g}

# Rep Invariant and Abstraction Function Example 3 (2)

- Here's what the AF and RI would look like for that representation:

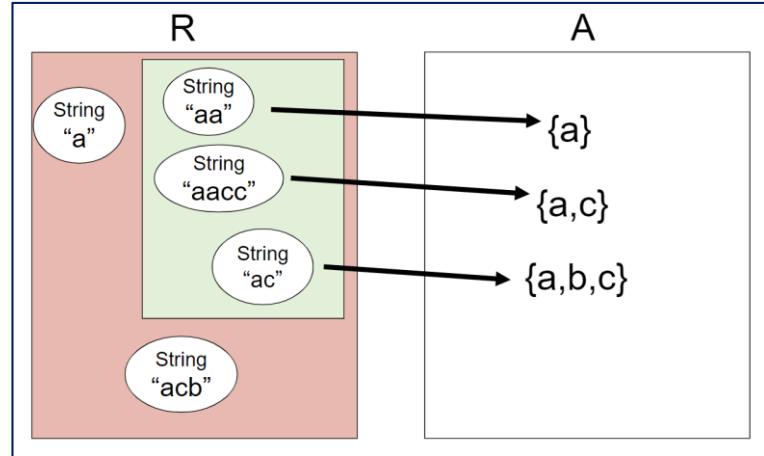
```
public class CharSet {  
    private String s;  
    // Rep invariant:  
    //   s.length() is even  
    //   s[0] <= s[1] <= ... <= s[s.length()-1]  
    // Abstraction function:  
    //   represents the union of the ranges  
    //   {s[i]...s[i+1]} for each adjacent pair  
    //   of characters in s  
    ...  
}
```



# In-Class Quiz 1.1

- Consider the rep of CharSet in Example 3:

```
public class CharSet {  
    private String s;  
    // Rep invariant:  
    //   s.length() is even  
    //   s[0] <= s[1] <= ... <= s[s.length()-1]  
    // Abstraction function:  
    //   represents the union of the ranges  
    //   {s[i]...s[i+1]} for each adjacent pair  
    //   of characters in s  
    ...  
}
```



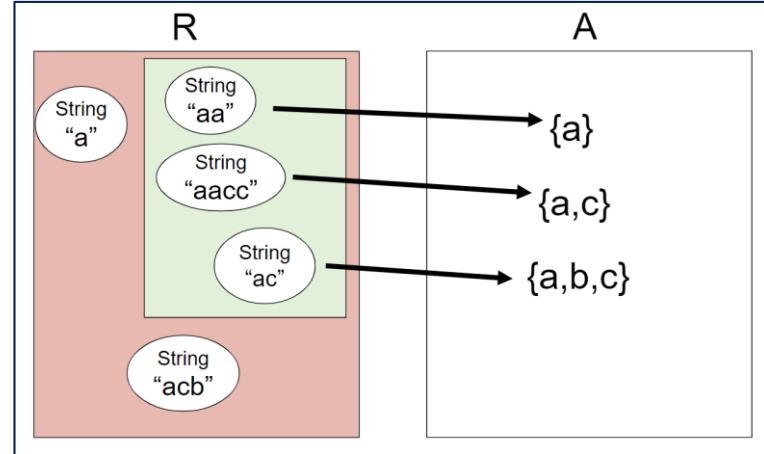
- Which of the following values of s satisfy this rep invariant?

- |                                 |                                 |
|---------------------------------|---------------------------------|
| <input type="checkbox"/> "abc"  | <input type="checkbox"/> "ad"   |
| <input type="checkbox"/> "abcd" | <input type="checkbox"/> "adad" |
| <input type="checkbox"/> "eeee" | <input type="checkbox"/> ""     |

# In-Class Quiz 1.2

- Consider the rep of CharSet in Example 3:

```
public class CharSet {  
    private String s;  
    // Rep invariant:  
    //   s.length() is even  
    //   s[0] <= s[1] <= ... <= s[s.length()-1]  
    // Abstraction function:  
    //   represents the union of the ranges  
    //   {s[i]...s[i+1]} for each adjacent pair  
    //   of characters in s  
    ...  
}
```

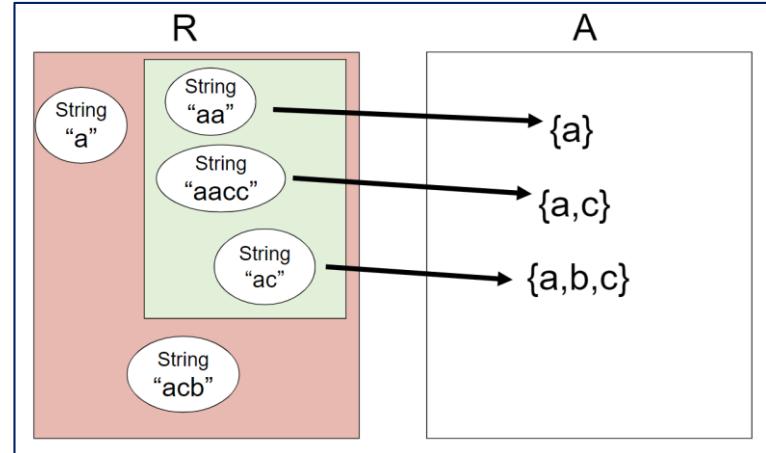


- Which of the following does AF("acfg") map to?
  - {a,b,c,d,e,f,g}
  - {a,b,c,f,g}
  - {a,c,f,g}
  - some other abstract value
  - no abstract value, because "acfg" does not satisfy the rep invariant

# In-Class Quiz 1.3

- Consider the rep of CharSet in Example 3:

```
public class CharSet {  
    private String s;  
    // Rep invariant:  
    //   s.length() is even  
    //   s[0] <= s[1] <= ... <= s[s.length()-1]  
    // Abstraction function:  
    //   represents the union of the ranges  
    //   {s[i]...s[i+1]} for each adjacent pair  
    //   of characters in s  
    ...  
}
```



- Which of these values does the abstraction function map to the same abstract value as it maps "tv"?
- |                                     |                                 |
|-------------------------------------|---------------------------------|
| <input type="checkbox"/> "ttv"      | <input type="checkbox"/> "ttuv" |
| <input type="checkbox"/> "tluuuvvv" | <input type="checkbox"/> "tuv"  |

# Choosing Rep Invariant and Abstraction Function

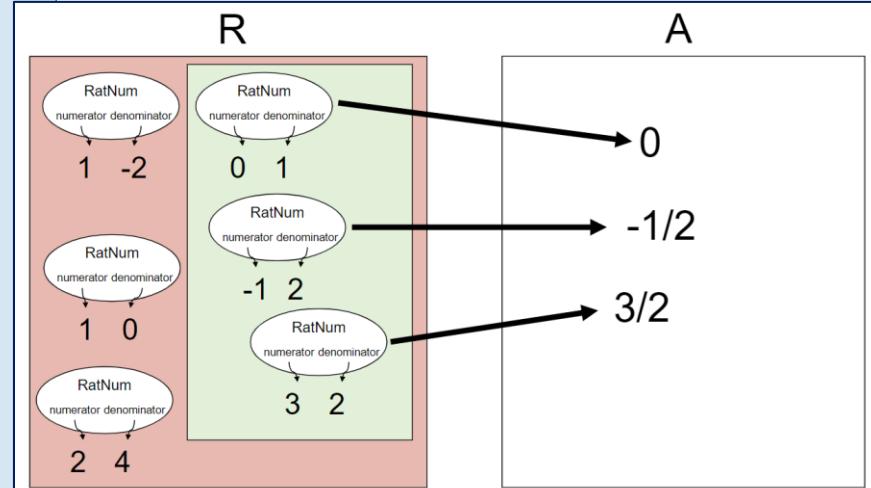
---

- The essential point is that designing an abstract type means **not only choosing the two spaces** — the abstract value space for the specification and the rep value space for the implementation — **but also deciding which rep values are legal and how to interpret them as abstract values**
- It's critically important to write down these assumptions in your code, as we've done above, so that future programmers (and your future self) are aware of what the representation actually means

# Example: Rational Numbers (1)

- Here's an example of an ADT for rational numbers with rep invariant and abstraction function

```
public class RatNum {  
    private final int numerator;  
    private final int denominator;  
    // Rep invariant:  
    // denom > 0  
    // numer/denom is in reduced form  
    // Abstraction Function:  
    // represents rational nbr numer/denom  
    /**  
     * Make a new Ratnum == n.  
     * @param n value  
     */  
    public RatNum(int n) {  
        numerator = n;  
        denominator = 1;  
        checkRep();  
    }  
}
```

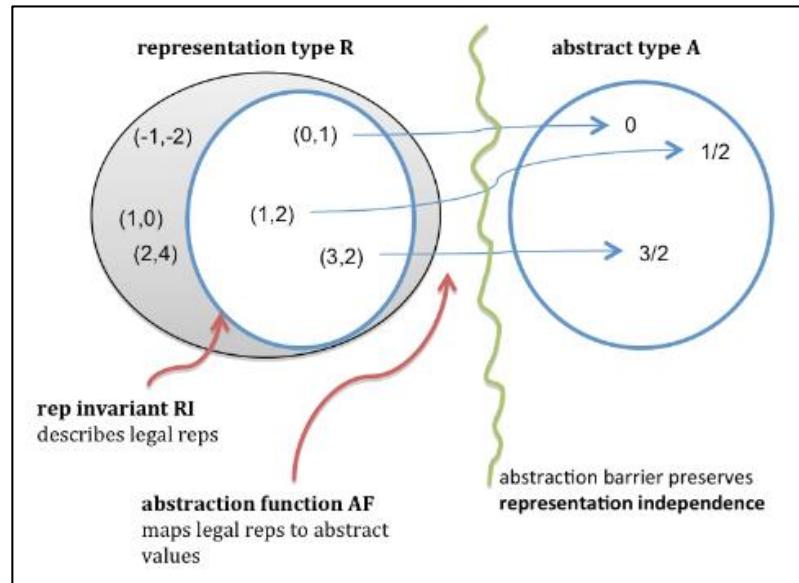


## Example: Rational Numbers (2)

```
/*
 * Make a new RatNum == (n / d).
 * @param n numerator
 * @param d denominator
 * @throws ArithmeticException if d == 0
 */
public RatNum(int n, int d) throws ArithmeticException {
    // reduce ratio to lowest terms
    int g = gcd(n, d);
    n = n / g;
    d = d / g;
    // make denominator positive
    if (d < 0) {
        numerator = -n;
        denominator = -d;
    } else {
        numerator = n;
        denominator = d;
    }
    checkRep();
}
```

# Example: Rational Numbers (3)

- Here we illustrate the abstraction function and rep invariant for the code
- The RI requires that numerator/denominator pairs be in reduced form (i.e., lowest terms), so pairs like (2,4) and (18,12) should be drawn as outside the RI



# Checking the Rep Invariant (1)

---

- The rep invariant isn't just a neat mathematical idea
  - If your implementation asserts the rep invariant at run time, then you can catch bugs early
- Here's a method for RatNum that tests its rep invariant:

```
// Check that the rep invariant is true
// *** Warning: this does nothing unless you turn on assertion checking
//       by running Java with -enableassertions or -ea
private void checkRep() {
    assert denominator > 0;
    assert gcd(Math.abs(numerator), denominator) == 1;
}
```

## Checking the Rep Invariant (2)

---

- You should certainly call `checkRep()` to assert the rep invariant at the end of **every** operation that creates or mutates the rep — in other words, creators, producers, and mutators
  - Look back at the `RatNum` code above, and you'll see that it calls `checkRep()` at the end of **both** constructors
- Observer methods don't normally need to call `checkRep()`, but it's good defensive practice to do so anyway
  - Why? Calling `checkRep()` in every method, including observers, means you'll be more likely to catch rep invariant violations caused by rep exposure

Why is `checkRep` private?

Who should be responsible for checking and enforcing a rep invariant — clients, or the implementation itself?

# No Null Values in the Rep (1)

---

- Recall from the Lecture 8 that null values are troublesome and unsafe, so much so that we try to remove them from our programming entirely
  - in this course, the preconditions and postconditions of our methods implicitly require that objects and arrays be non-null
- We **extend** that prohibition to the reps of abstract data types
  - by default, the rep invariant implicitly includes  $x \neq \text{null}$  for every reference  $x$  in the rep that has object type (including references inside arrays or lists)
  - so if your rep is:

```
class CharSet {  
    String s;  
}
```
  - then its rep invariant automatically includes  $s \neq \text{null}$ , and you don't need to state it in a rep invariant comment

## No Null Values in the Rep (2)

---

- When it's time to implement that rep invariant in a checkRep() method, however, you still must implement the `s != null` check, and make sure that your checkRep() correctly fails when `s` is null
- Often that check comes for free from Java, because checking other parts of your rep invariant will throw an exception if `s` is null
  - For example, if your checkRep() looks like this:

```
private void checkRep() {  
    assert s.length() % 2 == 0;  
    ...  
}
```

- then you **don't** need `assert s != null`, because the call to `s.length()` **will fail** just as effectively on a null reference
- But if `s` is not otherwise checked by your rep invariant, then `assert s != null` explicitly

# Documenting the AF, RI, and Safety from Rep Exposure (1)

---

- It's good practice to document the abstraction function and rep invariant in the class, using comments right where the private fields of the rep are declared
  - we've been doing that above
- Another piece of documentation that you need to write is **a rep exposure safety argument**
  - this is a comment that examines each part of the rep, looks at the code that handles that part of the rep (particularly with respect to parameters and return values from clients, because that is where rep exposure occurs), and presents a reason why the code doesn't expose the rep

# Documenting the AF, RI, and Safety from Rep Exposure (2)

- Here's an example of Tweet with its rep invariant, abstraction function, and safety from rep exposure fully documented:

```
// Immutable type representing a tweet.
public class Tweet {

    private final String author;
    private final String text;
    private final Date timestamp;

    // Rep invariant:
    //   author is a Twitter username (a nonempty string of letters, digits, underscores)
    //   text.length <= 280
    // Abstraction Function:
    //   represents a tweet posted by author, with content text, at time timestamp
    // Safety from rep exposure:
    //   All fields are private;
    //   author and text are Strings, so are guaranteed immutable;
    //   timestamp is a mutable Date, so Tweet() constructor and getTimestamp()
    //       make defensive copies to avoid sharing the rep's Date object with clients.
```

## Documenting the AF, RI, and Safety from Rep Exposure (3)

---

```
public Tweet(String author, String text, Date timestamp) { ... }

public String getAuthor() { ... }

public String getText() { ... }

public Date getTimestamp() { ... }

}
```

- Notice that we don't have any explicit rep invariant conditions on timestamp (aside from the conventional assumption that `timestamp != null`, which we have for all object references)
- But we still need to include timestamp in the rep exposure safety argument, because the immutability property of the whole type depends on all the fields remaining unchanged

# Documenting the AF, RI, and Safety from Rep Exposure (4)

- Compare the argument above with an example of a broken argument involving mutable Date objects:

```
public class Timespan {  
  
    private final Date start;  
    private final Date end;  
  
    // Rep invariant:  
    // !end.before(start)  
    // Abstraction Function:  
    // represents the time interval from start to end, inclusive  
    // Safety from rep exposure:  
    // All fields are private and immutable. (<== oops, false! Date is mutable)  
  
    ...  
}
```

# Documenting the AF, RI, and Safety from Rep Exposure (5)

- Here are the arguments for RatNum

```
// Immutable type representing a rational number.  
public class RatNum {  
    private final int numer;  
    private final int denom;  
  
    // Rep invariant:  
    //   denom > 0  
    //   numer/denom is in reduced form, i.e. gcd(|numer|,denom) = 1  
    // Abstraction Function:  
    //   represents the rational number numer / denom  
    // Safety from rep exposure:  
    //   All fields are private, and all types in the rep are immutable.  
  
    public RatNum(int n) { ... }  
    public RatNum(int n, int d) throws ArithmeticException { ... }  
    ...  
}
```

- Notice that an immutable rep is particularly easy to argue for safety from rep exposure

# How to Establish Invariants (1)

---

- An invariant is a property that is true for the entire program — which in the case of an invariant about an object, reduces to the entire lifetime of the object
- To make an invariant hold, we need to:
  - **make the invariant true in the initial state of the object;** and
  - **ensure that all changes to the object keep the invariant true**
- Translating this in terms of the types of ADT operations, this means:
  - creators and producers must ***establish the invariant*** for new object instances; and
  - mutators and observers must ***preserve the invariant***

## How to Establish Invariants (2)

---

- The risk of rep exposure makes the situation more complicated;  
If the rep is exposed, then the object might be changed anywhere in the program, not just in the ADT's operations, and we can't guarantee that the invariant still holds after those arbitrary changes

So the full rule for proving invariants is: (called **structural induction** )

- if an invariant of an abstract data type is
  - established by creators and producers;
  - preserved by mutators, and observers; and
  - no representation exposure occurs,

then the invariant is true of all instances of the abstract data type

# ADT invariants replace preconditions (1)

---

- Now let's bring a lot of pieces together
- An enormous advantage of a well-designed abstract data type is that it encapsulates and enforces properties that we would otherwise have to stipulate in a precondition
- For example, instead of a spec like this, with an elaborate precondition:

```
/**  
 * @param set1 is a sorted set of characters with no repeats  
 * @param set2 is likewise  
 * @return characters that appear in one set but not the other,  
 *         in sorted order with no repeats  
 */  
static String exclusiveOr(String set1, String set2);
```

## ADT invariants replace preconditions (2)

---

- We can instead use an ADT that captures the desired property:

```
/** @return characters that appear in one set but not the other */
static SortedSet<Character> exclusiveOr(SortedSet<Character> set1,
   SortedSet<Character> set2);
```

- This is easier to understand, because the *name of the ADT* conveys all the programmer needs to know
- It's also safer from bugs, because Java static checking comes into play, and the required condition (sorted with no repeats) can be enforced in exactly one place, the SortedSet type
- Many of the places where we used preconditions on the problem sets would have benefited from a custom ADT instead

# Inheritance, DMS, Comparability, Equality

---

- For the second part of the lecture, we will continue our discussion last week on
  - Inheritance
  - Dynamic Method Selection
    - also known as Polymorphism
  - Comparability
  - Equality

# Recall from last week: Which Method Selected

- Recall that if X is a superclass of Y, then an X variable can hold a reference to a Y

```
public static void main(String[] args) {  
    ListIF<String> list2 = new SLLList<String>();  
    list2.addLast("abc");  
    list2.print();  
}
```

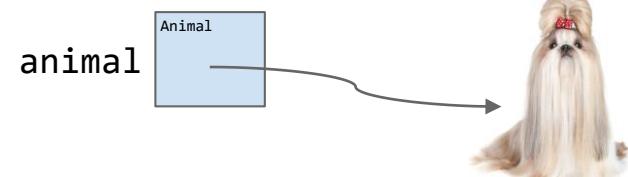
- Which print method will run when the code above executes?
  - SLLList.print(), and not ListIF.print()
  - How does it work?
    - Before we can answer, we need to understand static and dynamic type

# Recall from last week: Static Type vs Dynamic Type

- Every variable in Java has a ***compile-time*** type = **static** type
  - This is the type specified at declaration and it *never changes!*
- Variables also have a ***run-time*** type = **dynamic** type
  - This is the type specified *at instantiation* (e.g. when using new)
  - Equal to the type of the object *being* pointed at

very important  
to remember !

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



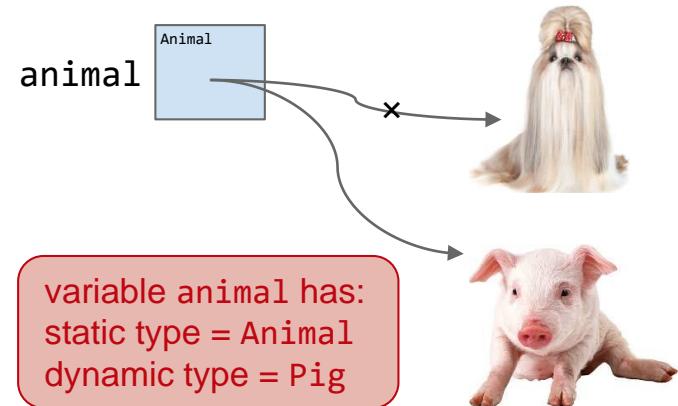
variable animal has:  
static type = Animal  
dynamic type = Dog

# Recall from last week: Static Type vs Dynamic Type

- Every variable in Java has a ***compile-time*** type = **static** type
  - This is the type specified at declaration and it *never changes!*
- Variables also have a ***run-time*** type = **dynamic** type
  - This is the type specified *at instantiation* (e.g. when using new)
  - Equal to the type of the object *being* pointed at

very important  
to remember !

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



# Recall from last week: DMS for Overridden Methods

---

- Suppose we call a method `m()` of an object using a variable with:
  - static type X and dynamic type Y
- First, the compiler records the X's method `m()`, to be used in run-time
- At run-time, if Y overrides the method `m()`, then Y's method `m()` is used *instead*
  - this is known as **dynamic method selection**

# Recall from last week: DMS for Print()

- Suppose we call a method `m()` of an object using a variable with:
  - static type `X` and dynamic type `Y`
- First, the compiler records the `X`'s method `m()`, to be used in run-time
- At run-time, if `Y` overrides the method `m()`, then `Y`'s method `m()` is used *instead*
  - this is known as **dynamic method selection**
- Therefore, `print()` that belongs to `SLList` is used:

```
public static void main(String[] args) {  
    ListIF<String> list2 = new SLList<String>();  
    list2.addLast("abc");  
    list2.print();  
}
```

# Dynamic Method Selection for Overridden Methods (1)

- Suppose we call a method `m()` of an object using a variable with:

- static type `X` and dynamic type `Y`

`Y` is a subclass of `X`

- that is: `X varName = new Y();`  
`varName.m();`

if `X` does **not** define method `m()`,  
(even if `Y` defines `m()`)  
it will cause a compile error

- First, the compiler records the `X`'s method `m()`, to be used in run-time
- At run-time, **if** `Y` overrides the method `m()`, **then** `Y`'s method `m()` is used *instead*
  - this is known as **dynamic method selection**

if `Y` does **not** override `m()`,  
(even if `Y` overload `m()`)  
`X.m()` will be used

# Dynamic Method Selection for Overridden Methods (2)

- Suppose we call a method `m()` of an object using a variable with:
  - static type X and dynamic type Y
  - that is: `X varName = new Y();`  
`varName.m();`
- First, the compiler records the X's method `m()`, to be used in run-time
- At run-time, if Y overrides the method `m()`, then Y's method `m()` is used *instead*
  - this is known as **dynamic method selection**

Thus, DMS happens if:  
1. X defines `m()`  
2. Y overrides `m()`

## Another Example

- Recall from last week, that RecoSLList is a subclass of SLList
  - which line causes error, and if not, whose method is called?

```
public static void main(String[] args) {  
    RecoSLList<Integer> rsl =  
        new RecoSLList<Integer>(5);  
    SLList<Integer> sl = rsl;  
  
    sl.addLast(10);  
    sl.delLast();  
  
    sl.printDelItems();  
    RecoSLList<Integer> rsl2 = sl;  
}
```

variable rsl has static type RecoSLList  
and dynamic type RecoSLList

variable sl has static type SLList  
and dynamic type RecoSLList,  
allowed because RecoSLList is  
a subclass of SLList

## Another Example

- Recall from last week, that RecoSLList is a subclass of SLList
  - which line causes error, and if not, whose method is called?

```
public static void main(String[] args) {  
    RecoSLList<Integer> rsl =  
        new RecoSLList<Integer>(5);  
    SLList<Integer> sl = rsl;  
  
    sl.addLast(10); ←  
    sl.delLast();  
  
    sl.printDelItems();  
    RecoSLList<Integer> rsl2 = sl;  
}
```

rsl static type: RecoSLList  
dynamic type: RecoSLList

sl static type: SLList  
dynamic type: RecoSLList

RecoSLList does not override  
addLast, use SLList.addLast

## Another Example

- Recall from last week, that RecoSLList is a subclass of SLList
  - which line causes error, and if not, whose method is called?

```
public static void main(String[] args) {  
    RecoSLList<Integer> rsl =  
        new RecoSLList<Integer>(5);  
    SLList<Integer> sl = rsl;  
  
    sl.addLast(10);  
    sl.delLast();  
  
    sl.printDelItems();  
    RecoSLList<Integer> rsl2 = sl;  
}
```

rsl static type: RecoSLList  
dynamic type: RecoSLList

sl static type: SLList  
dynamic type: RecoSLList

RecoSLList overrides delLast,  
use RecoSLList.delLast

# Another Example

- Recall from last week, that RecoSLList is a subclass of SLList
  - which line causes error, and if not, whose method is called?

```
public static void main(String[] args) {  
    RecoSLList<Integer> rsl =  
        new RecoSLList<Integer>(5);  
    SLList<Integer> sl = rsl;  
  
    sl.addLast(10);  
    sl.delLast();  
  
    sl.printDelItems();  
    RecoSLList<Integer> rsl2 = sl;  
}
```

rsl static type: RecoSLList  
dynamic type: RecoSLList

sl static type: SLList  
dynamic type: RecoSLList

compiler first checks that sl  
static type SLList does **not**  
have printDelItems method,  
this will cause a **compile error!**

even though its dynamic type  
RecoSLList has printDelItems

# Another Example

- Recall from last week, that RecoSLList is a subclass of SLList
  - which line causes error, and if not, whose method is called?

```
public static void main(String[] args) {  
    RecoSLList<Integer> rsl =  
        new RecoSLList<Integer>(5);  
    SLList<Integer> sl = rsl;  
  
    sl.addLast(10);  
    sl.delLast();  
  
    sl.printDelItems();  
    RecoSLList<Integer> rsl2 = sl;  
}
```

rsl static type: RecoSLList  
dynamic type: RecoSLList

sl static type: SLList  
dynamic type: RecoSLList

once again, compiler checks  
based on static type of sl, and  
because its static type is  
SLList which is **not** a subclass  
of RecoSLList, this will also  
cause a **compile error!**

compiler plays safe by allowing  
assignment based on static type

# Expressions have Static Types



- Method calls have compile-time/static type equal to their declared type
  - for example, we have `ShihTzu` a subclass of `Dog`, `Dog` has a name and a weight, and the method `public static Dog largerDog(Dog d1, Dog d2)` returns a `Dog` that has a larger weight
  - we call the method on two `ShihTzus`:

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

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

# Expressions have Static Types



- Method calls have compile-time/static type equal to their declared type
  - for example, we have ShihTzu a subclass of Dog, Dog has a name and a weight, and the method public static Dog largerDog(Dog d1, Dog d2) returns a Dog that has a larger weight
  - we call the method on two ShihTzus:

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

the static type of largerDog is Dog,  
just as declared

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

# Expressions have Static Types



- Method calls have compile-time/static type equal to their declared type
  - for example, we have `ShihTzu` a subclass of `Dog`, `Dog` has a name and a weight, and the method `public static Dog largerDog(Dog d1, Dog d2)` returns a `Dog` that has a larger weight
  - we call the method on two `ShihTzus`:

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

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

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

# Casting

- To specify the static type of any expression, we use **casting**
  - largerDog(baobei, jiaozi) has static type Dog
  - (ShihTzu) largerDog(baobei, jiaozi) has static type ShihTzu

we've used it in ARDeque

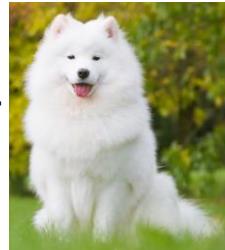
tell compiler the type of the expression

since now the static type of the expression  
is ShihTzu, the code compiles

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

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

# Casting is Dangerous



- Casting tells the compiler to *ignore* its type checking duties
  - for example, suppose you also have Samoyed, another subclass of Dog
  - and say its object has a larger weight
- The result of the method at runtime still has type Samoyed!
  - Casting does **not** change it!

```
ShihTzu baobei = new ShihTzu("Baobei", 5);  
Samoyed xiaxue = new Samoyed("Xiaxue", 20);
```

the compiler is told to pretend to see  
the expression has type ShihTzu,  
so it will **allow this at compile-time**

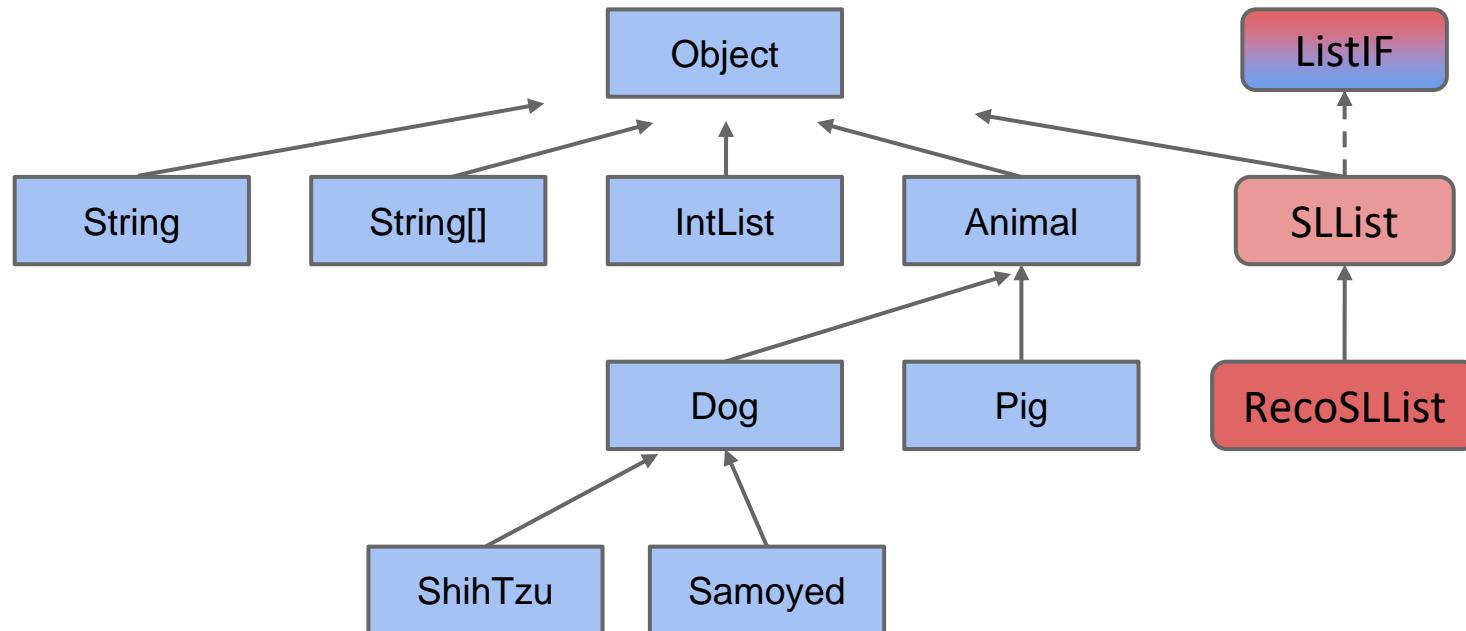
```
ShihTzu largerShihTzu = (ShihTzu)largerDog(baobei, xiaxue);
```

when you run the code, **at runtime** you will get  
**ClassCastException**, and effectively break the  
type checking we discussed in earlier weeks

use Casting with great care!

# The Object Class (1)

- Every type in Java is a subclass of the Object class
  - SLLList implicitly extends Object



# The Object Class (2)

- Object class is a concrete class (as opposed to abstract class)
  - it has implementations, which means your class actually inherits:

| Modifier and Type           | Method                                      | Description                                                                                                                                                            |
|-----------------------------|---------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| protected                   | <code>Object clone()</code>                 | Creates and returns a copy of this object.                                                                                                                             |
| boolean                     | <code>equals (Object obj)</code>            | Indicates whether some other object is "equal to" this one.                                                                                                            |
| protected void              | <code>finalize()</code>                     | <b>Deprecated.</b><br>The finalization mechanism is inherently problematic.                                                                                            |
| <code>Class&lt;?&gt;</code> | <code>getClass()</code>                     | Returns the runtime class of this Object.                                                                                                                              |
| int                         | <code>hashCode()</code>                     | Returns a hash code value for the object.                                                                                                                              |
| void                        | <code>notify()</code>                       | Wakes up a single thread that is waiting on this object's monitor.                                                                                                     |
| void                        | <code>notifyAll()</code>                    | Wakes up all threads that are waiting on this object's monitor.                                                                                                        |
| <code>String</code>         | <code>toString()</code>                     | Returns a string representation of the object.                                                                                                                         |
| void                        | <code>wait()</code>                         | Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> .                                                     |
| void                        | <code>wait (long timeout)</code>            | Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed. |
| void                        | <code>wait (long timeout, int nanos)</code> | Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed. |

we will be using

## Another Example (1)

---

- Suppose Dog has a method bark() that is overridden by ShihTzu
  - For each bark(), is there an error, or if not, which bark() is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

((Dog) obj1).bark();

Object obj2 = (Dog) obj1;
obj2.bark();
```

## Another Example (2)

- Suppose Dog has a method bark() that is overridden by ShihTzu
  - For each bark(), is there an error, or if not, which bark() is called?

```
Object obj1 = new ShihTzu("Baobei", 5);
```

ShihTzu is an Object, so this is allowed

```
ShihTzu s1 = (ShihTzu) obj1;  
s1.bark();
```

Object is cast into a ShihTzu, and then  
assigned to a ShihTzu -> allowed

```
Dog d1 = (Dog) obj1;  
d1.bark();
```

ShihTzu.bark() is called

```
((Dog) obj1).bark();
```

```
Object obj2 = (Dog) obj1;  
obj2.bark();
```

## Another Example (3)

- Suppose Dog has a method bark() that is overridden by ShihTzu
  - For each bark(), is there an error, or if not, which bark() is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1; ←
d1.bark(); ←

((Dog) obj1).bark();

Object obj2 = (Dog) obj1;
obj2.bark();
```

Object is cast into a Dog, and  
then assigned to a Dog -> allowed

1. Check if static type Dog has bark: yes  
2. Dynamic type ShihTzu override it: yes  
-> ShihTzu.bark() is called

## Another Example (4)

- Suppose Dog has a method bark() that is overridden by ShihTzu
  - For each bark(), is there an error, or if not, which bark() is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

((Dog) obj1).bark(); ←

Object obj2 = (Dog) obj1;
obj2.bark();
```

The same as the previous line  
1. Check if static type Dog has bark: yes  
2. Dynamic type ShihTzu override it: yes  
-> ShihTzu.bark() is called

Casting does **not** have lasting effect,  
it does **not** change the static type of obj1,  
it only enforce a static type within the (...)

## Another Example (5)

- Suppose Dog has a method bark() that is overridden by ShihTzu
  - For each bark(), is there an error, or if not, which bark() is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

((Dog) obj1).bark();

Object obj2 = (Dog) obj1; ←
obj2.bark();
```

because casting, the static type  
of right hand side is Dog,  
next, is a Dog an Object? yes  
-> allowed

## Another Example (6)

- Suppose Dog has a method bark() that is overridden by ShihTzu
  - For each bark(), is there an error, or if not, which bark() is called?

```
Object obj1 = new ShihTzu("Baobei", 5);

ShihTzu s1 = (ShihTzu) obj1;
s1.bark();

Dog d1 = (Dog) obj1;
d1.bark();

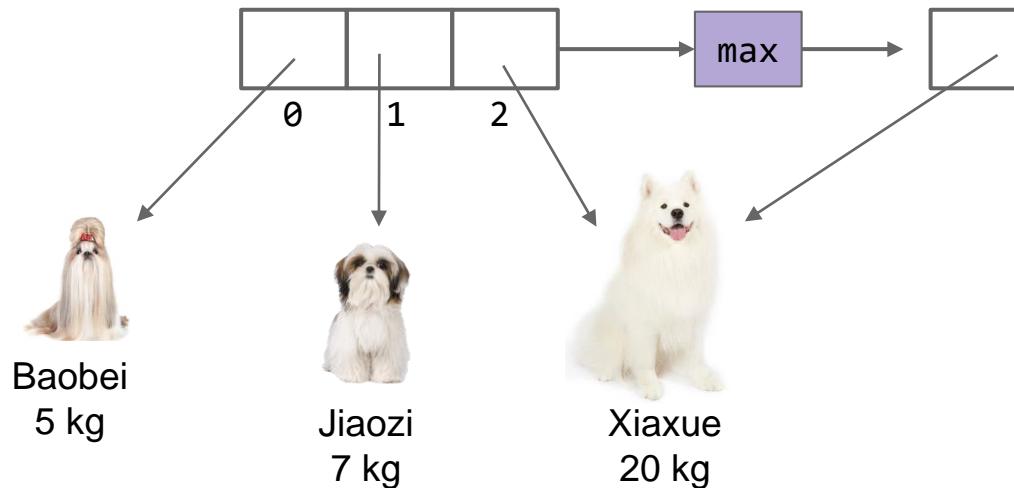
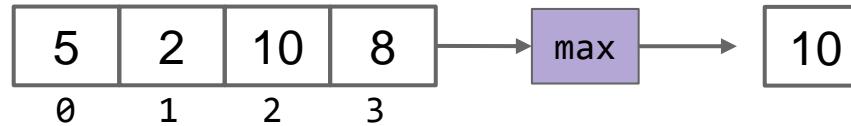
((Dog) obj1).bark();

Object obj2 = (Dog) obj1;
obj2.bark(); ←
```

the static type of obj2 is Object  
does an Object define a bark()  
method? no  
-> compile error

# Max Function

- Suppose we want to write a method `max()` that returns the max of *any* array, regardless of the type



# Max Function

- Suppose we want to write a method `max()` that returns the max of *any* array, regardless of the type

```
public static Object max(Object[] items) {  
    int maxIndex = 0;  
    for (int i = 0; i < items.length; i += 1) {  
        if (items[i] > items[maxIndex]) {  
            maxIndex= i;  
        }  
    }  
    return items[maxIndex];  
}
```

say your friend wrote this code.  
there is a bug in the code.  
can you find where the bug is?

```
public static void main(String[] args) {  
    Dog[] dogs = {new Dog("Baobei", 5), new Dog("Jiaozi", 7),  
                 new Dog("Xiaxue", 20});  
    Dog maxDog = (Dog) StatLib.max(dogs);  
    maxDog.bark();  
}
```

# In-Class Quiz 2

- Suppose we want to write a method `max()` that returns the max of *any* array, regardless of the type

```
public static Object max(Object[] items) {  
    int maxIndex = 0;  
    for (int i = 0; i < items.length; i += 1) {  
        if (items[i] > items[maxIndex]) {  
            maxIndex= i;  
        }  
    }  
    return items[maxIndex];  
}
```

say your friend wrote this code.  
there is a bug in the code.  
can you find where the bug is?

1

2

3

```
public static void main(String[] args) {  
    Dog[] dogs = {new Dog("Baobei", 5), new Dog("Jiaozi", 7),  
                 new Dog("Xiaxue", 20)};  
    Dog maxDog = (Dog) StatLib.max(dogs);  
    maxDog.bark();  
}
```

# Max Function Bug (1)

- Suppose we want to write a method `max()` that returns the max of *any* array, regardless of the type

```
public static Object max(Object[] items) {  
    int maxIndex = 0;  
    for (int i = 0; i < items.length; i += 1) {  
        if (items[i] > items[maxIndex]) {  
            maxIndex= i;  
        }  
    }  
    return items[maxIndex];  
}
```

we pass an array of Dog here.  
because a Dog is an Object,  
an array of Dog is also an  
array of Object.  
**no bug here.**

```
public static void main(String[] args) {  
    Dog[] dogs = {new Dog("Baobei", 5), new Dog("Jiaozi", 7),  
                 new Dog("Xiaxue", 20)};  
    Dog maxDog = (Dog) StatLib.max(dogs);  
    maxDog.bark();  
}
```

this is a good use of casting.  
**no bug here.**

# Max Function Bug (2)

- Suppose we want to write a method `max()` that returns the max of *any* array, regardless of the type

```
public static Object max(Object[] items) {  
    int maxIndex = 0;  
    for (int i = 0; i < items.length; i += 1) {  
        if (items[i] > items[maxIndex]) {  
            maxIndex= i;  
        }  
    }  
    return items[maxIndex];  
}
```

the `>` operator does not work with arbitrary object types.  
this **is** a bug.

```
public static void main(String[] args) {  
    Dog[] dogs = {new Dog("Baobei", 5), new Dog("Jiaozi", 7),  
                 new Dog("Xiaxue", 20});  
    Dog maxDog = (Dog) StatLib.max(dogs);  
    maxDog.bark();  
}
```

what should we do if we want to compare and find max dog?

# Max Dog

---

- In order to make the `max()` method work on your `Dog` class:
  - Step 1: make your `Dog` class comparable by implementing the `Comparable<T>` interface

```
public interface Comparable<T> {  
    public int compareTo(T other);  
}
```

- Step 2: make your `max` method call `compareTo` to compare the `Comparable` objects in an array of `Comparables`

# Step 1: Comparable Dog

- To make your Dog class comparable:
  - Declare the class implementing the Comparable<T> interface

```
public class Dog implements Comparable<Dog> {
```

- Implement the method compareTo

```
@Override  
public int compareTo(Dog other) {  
  
    if (this.weight < other.weight) {  
        return -1;  
    } else if (this.weight == other.weight) {  
        return 0;  
    }  
    return 1;  
}
```

we compare the dogs based on their weights  
return -1 if smaller than other dog  
return 0 if the same  
return 1 if larger

# Step 1: Comparable Dog 2

---

- To make your Dog class comparable:
  - Declare the class implementing the Comparable<T> interface

```
public class Dog implements Comparable<Dog> {
```

- Implement the method compareTo

```
@Override  
public int compareTo(Dog other) {  
  
    return this.weight - other.weight;  
}
```

alternatively,  
return negative if smaller than other dog  
return 0 if the same  
return positive if larger

## Step 2: Max Comparables

- Make your max method call compareTo to compare the Comparable objects in an array of Comparables

```
public static Comparable max(Comparable[] items) {  
    int maxIndex = 0;  
    int cmp;  
    for (int i = 0; i < items.length; i++) {  
  
        cmp = items[i].compareTo(items[maxIndex]);  
  
        if (cmp > 0) {  
            maxIndex = i;  
        }  
    }  
    return items[maxIndex];  
}
```

compareTo will return a positive number  
if items[maxIndex] is larger

now you can run your max method  
on any Comparable objects!  
this is called **Generic Methods**

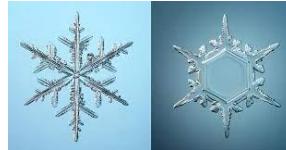
# Equality: Introduction (1)

---

- Now we will discuss about equality
- In the first part of the lecture, we have developed a rigorous notion of data abstraction by creating types that are characterized by their operations, not by their representation
  - For an abstract data type, the abstraction function explains how to *interpret a concrete representation value as a value of the abstract type*, and we saw how the choice of abstraction function determines how to write the code implementing each of the ADT's operations
- In this part we turn to how we define the notion of **equality of values in a data type**: the abstraction function will give us a way to cleanly define the equality operation on an ADT

# Equality: Introduction (2)

---

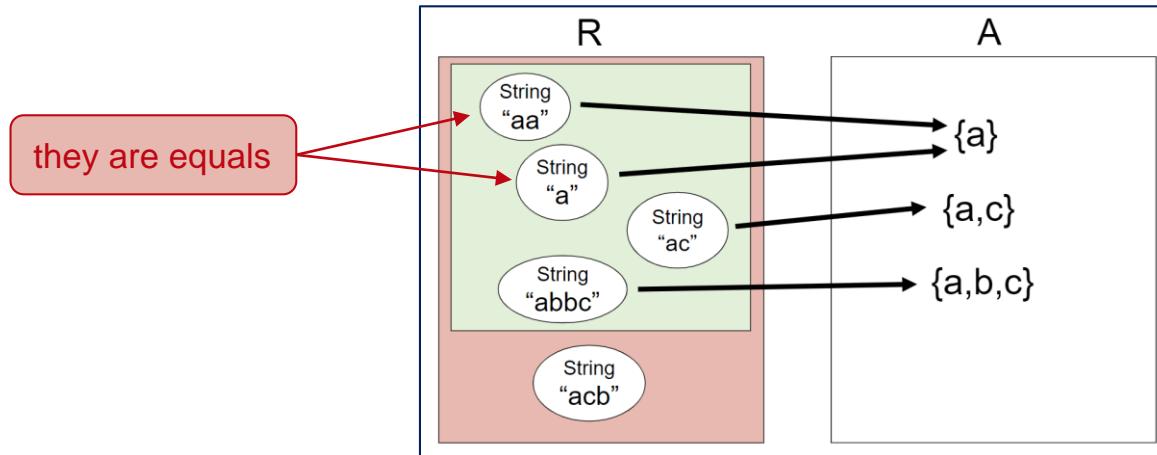


- In the *physical world*, every object is distinct — at some level, even two snowflakes are different, even if the distinction is just the position they occupy in space
  - This isn't strictly true of all subatomic particles, but true enough of large objects like snowflakes, baseballs and people
  - So two physical objects are never truly “equal” to each other; they only have degrees of similarity
- In the world of *human language*, however, and in the world of *mathematical concepts*, you can have multiple names for the same thing
  - so it's natural to ask when two expressions represent the same thing:  $1+2$ ,  $\sqrt{9}$ , and  $3$  are alternative expressions for the same ideal mathematical value

# Three Ways to Regard Equality (1)

Formally, we can regard equality in several ways

- **Using an abstraction function.** Recall that an abstraction function  $AF: R \rightarrow A$  maps concrete instances of a data type to their corresponding abstract values
  - To use AF as a definition for equality, we would say that  $a$  equals  $b$  if and only if  $AF(a) = AF(b)$
  - For example, when we use strings that allows duplicates and at the same time require that the characters of that strings to be sorted, to represent sets of characters



## Three Ways to Regard Equality (2)

---

- **Using a relation.** An equivalence is a relation  $E \subseteq T \times T$  that is:
  - reflexive:  $E(t,t) \forall t \in T$
  - symmetric:  $E(t,u) \Rightarrow E(u,t)$
  - transitive:  $E(t,u) \wedge E(u,v) \Rightarrow E(t,v)$

To use  $E$  as a definition for equality, we would say that  $a$  equals  $b$  if and only if  $E(a,b)$

These two notions are ***equivalent***:

- An equivalence relation induces an abstraction function (the relation partitions  $T$ , so  $f$  maps each element to its partition class)
- The relation induced by an abstraction function is an equivalence relation (check for yourself that the three properties hold)

## Three Ways to Regard Equality (2)

---

A third way we can talk about the equality between abstract values is in terms of what an outsider (a client) can observe about them

- **Using observation.** We can say that two objects are equal when they *cannot be distinguished by observation* — **every operation** we can apply **produces the same result for both objects**

Consider the set expressions  $\{1,2\}$  and  $\{2,1\}$  — using the observer operations available for sets, cardinality  $| \dots |$  and membership  $\in$ , these expressions are indistinguishable:

- $|\{1,2\}| = 2$  and  $|\{2,1\}| = 2$
- $1 \in \{1,2\}$  is true, and  $1 \in \{2,1\}$  is true
- $2 \in \{1,2\}$  is true, and  $2 \in \{2,1\}$  is true
- $3 \in \{1,2\}$  is false, and  $3 \in \{2,1\}$  is false

In terms of abstract data types, ***observation*** means calling operations on the objects

So **two objects are equal** if and only if **they cannot be distinguished by calling any operations** of the abstract data type

# In-Class Quiz 3

---

- Consider the following rep for an abstract data type:

```
/** Immutable type representing a set of letters, ignoring case */
class LetterSet {
    private String s;
    // Abstraction function:
    //     AF(s) = the set of the letters that are found in s
    //             (ignoring non-letters and alphabetic case)
    // Rep invariant:
    //     true
    /*
     * Make a LetterSet consisting of the letters found in chars
     * (ignoring alphabetic case and non-letters).
     */
    public LetterSet(String chars) {
        s = chars;
    }
    ... // observer and producer operations
}
```

- Using the *abstraction function* definition of equality for LetterSet, which of the following should be considered equal to new LetterSet("abc")?
  - new LetterSet("aBc")
  - new LetterSet("")
  - new LetterSet("bbbbbc")
  - new LetterSet("1a2b3c")

## **== vs equals() (1)**

---

- Like many languages, Java has **two** different operations for testing equality, with different semantics
- The **== operator** compares *references*
  - more precisely, it tests **referential equality**
  - two references are == if they point to *the same storage* in memory
  - in terms of the snapshot diagrams we've been drawing, two references are == if their arrows point to the same object bubble
- The **equals()** operation compares *object contents*
  - in other words, **object equality**, in the sense that we've been talking about in this lecture
  - the equals operation has to be defined appropriately for every abstract data type

## **== vs equals() (2)**

---

- For comparison, here are the equality operators in several languages:

|             | <i>referential<br/>equality</i> | <i>object<br/>equality</i> |
|-------------|---------------------------------|----------------------------|
| Java        | <code>==</code>                 | <code>equals()</code>      |
| Objective C | <code>==</code>                 | <code>isEqual:</code>      |
| C#          | <code>==</code>                 | <code>Equals()</code>      |
| Python      | <code>is</code>                 | <code>==</code>            |
| Javascript  | <code>==</code>                 | n/a                        |

## **== vs equals() (3)**

---

- Note that == unfortunately *flips its meaning* between Java and Python!
  - Don't let that confuse you: == in Java just tests reference identity, it *doesn't* compare object contents
- As programmers in any of these languages, we can't change the meaning of the referential equality operator
  - In Java, == always means *referential equality*
  - But when we define a new data type, it's our responsibility to decide what object equality means for values of the data type, and implement the equals() operation appropriately

# equals in ARSet.contains

- In fact, you have encountered this issue on your ARSet.contains method

```
public boolean contains(T item) {  
    if (item == null) {  
        return false;  
    }  
    for (int i = 0; i < size; i++) {  
        if (items[i].equals(item)) {  
            return true;  
        }  
    }  
    return false;  
}
```

use **equals**, instead of **==**

your set may contain objects with the same content,  
but located at different address in memory

as a consequence, you need to have a special case if  
you allow to add and store null, because equals  
called on null will cause runtime NullPointerException

# An Example

---

- Here's a simple example of an immutable ADT we will be using in discussing equality:

```
public class Duration {  
    private final int mins;  
    private final int secs;  
    // rep invariant:  
    //    mins >= 0, secs >= 0  
    // abstraction function:  
    //    represents a span of time of mins minutes and secs seconds  
    /** Make a duration lasting for m minutes and s seconds. */  
    public Duration(int m, int s) {  
        mins = m; secs = s;  
    }  
    /** @return length of this duration in seconds */  
    public int getLength() {  
        return mins*60 + secs;  
    }  
}
```

# Equality of Immutable Types (1)

---

- As previously mentioned, the `equals()` method is defined by `Object`, and its default implementation looks like this:

```
public class Object {  
    ...  
    public boolean equals(Object that) {  
        return this == that;  
    }  
}
```

- In other words, the **default meaning** of `equals()` is **the same** as referential equality!
- For immutable data types, this is almost always wrong!
  - So you have to **override** the `equals()` method, replacing it with your own implementation

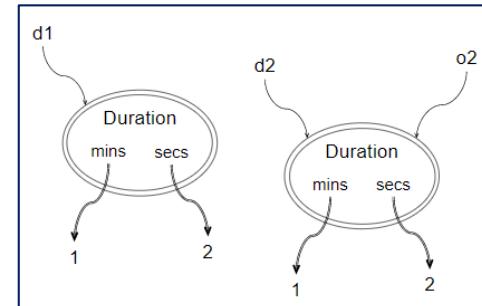
# Equality of Immutable Types (2)

- Here's our first try for Duration:

```
public class Duration {  
    ...  
    // Problematic definition of equals()  
    public boolean equals(Duration that) {  
        return this.getLength() == that.getLength();  
    }  
}
```

- There's a subtle problem here. Why doesn't this work? Let's try this code:

```
Duration d1 = new Duration (1, 2);  
Duration d2 = new Duration (1, 2);  
Object o2 = d2;  
d1.equals(d2) → true  
d1.equals(o2) → false
```



## Equality of Immutable Types (3)

---

- You'll see that even though d2 and o2 end up referring to the very same object in memory, you still get different results for them from equals()
- What's going on?
- It turns out that Duration has **overloaded** the equals() method, because the method signature was **not** identical to Object's !
  - we actually have **two** equals() methods in Duration: an implicit equals(Object) inherited from Object, and the new equals(Duration)

## Equality of Immutable Types (4)

```
public class Duration extends Object {  
    // explicit method that we declared:  
    public boolean equals (Duration that) {  
        return this.getLength() == that.getLength();  
    }  
    // implicit method inherited from Object:  
    public boolean equals (Object that) {  
        return this == that;  
    }  
}
```

equals is overloaded !

## Equality of Immutable Types (5)

---

- Recall that the Java compiler selects between overloaded operations using the compile-time type of the parameters
  - for example, when you use the / operator, the compiler *chooses* either integer division or floating-point division based on whether the arguments are ints or doubles
- The same compile-time selection happens here
  - If we pass an Object reference, as in d1.equals(o2), we end up calling the equals(Object) implementation
  - If we pass a Duration reference, as in d1.equals(d2), we end up calling the equals(Duration) version
  - This happens even though o2 and d2 both point to the same object at runtime!
  - Equality has become inconsistent

## Equality of Immutable Types (6)

---

- It's easy to make a mistake in the method signature, and overload a method when you meant to override it
- Recall last week, that this is such a common error that Java has a language feature, the annotation `@Override`, which you should use whenever your intention is to override a method in your superclass
  - With this annotation, the Java compiler will check that a method with the same signature actually exists in the superclass, and give you a compiler error if you've made a mistake in the signature

## Equality of Immutable Types (7)

- So here's the right way to implement Duration's equals() method:

```
@Override  
public boolean equals (Object that) {  
    if (that == null) return false;  
    if (!(that instanceof Duration)) return false;  
    Duration thatDuration = (Duration) that;  
    return this.getLength() == thatDuration.getLength();  
}
```

- This fixes the problem:

```
Duration d1 = new Duration(1, 2);  
Duration d2 = new Duration(1, 2);  
Object o2 = d2;  
d1.equals(d2) → true  
d1.equals(o2) → true
```

# instanceof

---

- `instanceof` tests whether an object is an instance of a particular type
- Using `instanceof` is dynamic type checking, not the static type checking we prefer
- In our course and most good Java programming: `instanceof` is *disallowed* anywhere **except for implementing equals**
  - this prohibition also includes other ways of inspecting objects' runtime types
  - for example, `getClass()` is also *disallowed*

# Optimization

---

- To optimize the code, no further comparison needed if they are the exact same object:

```
@Override  
public boolean equals (Object that) {  
    if (this == that) return true; located at the same  
place in memory  
    if (that == null) return false;  
    if (!(that instanceof Duration)) return false;  
    Duration thatDuration = (Duration) that;  
    return this.getLength() == thatDuration.getLength();  
}
```

# The Object Contract

---

- The specification of the Object class is so important that it is often referred to as the **Object Contract**
  - The contract can be found in the method specifications for the Object class
  - Here we will focus on the contract for equals
- When you override the equals method, you must adhere to its general contract, that states:
  - equals must define **an equivalence relation** — that is, a relation that is reflexive, symmetric, and transitive
  - equals must be **consistent**: repeated calls to the method must yield the same result provided no information used in equals comparisons on the object is modified
  - for a **non-null** reference x, x.equals(null) should return false
  - **hashCode** must produce the same result for two objects that are deemed equal by the equals method



will talk about this next week

# Breaking the Equivalence Relation (1)

---

- Let's start with the equivalence relation
- We have to make sure that the definition of equality implemented by `equals()` is actually an equivalence relation as defined earlier: reflexive, symmetric, and transitive
- If it isn't, then operations that depend on equality (like sets, searching) will behave erratically and unpredictably
- You don't want to program with a data type in which sometimes `a.equals(b)`, but `b.equals(a)`
  - subtle and painful bugs will result

## Breaking the Equivalence Relation (2)

- Here's an example of how an innocent attempt to make equality more flexible can go wrong
- Suppose we wanted to allow for a tolerance in comparing Duration objects, because different computers may have slightly unsynchronized clocks:

```
private static final int CLOCK_SKEW = 5; // seconds
@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
}
```

- This will violate the equivalence relation

asked in Lecture Quiz 11

# Equality of Mutable Types (1)

---

- We've been focusing on equality of immutable objects; what about mutable objects?
- Equality must still be an equivalence relation, as required by the Object contract
- We also want equality to respect the abstraction function and respect operations
  - But with mutable objects, there is a new possibility: by calling a mutator on one of the objects before doing the observation, we may change its state and thus create an observable difference between the two objects

# Equality of Mutable Types (2)

---

So let's refine our definition and allow for two notions of equality based on observation:

- **Observational equality** means that two references *cannot be distinguished now*, in the current state of the program
  - A client can try to distinguish them *only by calling operations that don't change the state of either object* (i.e. only observers and producers, not mutators) and *comparing the results* of those operations
  - This tests whether the two references “look” the same for the current state of the object
- **Behavioral equality** means that two references cannot be distinguished now or in the future, *even if a mutator is called to change the state of one object but not the other*
  - This tests whether the two references will “behave” the same, in this and all future states

often, we implement behavioral equality with referential equality ==

# Equality of Mutable Types (3)

---

- For immutable objects, observational and behavioral equality are *identical*
  - because there aren't any mutator methods that can change the state of the objects
- For *mutable objects*, it can be very tempting to use observational equality as the design choice, but we *should use behavioral equality*
  - Java uses observational equality for most of its mutable data types, in fact
  - If two distinct List objects contain the same sequence of elements, then equals() reports that they are equal
  - But the presence of mutators unfortunately leads to subtle bugs, because it means that equality isn't consistent over time
  - Two objects that are observationally equal at one moment in the program may stop being equal after a mutation
- We will continue talking about this next week

so, for mutable data structure,  
we sometimes use observational,  
and sometimes we use behavioral equality

# Equality of Mutable Types (4)

---

- Consider these three lists of integers:

```
List<Integer> listA = List.of(1, 2, 3);
List<Integer> listB = new ArrayList<>(listA);
List<Integer> listC = listB;
```

- If we want to explore whether the List values referred to by listA, listB, and listC are equal by **observational equality**, what could we do?
- Which objects are equal by **observational equality**?

# Equality of Mutable Types (5)

---

- Consider these three lists of integers:

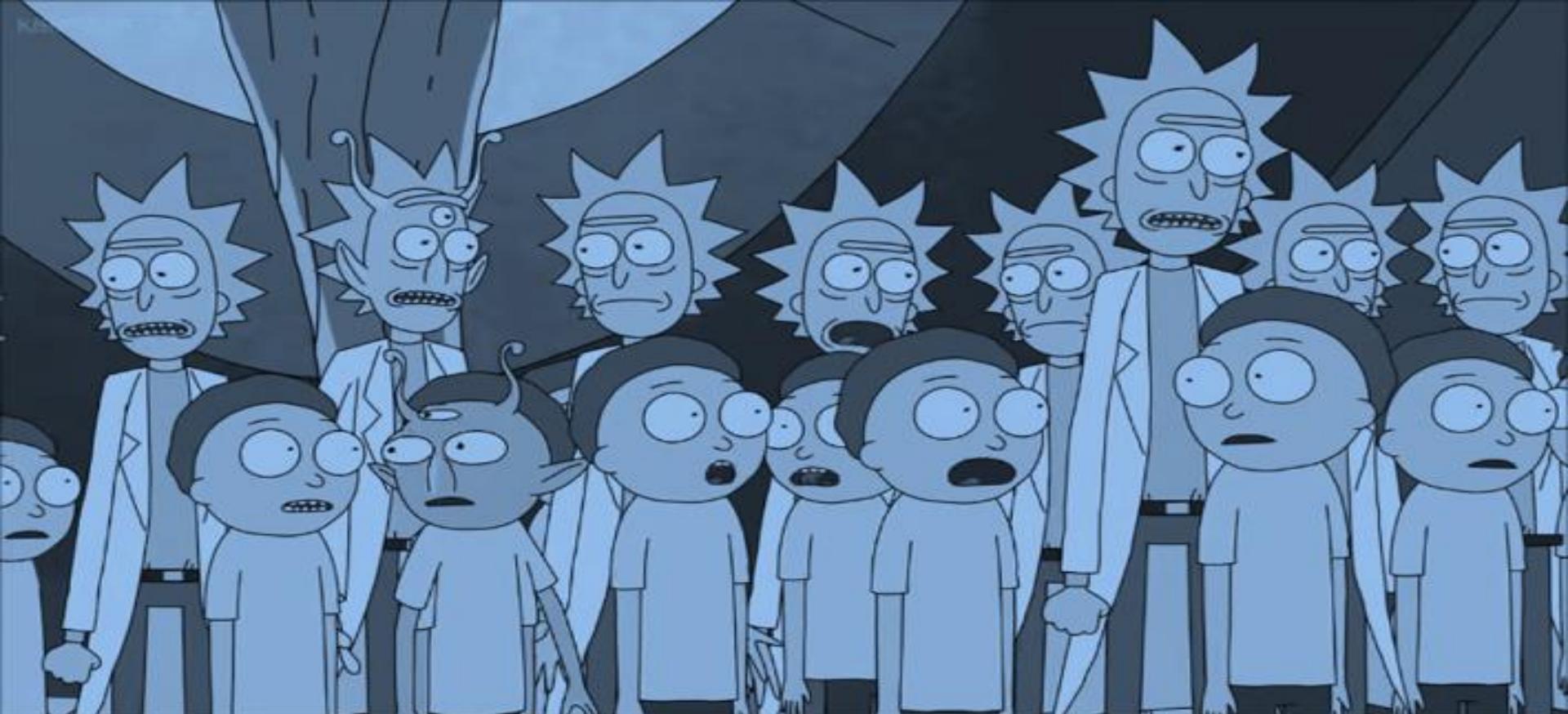
```
List<Integer> listA = List.of(1, 2, 3);
List<Integer> listB = new ArrayList<>(listA);
List<Integer> listC = listB;
```

- If we want to explore whether the List values referred to by listA, listB, and listC are equal by **behavioral equality**, what could we do?
- Which objects are equal by **behavioral equality**?

# Thank you for your attention !

---

- In this lecture, you have learned:
  - about invariant that is always true of an ADT object instance for its lifetime
  - to use abstraction function and to avoid representation exposure
  - to use casting carefully
  - about using dynamic method selection
  - to make a class or a data structure support comparability
  - to differentiate between reference equality and object equality
  - to implement equality correctly for mutable and immutable types
- Please continue to Lecture Quiz 11 and Lab 11:
  - to do Lab Exercise 11.1 - 11.2, and
  - to do Exercise 11.1 - 11.2



# Advanced Object-Oriented Programming

CPT204 – Lecture 12  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming

### Lecture 12

**Hash Table, Comparator,  
Concurrency, Thread Safety**

# Welcome !

---

- Welcome to Lecture 12 !
- In this lecture we are going to
  - learn about hash codes and hash functions
  - learn about a data structure called Hash Table
  - learn about comparator
  - learn about anonymous class
  - learn about concurrency

# Admin Stuff : Important Dates

---

|            |      |    |    |    |    |    |    |    |                          |
|------------|------|----|----|----|----|----|----|----|--------------------------|
| Semester 2 | June | 17 | 18 | 19 | 20 | 21 | 22 | 23 | Week 12                  |
|            |      | 24 | 25 | 26 | 27 | 28 | 29 | 30 | Week 13                  |
|            |      | 31 | 1  | 2  | 3  | 4  | 5  | 6  | Week 14                  |
|            |      | 7  | 8  | 9  | 10 | 11 | 12 | 13 | Examination days;        |
|            |      | 14 | 15 | 16 | 17 | 18 | 19 | 20 | Dragon Boat Day; Marking |

- Tuesday, May 25 Task Sheet released
- Friday, June 4, Part A, B Online Submission  
10:00 – 11:00
- Friday, June 4, Part C Lab Test  
Lab group schedule
- Monday, June 7, Final Exam in Labs  
14:00 – 16:00

# Part 1 : Higher Order Functions, Comparator, Hash Tables

---

- In the first part of the lecture, we will
  - continue our discussion last week on equals and hashCode
  - continue our discussion last week on Comparable
  - introduce its closely related comparison tool called Comparator
  - learn about Higher Order Function technique used to create Comparator
- Importantly, we will also build a hash-based Set
  - and introduce techniques you will use in lab to build hash-based Map

# The Object Contract

- The specification of the Object class is so important that it is often referred to as the **Object Contract**
  - The contract can be found in the method specifications for the Object class
  - Here we will focus on the contract for equals
- When you override the equals method, you must adhere to its general contract, that states:
  - equals must define **an equivalence relation** — that is, a relation that is reflexive, symmetric, and transitive
  - equals must be **consistent**: repeated calls to the method must yield the same result provided no information used in equals comparisons on the object is modified
  - for a **non-null** reference x, x.equals(null) should return false
  - **hashCode** must produce the same result for two objects that are deemed equal by the equals method

will talk about this

# Hash Tables (1)

---

- Last week we discussed about the Object contract
- To understand the part of the contract relating to the **hashCode method**, we now discuss about how hash tables work
  - two very common collection implementations, HashSet and HashMap, use a hash table data structure, and depend on the hashCode method to be implemented correctly for the objects stored in the set and used as keys in the map
- A **hash table** is a representation for a mapping: an abstract data type that maps keys to values
  - hash tables offer constant time lookup, so they tend to perform better than trees or lists
  - keys don't have to be ordered, or have any particular property, except for offering equals and hashCode

## Hash Tables (2)

---

- Here's how a hash table works
  - It contains an array that is initialized to a size corresponding to the number of elements that we expect to be inserted
  - When a key and a value are presented for insertion, we compute **the hash code** of the key, and *convert* it into an index in the array's range (e.g., by a modulo division)
  - The value is then inserted at that index
- The *rep invariant of a hash table* includes the fundamental constraint that a key can be found by starting from the slot determined by its hash code

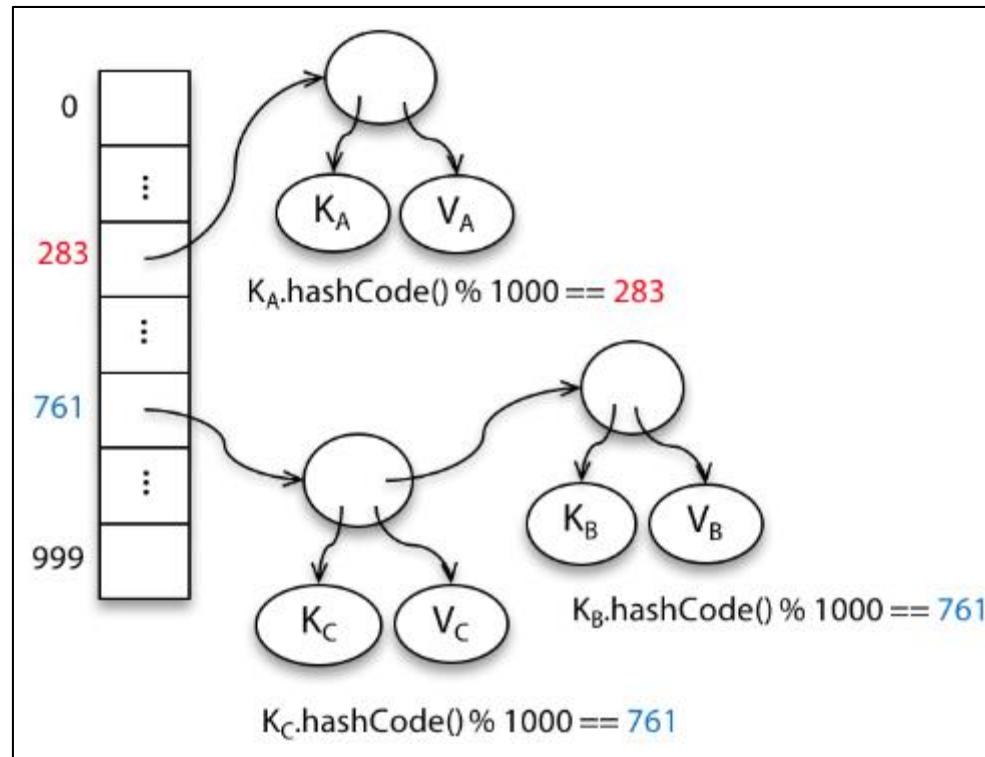
## Hash Tables (3)

---

- Hash codes are designed so that the keys will be spread evenly over the indices
  - But occasionally a conflict occurs, and two keys are placed at the same index
  - So rather than holding a single value at an index, a hash table actually holds *a list of key/value pairs*, usually called a **hash bucket**
- A key/value pair is implemented in Java simply as an object with two fields
  - On *insertion*, you add a pair to the list in the array slot determined by the hash code
  - For *lookup*, you hash the key, find the right slot, and then examine each of the pairs until one is found whose key equals the query key

# Hash Tables (4)

- An example of a hash table:



# Breaking Hash Tables (1)

---

- Now it should be clear why the Object contract requires *equal objects to have the same hash code*
  - if two equal objects had *distinct* hash codes, they might be placed in *different* slots
  - so if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may *fail*
- Object's **default hashCode()** implementation is consistent with its default equals():

```
public class Object {  
    . . .  
    public boolean equals(Object that) { return this == that; }  
    public int hashCode() { return /* the memory address of this */; }  
}
```

- For references a and b, if  $a == b$ , then the address of a == the address of b, so the Object contract **is** satisfied

## Breaking Hash Tables (2)

---

- But immutable objects need a different implementation of hashCode()
- For Duration, since we *haven't* overridden the *default* hashCode() yet, we're currently breaking the Object contract:

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
d1.equals(d2) → true
d1.hashCode() → 2392
d2.hashCode() → 4823
```

- d1 and d2 are equal(), but they have *different* hash codes
  - so we need to fix that

## Breaking Hash Tables (3)

---

- A simple and drastic way to ensure that the contract is met is for hashCode to always return *some constant value*, so every object's hash code is the same
  - this satisfies the Object contract, but it would have a disastrous performance effect, since every key will be stored in the *same* slot, and every lookup will *degenerate* to a linear search along a long list
- The standard way to construct a more reasonable hash code that still satisfies the contract is to compute a hash code for each component of the object that is used in the determination of equality
  - usually by calling the hashCode method of *each component*
  - and then *combining* these, using a few arithmetic operations

## Breaking Hash Tables (4)

---

- For Duration, this is easy because the abstract value of the class is already an integer value:

```
@Override  
public int hashCode() {  
    return getLength();  
}
```

- Our reference textbook, **Effective Java**, explains this issue in more detail, and gives some strategies for writing decent hash code functions and the advice is summarized in a good StackOverflow post:  
<https://stackoverflow.com/questions/113511/best-implementation-for-hashcode-method-for-a-collection>
- Recent versions of Java now have a utility method `Objects.hash()` that makes it easier to implement a hash code involving multiple fields

## Breaking Hash Tables (5)

---

- Note, however, that as long as you satisfy the requirement that equal objects have the same hash code value, then the particular hashing technique you use doesn't make a difference to the correctness of your code
  - it may affect its performance, by creating unnecessary collisions between different objects, but *even* a poorly-performing hash function is better than one that breaks the contract
- Most crucially, note that if you *don't override* hashCode at all, you'll get the one from Object, which is based on the address of the object
  - if you have overridden equals, this will mean that you will have *almost certainly* violated the contract

## Breaking Hash Tables (6)

---

- So as a general rule:
  - **always override hashCode when you override equals**
- It has happened that a student spent hours tracking down a bug in a project that amounted to nothing more than misspelling hashCode as hashcode !
  - this created a new method that didn't override the hashCode method of Object at all, and strange things happened
  - use **@Override** !

# Equality of Mutable Types

Now about mutable objects, recall our definition last week: two objects are equal when they cannot be distinguished by observation

With mutable objects, there are two ways to interpret this:

- when they cannot be distinguished by observation *that doesn't change the state of the objects*, i.e., by calling *only* observer, producer, and creator methods

This is called **observational equality**, since it tests whether the two objects *look* the same in the current state of the program

- when they cannot be distinguished by *any* observation, even state changes

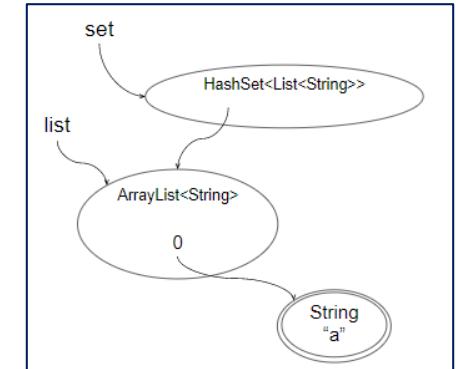
This interpretation allows calling *any* methods on the two objects, including mutators

This is called **behavioral equality**, since it tests whether the two objects will *behave* the same, in this and all future states

# Equality and hashCode Mutable Types Bug (1)

- For immutable objects, observational and behavioral equality are identical, because there aren't any mutator methods
  - For mutable objects, it's tempting to implement strict observational equality
- Java uses **observational equality** for most of its mutable data types, in fact
  - if two distinct List objects contain the same sequence of elements, then equals() reports that they are equal
  - but using observational equality leads to subtle bugs and in fact allows us to easily break the rep invariants of other collection data structures
- Suppose we make a List, and then drop it into a HashSet:

```
List<String> list = new ArrayList<>();
list.add("a");
Set<List<String>> set = new HashSet<List<String>>();
set.add(list);
```



# In-Class Quiz 1

---

- We can check that the set contains the list, and it does:

```
set.contains(list) → true
```

- But now we mutate the list:

```
list.add("goodbye");
```

- What is the output of :

```
set.contains(list) → ?
```

- And, what is the output of:

```
for (List<String> l : set) {  
    set.contains(l) → ?  
}
```

Select one:

- true
- false
- true, true
- false, true
- true, false
- false, false

# Equality and hashCode Mutable Types Bug (2)

- We can check that the set contains the list, and it does:

```
set.contains(list) → true
```

- But now we mutate the list:

```
list.add("goodbye");
```

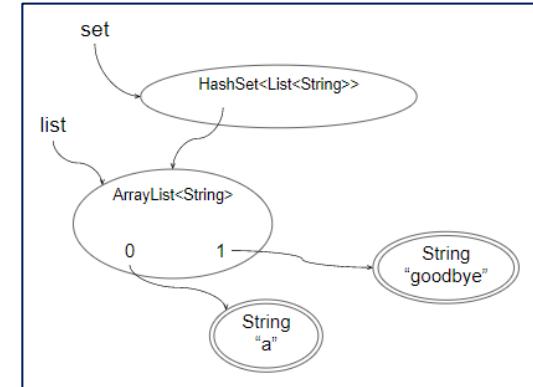
- And it **is still there** in the set!

```
set.contains(list)
```

- It's worse than that, in fact: when we **iterate** over the members of the set, we **still find** the list **in the set**, but **contains()** says it's **not there!**

```
for (List<String> l : set) {  
    set.contains(l)  
}
```

- If the set's own iterator and its own contains() method **disagree** about whether an element is in the set, then the set clearly is broken



# Equality and hashCode Mutable Types Bug (3)

---

- What's going on?
- `List<String>` is a mutable object, and in the standard Java implementation of collection classes like `List`, mutations affect the result of `equals()` and `hashCode()`
- When the list is first put into the `HashSet`, it is stored in the hash bucket *corresponding* to its `hashCode()` result at that time
- When the list is subsequently *mutated*, **its `hashCode()` changes**, but `HashSet` **doesn't** realize it should be moved to a different bucket
  - so it can *never* be found again
- When `equals()` and `hashCode()` can be affected by mutation, we can break the rep invariant of a hash table that uses that object as a key
- Here's a telling quote from the specification of `java.util.Set`:  
*Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is **not** specified if the value of an object is **changed** in a manner that affects equals comparisons while the object is an element in the set*

# Equality and hashCode Mutable Types Bug (4)

---

- The Java library is unfortunately *inconsistent* about its interpretation of equals() for mutable classes
  - Collections like List, Set, and Map use observational equality, but other mutable classes like StringBuilder and arrays use behavioral equality
- The lesson we should draw from this example is that equals() **should implement behavioral equality** for mutable objects
  - In general, that means that two references should be equals() if and only if they are aliases for the same object
  - So mutable objects should just inherit equals() and hashCode() from Object
- For clients that need a notion of observational equality (whether two mutable objects “look” the same in the current state), it's better to define a new method, e.g., similar()

# The Final Rule for equals() and hashCode() (1)

---

- For immutable types:
  - equals() should compare abstract values
    - this is the same as saying equals() should provide behavioral equality
  - hashCode() should map the abstract value to an integer
- So immutable types **must** override both equals() and hashCode()

# The Final Rule for equals() and hashCode() (2)

---

- For mutable types:
  - equals() should compare references, just like ==
    - Again, this is the same as saying equals() should provide behavioral equality
  - hashCode() should map the reference into an integer
- So mutable types **should not** override equals() and hashCode() at all, and should simply use the default implementations provided by Object
- Java **doesn't** follow this rule for its collections, unfortunately, leading to the pitfalls that we saw above

in this course, also, sometimes we still use observational equality for mutables objects

# Autoboxing and Equality (1)

---

- One more instructive pitfall in Java
- We've talked about primitive types and their object type equivalents — for example, `int` and `Integer`
- The object type implements `equals()` in the correct way, so that if you create two `Integer` objects with the same value, they'll be `equals()` to each other:

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

- But there's a subtle problem here; `==` is overloaded :  
For *reference* types like `Integer`, it implements **referential equality**:

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

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

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

## Autoboxing and Equality (2)

- So you **can't** really use Integer interchangeably with int
- The fact that Java **automatically converts** between int and Integer (this is called *autoboxing* and *autounboxing*) can lead to subtle bugs!
- You have to be aware what the compile-time types of your expressions are
- Consider this:

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

**Autoboxing** : when an int 130 is put into the map, it is automatically *boxed* up into a new Integer object

# Autoboxing and Equality (3)

- So you **can't** really use Integer interchangeably with int
- The fact that Java **automatically converts** between int and Integer (this is called *autoboxing* and *autounboxing*) can lead to subtle bugs!
- You have to be aware what the compile-time types of your expressions are
- Consider this:

```
Map<String, Integer> a = new HashMap<>(), b = new HashMap<>();  
a.put("c", 130);      // put ints into the map  
b.put("c", 130);  
int i = a.get("c")   // what do we get out of the maps?
```

Autounboxing : get() return an Integer object representing 130, but after that, if we assign the get() result to an int variable, the assignment automatically unboxes the Integer object into int value 130

# Fast Searching

---

- In the next part of this lecture, we will look at a data structure for ***efficiently searching for the existence of items*** within the data structure
- Yes, it is the **Hash Table**
  - we actually have learned to use it in Week 2 to do problem solving, as we learn and practice unit testing
  - we want to use hash code to cleverly to build a data structure

I personally know a couple friends who were asked about this very topic in their coding interview!

# Using an Array

---

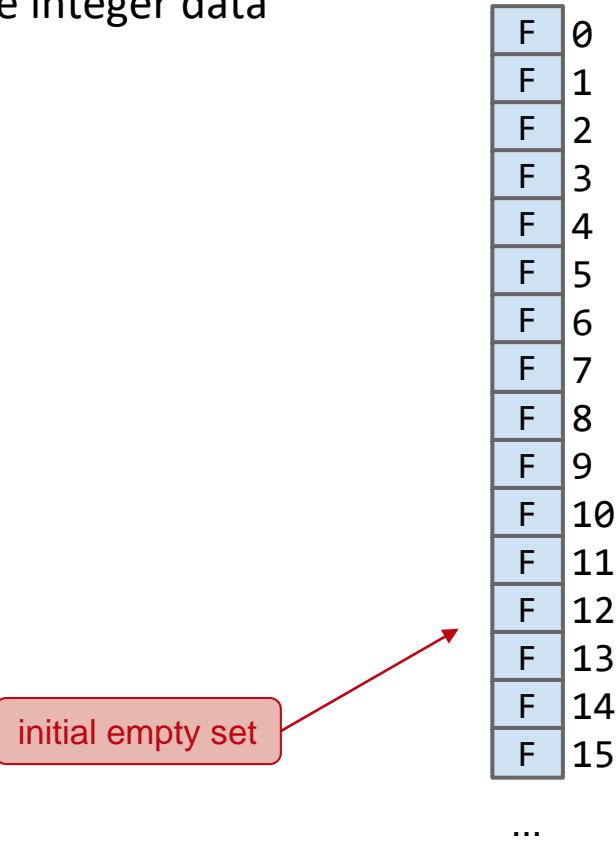
- Recall that in lab, you have created **ARSet**, an array-based set, that can do search
  - however, the searching method `contains()` cost is  $O(N)$ 
    - in the worst case, you have to search the whole array
  - we want to do much better than that, and yet *still using an array* as the rep !
- Let us start simple by *only* storing *integer* data
  - and call our set implementation **INTSet**
  - remember, our goal is to *make searching as fast as possible!*

# INTSet

---

- Idea: Use an array of boolean that is indexed by the integer data
  - that is, the index is actually the data
  - initially all boolean values are false

```
INTSet set;  
set = new INTSet();
```



# INTSet

---

- Idea: Use an array of boolean that is indexed by the integer data
  - that is, the index is actually the data
  - initially all boolean values are false
  - when an item is added, set corresponding index to true

```
INTSet set;  
set = new INTSet();  
set.add(5);
```

set contains 5

|     |    |
|-----|----|
| F   | 0  |
| F   | 1  |
| F   | 2  |
| F   | 3  |
| F   | 4  |
| T   | 5  |
| F   | 6  |
| F   | 7  |
| F   | 8  |
| F   | 9  |
| F   | 10 |
| F   | 11 |
| F   | 12 |
| F   | 13 |
| F   | 14 |
| F   | 15 |
| ... |    |



# INTSet

---

- Idea: Use an array of boolean that is indexed by the integer data
  - that is, the index is actually the data
  - initially all boolean values are false
  - when an item is added, set corresponding index to true

```
INTSet set;  
set = new INTSet();  
set.add(5);  
set.add(2);
```

set contains 5, 2

|     |    |
|-----|----|
| F   | 0  |
| F   | 1  |
| T   | 2  |
| F   | 3  |
| F   | 4  |
| T   | 5  |
| F   | 6  |
| F   | 7  |
| F   | 8  |
| F   | 9  |
| F   | 10 |
| F   | 11 |
| F   | 12 |
| F   | 13 |
| F   | 14 |
| F   | 15 |
| ... |    |



# INTSet

---

- Idea: Use an array of boolean that is indexed by the integer data
  - that is, the index is actually the data
  - initially all boolean values are false
  - when an item is added, set corresponding index to true

```
INTSet set;  
set = new INTSet();  
set.add(5);  
set.add(2);  
set.add(10);
```

contains(i) simply check  
whether value at index i is T or F

set contains 5, 2, 10

|     |    |
|-----|----|
| F   | 0  |
| F   | 1  |
| T   | 2  |
| F   | 3  |
| F   | 4  |
| T   | 5  |
| F   | 6  |
| F   | 7  |
| F   | 8  |
| F   | 9  |
| T   | 10 |
| F   | 11 |
| F   | 12 |
| F   | 13 |
| F   | 14 |
| F   | 15 |
| ... |    |



# INTSet Implementation

- The implementation is very simple:

```
public class INTSet {  
    private boolean[] items;  
  
    public INTSet() {  
        items = new boolean[2000000000];  
    }  
  
    public void add(int i) {  
        items[i] = true;  
    }  
  
    public boolean contains(int i) {  
        return items[i];  
    }  
}
```

roughly the number of integers

since accessing elements in  
an array is constant time,  
these operations =  $O(1)$  time

# INTSet Drawbacks

- There are some obvious problems with this implementation:

```
public class INTSet {  
    private boolean[] items;  
  
    public INTSet() {  
        items = new boolean[2000000000];  
    }  
  
    public void add(int i) {  
        items[i] = true;  
    }  
  
    public boolean contains(int i) {  
        return items[i];  
    }  
}
```

1. extremely wasteful of memory!  
even if you only want to store  
a few integers you still need  
two billion booleans

2. you can only store integers  
need a way to generalize

essentially, when we can solve  
those two issues, we will get  
ourselves a hash table !

# ENGSet

---

- Now suppose we want to use the same idea, but we want to store **lowercase English words** instead of just integers
- We want a set ENGSet that can:

```
ENGSet set;  
set = new ENGSet();  
set.add("cat");  
set.add("dog");  
set.add("animal");
```

but where should we put  
"cat", "dog", and "animal" ?

|     |    |
|-----|----|
| F   | 0  |
| F   | 1  |
| F   | 2  |
| F   | 3  |
| F   | 4  |
| F   | 5  |
| F   | 6  |
| F   | 7  |
| F   | 8  |
| F   | 9  |
| F   | 10 |
| F   | 11 |
| F   | 12 |
| F   | 13 |
| F   | 14 |
| F   | 15 |
| ... |    |

...

# Add into ENGSet (1)

---

- Suppose we want to add("cat")
  - what is the index for "cat" ?
  - idea: use the first letter as index  
 $a = 1, b = 2, c = 3, \dots, z = 26$

|     |    |
|-----|----|
| F   | 0  |
| F   | 1  |
| F   | 2  |
| T   | 3  |
| F   | 4  |
| F   | 5  |
| F   | 6  |
| F   | 7  |
| F   | 8  |
| F   | 9  |
| F   | 10 |
| F   | 11 |
| F   | 12 |
| F   | 13 |
| F   | 14 |
| F   | 15 |
| ... |    |

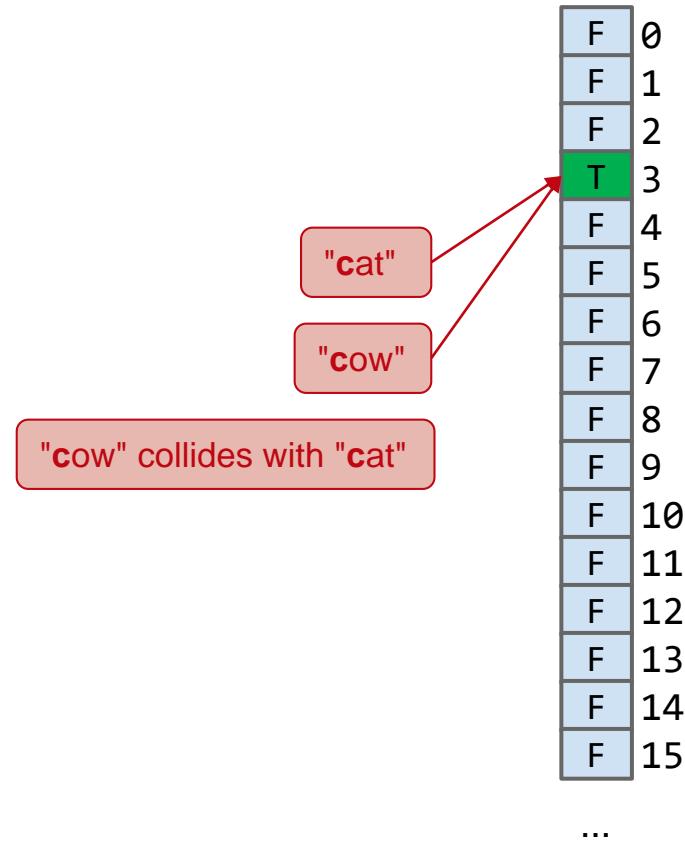
"cat"

...

## Add into ENGSet (2)

---

- Suppose we want to add("cat")
  - what is the index for "cat" ?
  - idea: use the first letter as index  
 $a = 1, b = 2, c = 3, \dots, z = 26$
  - however, contains("cow")  $\rightarrow T$
  - we have a **collision !**



## Add into ENGSet (3)

- Suppose we want to add("cat") with no collision
  - to avoid collision, idea: use **all** letters by multiplying each by a power of 27
  - 27 because there are 26 letters in English alphabet, and multiply by powers like a decimal number in base 10, e.g.  
 $521_{10} = (5 \times 10^2) + (2 \times 10^1) + (1 \times 10^0)$

|      |   |     |
|------|---|-----|
| 0    | F |     |
| 1    | F | a   |
| 2    | F | b   |
| 3    | F | c   |
| 4    | F | d   |
| ...  |   |     |
| 25   | F | y   |
| 26   | F | z   |
| ...  |   |     |
| 2233 | F | cas |
| 2234 | F | cat |
| 2235 | F | cau |
| ...  |   |     |

## Add into ENGSet (4)

- Suppose we want to add("cat") with no collision
  - to avoid collision, idea: use **all** letters by multiplying each by a power of 27
  - 27 because there are 26 letters in English alphabet, and multiply by powers like a decimal number in base 10, e.g.  
 $521_{10} = (5 \times 10^2) + (2 \times 10^1) + (1 \times 10^0)$
  - so, the index of "cat" is  $(3 \times 27^2) + (1 \times 27^1) + (20 \times 27^0) = 2234$

since we pick a base  $\geq 26$ , it is guaranteed to give each lowercase English word a unique number!  
we will have **no** collision!

|      |   |     |
|------|---|-----|
| 0    | F |     |
| 1    | F | a   |
| 2    | F | b   |
| 3    | F | c   |
| 4    | F | d   |
| ...  |   |     |
| 25   | F | y   |
| 26   | F | z   |
| ...  |   |     |
| 2233 | F | cas |
| 2234 | T | cat |
| 2235 | F | cau |
| ...  |   |     |

"cat"

2234

# Convert English String to Integer

- Method letterNum converts letter to number, and englishToInt computes the index

```
/* Converts ith character of String to a letter number.  
 * e.g. 'a' -> 1, 'b' -> 2, 'z' -> 26 */  
private static int letterNum(String s, int i) {  
    int ithChar = s.charAt(i);  
    if ((ithChar < 'a') || (ithChar > 'z'))  
        { throw new IllegalArgumentException(); }  
    return ithChar - 'a' + 1;  
}  
  
private static int englishToInt(String s) {  
    int intRep = 0;  
    for (int i = 0; i < s.length(); i++) {  
        intRep = intRep * 27;  
        intRep = intRep + letterNum(s, i);  
    }  
    return intRep;  
}
```

in Java, char 'a' is equivalent to 97  
this will give 1 to 'a', 2 to 'b', ...

# ENGSet Implementation

- Similar to INTSet, simply needs to compute the index first

```
public class ENGSet {  
    private boolean[] items;  
  
    public ENGSet() {  
        items = new boolean[2000000000];  
    }  
  
    public void add(String s) {  
        items[englishToInt(s)] = true;  
    }  
  
    public boolean contains(String s) {  
        return items[englishToInt(s)];  
    }  
}
```

compute index of s

|     |   |   |
|-----|---|---|
| 0   | F | a |
| 1   | F | b |
| 2   | F | c |
| 3   | F | d |
| 4   | F |   |
| ... |   |   |

|    |   |   |
|----|---|---|
| 25 | F | y |
| 26 | F | z |

|      |   |     |
|------|---|-----|
| 2233 | F | cas |
| 2234 | F | cat |
| 2235 | F | cau |

# ASCII Characters

- In addition to lowercase letter, what if we want to store uppercase? numbers? special chars?
- We can use ASCII encoding value as the index
  - each char is assigned a value between 0 and 127: `char c = 'a'` is equiv with `char c = 97` in Java

characters 33 - 126  
are printable

the others do other  
things, like the  
new line character

|    |    |    |   |    |   |    |   |     |   |     |   |
|----|----|----|---|----|---|----|---|-----|---|-----|---|
| 33 | !  | 49 | 1 | 65 | A | 81 | Q | 97  | a | 113 | q |
| 34 | "  | 50 | 2 | 66 | B | 82 | R | 98  | b | 114 | r |
| 35 | #  | 51 | 3 | 67 | C | 83 | S | 99  | c | 115 | s |
| 36 | \$ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | %  | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | &  | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | '  | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | (  | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 | )  | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | *  | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | +  | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { |
| 44 | ,  | 60 | < | 76 | L | 92 | \ | 108 | l | 124 |   |
| 45 | -  | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } |
| 46 | .  | 62 | > | 78 | N | 94 | ^ | 110 | n |     |   |
| 47 | /  | 63 | ? | 79 | O | 95 | _ | 111 | o |     |   |
| 48 | 0  | 64 | @ | 80 | P | 96 | ` | 112 | p |     |   |

## ASCII-based SET

---

- Maximum possible value for english-only text including punctuation is 126
- We can use 126 as our base in order to ensure unique values for all possible strings
- For examples:
  - $\text{bee}_{126} = (98 \times 126^2) + (101 \times 126^1) + (101 \times 126^0) = 1,568,675$
  - $\text{2pac}_{126} = (50 \times 126^3) + (112 \times 126^2) + (97 \times 126^1) + (99 \times 126^0) = 101,809,233$

# ASCII-based SET Implementation

---

- We can use the following method to convert the index:

```
public static int asciiToInt(String s) {  
    int intRep = 0;  
    for (int i = 0; i < s.length(); i++) {  
        intRep = intRep * 126;  
        intRep = intRep + s.charAt(i);  
    }  
    return intRep;  
}
```

but what we want store even more characters? e.g., Chinese characters?

# Unicode Characters



- We can also use Unicode encoding value, also supported in Java
  - for example, `char c = '\u2f04'` is equivalent to `char c = 40140`

|        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 一      |        | 丶      | ノ      | 乙      | 丶      | 二      | 一      | 人      | 儿      | 入      | 八      | 口      | 一      | ノ      | 几      |
| U+2F00 | U+2F01 | U+2F02 | U+2F03 | U+2F04 | U+2F05 | U+2F06 | U+2F07 | U+2F08 | U+2F09 | U+2F0A | U+2F0B | U+2F0C | U+2F0D | U+2F0E | U+2F0F |
| 口      | 刀      | 力      | 匚      | 匕      | 匚      | 匚      | 十      | ト      | 匚      | 厂      | ム      | 又      | 口      | 口      | 土      |
| U+2F10 | U+2F11 | U+2F12 | U+2F13 | U+2F14 | U+2F15 | U+2F16 | U+2F17 | U+2F18 | U+2F19 | U+2F1A | U+2F1B | U+2F1C | U+2F1D | U+2F1E | U+2F1F |
| 士      | 夕      | 夕      | 夕      | 大      | 女      | 子      | 宀      | 寸      | 小      | 尢      | 尸      | 山      | 《      | 工      | U+2F20 |
| 己      | 巾      | 干      | 幺      | 广      | 疋      | 升      | 弋      | 弓      | 丂      | 丂      | 彳      | 心      | 戈      | 戠      | 手      |
| 支      | 支      | 文      | 斗      | 斤      | 方      | 无      | 日      | 日      | 月      | 木      | 欠      | 止      | 歹      | 殳      | 毋      |
| U+2F30 | U+2F31 | U+2F32 | U+2F33 | U+2F34 | U+2F35 | U+2F36 | U+2F37 | U+2F38 | U+2F39 | U+2F3A | U+2F3B | U+2F3C | U+2F3D | U+2F3E | U+2F3F |
| 比      | 毛      | 氏      | 气      | 水      | 火      | 爪      | 父      | 爻      | 爿      | 片      | 牙      | 牛      | 犬      | 玄      | 玉      |
| 瓜      | 瓦      | 甘      | 生      | 用      | 田      | 疋      | 广      | 宀      | 白      | 皮      | 皿      | 目      | 矛      | 矢      | 石      |
| U+2F60 | U+2F61 | U+2F62 | U+2F63 | U+2F64 | U+2F65 | U+2F66 | U+2F67 | U+2F68 | U+2F69 | U+2F6A | U+2F6B | U+2F6C | U+2F6D | U+2F6E | U+2F6F |
| 示      | 内      | 禾      | 穴      | 立      | 竹      | 米      | 糸      | 缶      | 网      | 羊      | 羽      | 老      | 而      | 未      | 耳      |
| 聿      | 肉      | 臣      | 自      | 至      | 臼      | 舌      | 舛      | 舟      | 良      | 色      | 艸      | 虎      | 虫      | 血      | 行      |
| U+2F80 | U+2F81 | U+2F82 | U+2F83 | U+2F84 | U+2F85 | U+2F86 | U+2F87 | U+2F88 | U+2F89 | U+2F8A | U+2F8B | U+2F8C | U+2F8D | U+2F8E | U+2F8F |
| 衣      | 丂      | 見      | 角      | 言      | 谷      | 豆      | 豕      | 豸      | 貝      | 赤      | 走      | 足      | 身      | 車      | 辛      |
| 辰      | 辵      | 邑      | 酉      | 采      | 里      | 金      | 長      | 門      | 阜      | 隶      | 佳      | 雨      | 青      | 非      | 面      |
| 革      | 韋      | 韭      | 音      | 頁      | 風      | 飛      | 食      | 首      | 香      | 馬      | 骨      | 高      | 彫      | 門      | 鬯      |
| 离      | 鬼      | 魚      | 鳥      | 鹵      | 鹿      | 麥      | 麻      | 黃      | 黍      | 黑      | 黹      | 鼴      | 鼎      | 鼓      | 鼠      |
| U+2FC0 | U+2FC1 | U+2FC2 | U+2FC3 | U+2FC4 | U+2FC5 | U+2FC6 | U+2FC7 | U+2FC8 | U+2FC9 | U+2FCA | U+2FCB | U+2FCC | U+2FCD | U+2FCE | U+2FCF |

just some tiny part,  
the table is too big to display

# Unicode-based SET

---

- The largest possible value for Chinese characters is 40,959
  - (probably more based on the most recent version of Unicode that is continuously updated)
- So we would need to use that number as our base if we want to have a *unique* representation for all possible strings of Chinese characters
- For example:

$$\text{守门员}_{40959} = (23432 \times 40959^2) + (38376 \times 40959^1) + (21592 \times 40959^0) = \\ = 39,312,024,869,368$$



|                    |     |     |
|--------------------|-----|-----|
| ...                | F   | 守门呂 |
| 39,312,024,869,367 | T   | 守门员 |
| 39,312,024,869,368 | F   | 守门丙 |
| 39,312,024,869,369 | ... |     |

# Integer Overflow

---

- However, it is not possible to have that large of an integer index in Java
- In Java, the largest possible integer is 2,147,483,647
  - If you go over this limit, you **overflow**, starting back over at the smallest integer, which is -2,147,483,648

```
int x = 2147483647;  
System.out.println(x);  
System.out.println(x + 1);
```

→ 2147483647  
→ -2147483648

# Overflow causes Invalid Index and Collision

---

- Because Java has a maximum integer, we may not get the number we expect
  - With base 126, we will run into overflow *even* for short strings  
For example:  $\text{omens}_{126} = 28,196,917,171$   
which is much greater than the maximum integer,  
and `asciiToInt('omens')` will give us -1,867,853,901 instead
  - Overflow can result in collisions, causing incorrect answers

```
public void testOverflow() {  
    ENGSet set = new ENGSet();  
    set.add("melt banana");  
    set.contains("subterrestrial anticosmetic");  
    // asciiToInt for these strings is 839099497  
}
```

→ true

# Hash Code

---



- The term for the index/number we're computing is **hash code**
- According to Wolfram Alpha, a hash code "projects a value from a set with many (or even an infinite number of) members to a value from a set with a fixed number of (fewer) members."
  - here, our target set is the set of Java integers, which is of size 4294967296
  - thus, there can only be 4294967296 possible hash codes

# Pigeonhole Principle

- Pigeonhole principle tells us that if there are *more* than 4294967296 possible items, *multiple* items will share the *same* hash code

- there are more than 4294967296 planets
    - Each has mass, xPos, yPos, xVel, yVel, name
  - there are more than 4294967296 strings
    - "one", "two", ..., "nineteen quadrillion", ...



there are 9 boxes and 10 pigeons, no matter how you arrange the pigeons, it is guaranteed that at least 1 box is going to contain more than 1 pigeon

- Thus, **collisions are inevitable !**

# Two Fundamental Challenges

---

Two fundamental challenges:

- How do we resolve the hash code collisions?
  - let us call this **collision handling**
- How do we compute a hash code for *arbitrary* objects?
  - let us call this **computing a hash code**

# Handling Collision

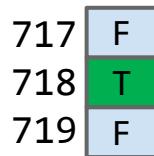
---

- Suppose N items have the same hash code h

suppose hash code for both  
"bee" and "w@\$p" is 718



...

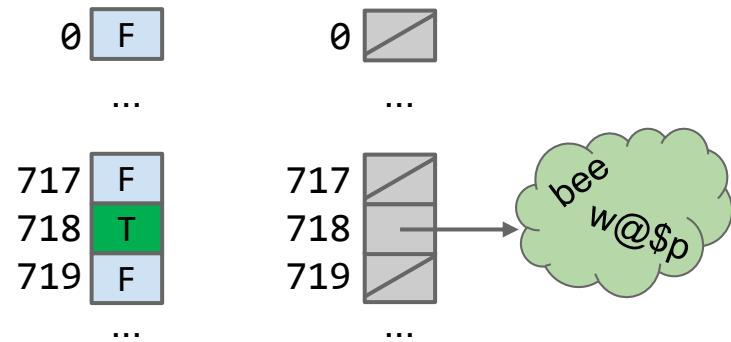


...

# Handling Collision

- Suppose N items have the same hash code h
  - instead of storing true in position h, store a **bucket** of these N items at position h

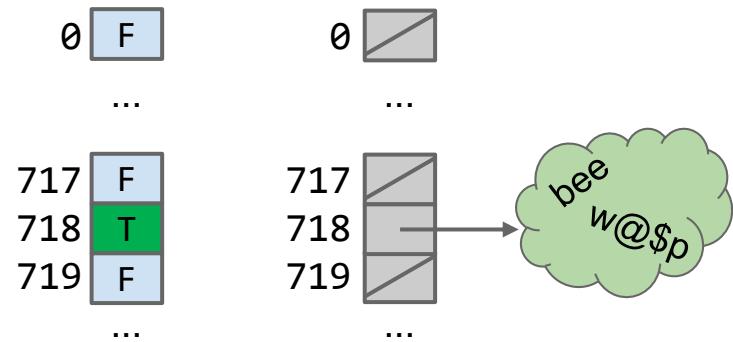
suppose hash code for both  
"bee" and "w@\$p" is 718



# Handling Collision

- Suppose N items have the same hash code h
  - instead of storing true in position h, store a **bucket** of these N items at position h
- How to implement a bucket?
  - use SLList
  - could use ARList
  - could also use ARSet
  - any data structure which you can search and iterate
- This idea is called **separate chaining**

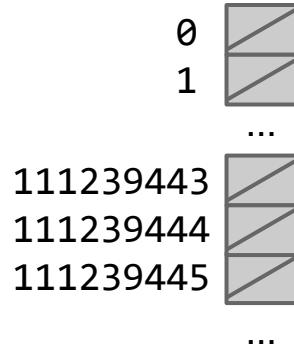
suppose hash code for both  
"bee" and "w@\$p" is 718



# Separate Chaining

---

- In separate chaining, when an item  $x$  gets added at index  $h$ :
  - If bucket  $h$  is empty, we create a new list containing  $x$  and store it at index  $h$
  - If bucket  $h$  is already a list, we add  $x$  to this list *if it is not already present*

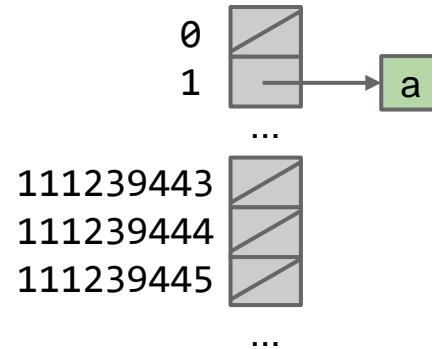


initially, all  
buckets are empty

# Separate Chaining

- In separate chaining, when an item  $x$  gets added at index  $h$ :
  - If bucket  $h$  is empty, we create a new list containing  $x$  and store it at index  $h$
  - If bucket  $h$  is already a list, we add  $x$  to this list *if it is not already present*

add("a")



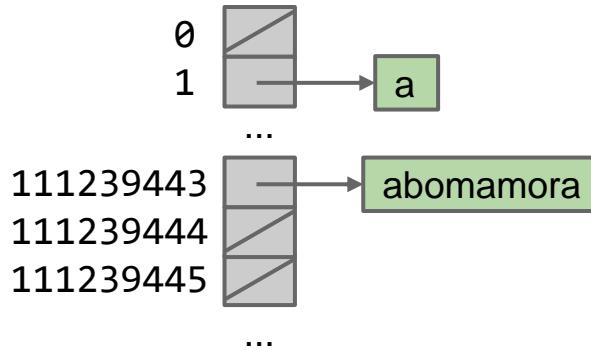
bucket 1 now has  
a list of length 1

# Separate Chaining

- In separate chaining, when an item  $x$  gets added at index  $h$ :
  - If bucket  $h$  is empty, we create a new list containing  $x$  and store it at index  $h$
  - If bucket  $h$  is already a list, we add  $x$  to this list *if it is not already present*

```
add("a")
add("abomamora")
```

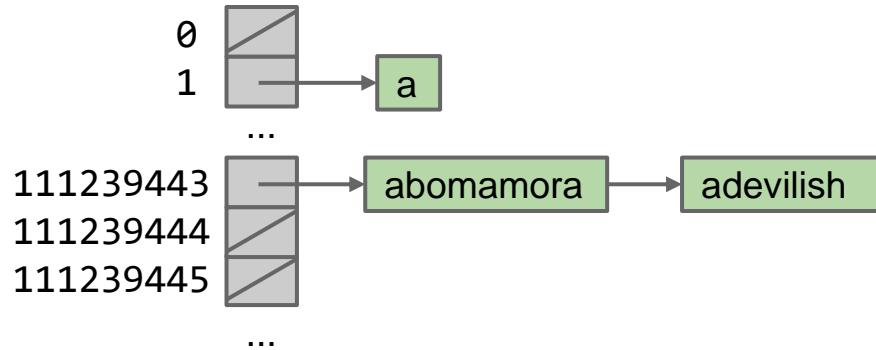
using englishToInt,  
abomamora hashes to  
111239443, create new list



# Separate Chaining

- In separate chaining, when an item  $x$  gets added at index  $h$ :
  - If bucket  $h$  is empty, we create a new list containing  $x$  and store it at index  $h$
  - If bucket  $h$  is already a list, we add  $x$  to this list *if it is not already present*

```
add("a")
add("abomamora")
add("adevilish")
```

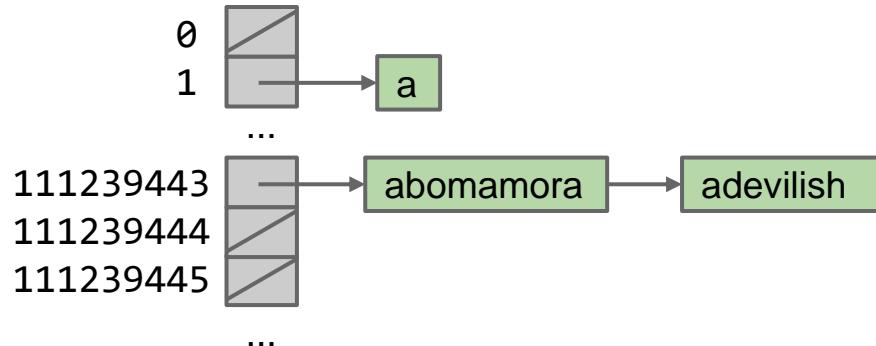


adevilish also hashes to  
111239443, added in list

# Separate Chaining

- In separate chaining, when an item  $x$  gets added at index  $h$ :
  - If bucket  $h$  is empty, we create a new list containing  $x$  and store it at index  $h$
  - If bucket  $h$  is already a list, we add  $x$  to this list *if it is not already present*

```
add("a")
add("abomamora")
add("adevilish")
add("abomamora")
```

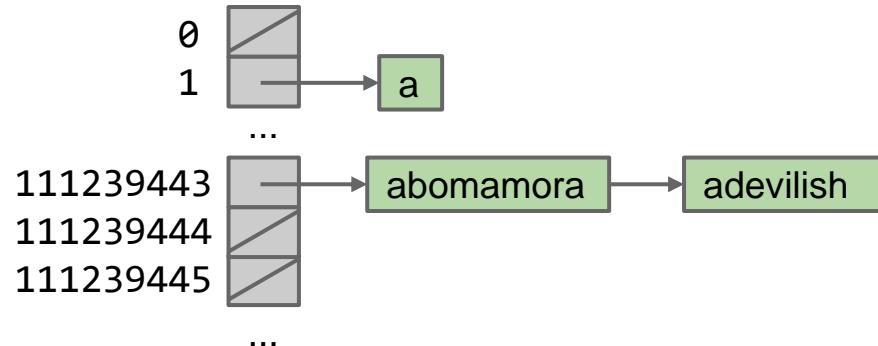


already present, not added

# Separate Chaining

- In separate chaining, when an item  $x$  gets added at index  $h$ :
  - If bucket  $h$  is empty, we create a new list containing  $x$  and store it at index  $h$
  - If bucket  $h$  is already a list, we add  $x$  to this list *if it is not already present*

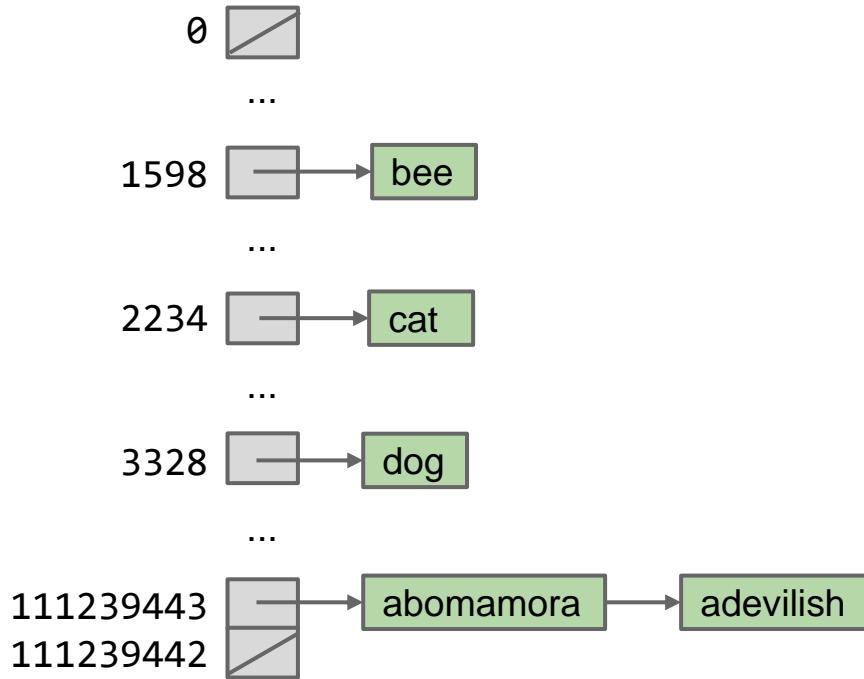
```
add("a")
add("abomamora")
add("adevilish")
add("abomamora")
contains("adevilish")
```



→ return true

# Separate Chaining Performance

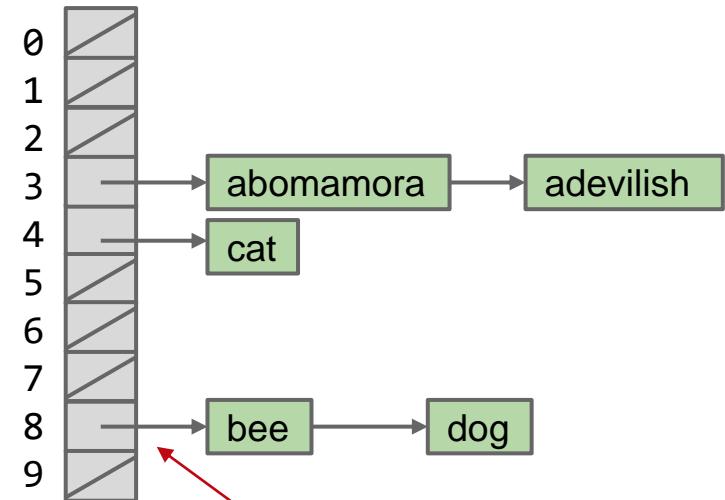
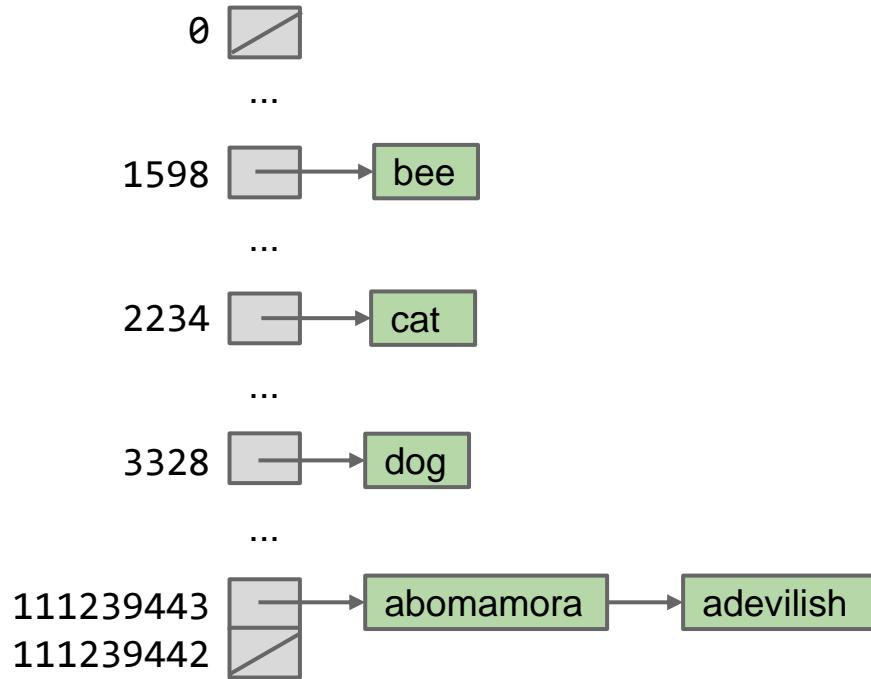
- The running time of add and contain is  $O(Q)$ , where  $Q$  is the length of the longest list
  - no longer  $O(1)$ , since to add, need to check whether the item is already in the list



let us improve our design.  
notice that we don't need  
billions of buckets.  
what can we do to save space?

# Saving Memory

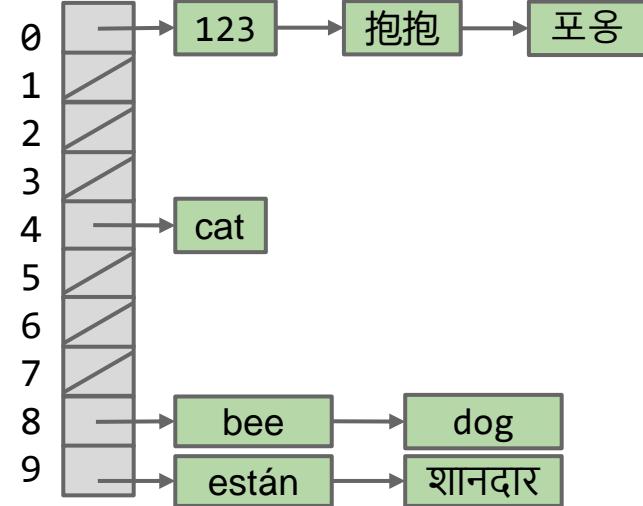
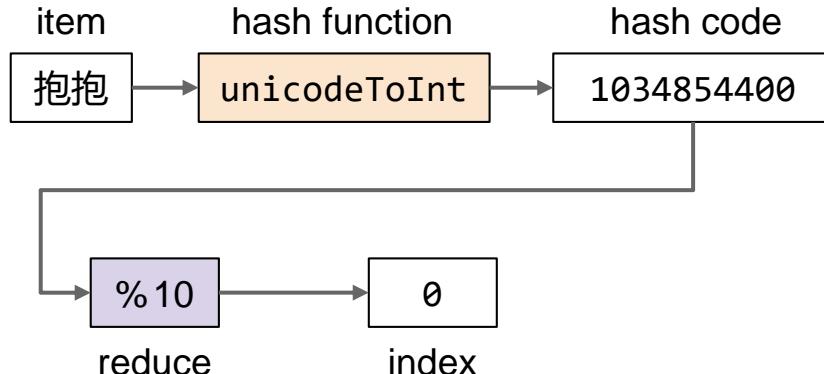
- We could reduce the number of buckets, and use modulus of hash code to compute index
  - for example, use 10 buckets, and compute bucket number = hash code % 10



bee:  $1598 \% 10 \rightarrow 8$   
dog:  $3328 \% 10 \rightarrow 8$

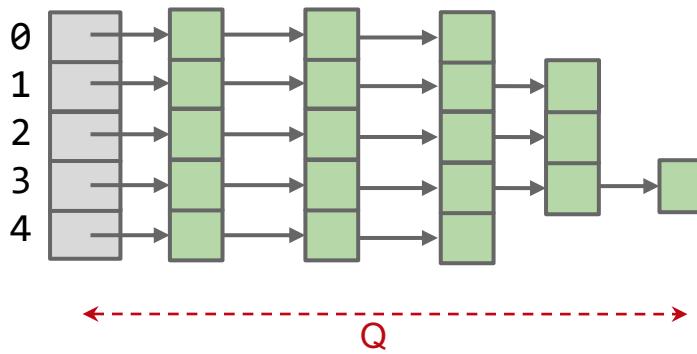
# Hash Table

- We have just created the so-called **hash table**
  - The item is converted by a **hash function** into an integer representation called a **hash code**
  - The hash code is then reduced to a **valid index**, usually using the modulus operator



# Hash Table Performance

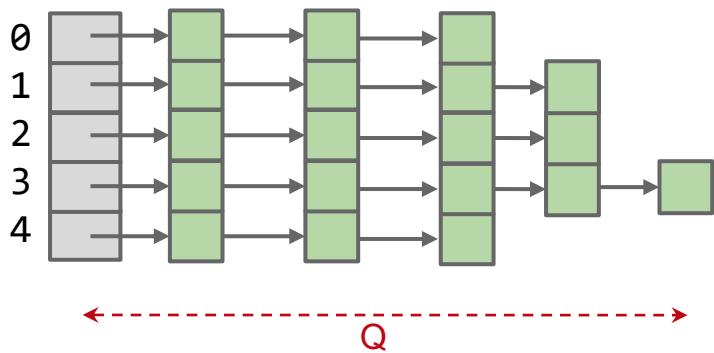
- Hash table uses way less memory/space than INTSet or ENGSet, but now the running time is no longer constant  $O(1)$ , it becomes  $O(Q)$ 
  - where  $Q$  is the length of the longest list



for this hash table that has 5 buckets,  
and stores  $N$  items,  
what is the order of growth of  $Q$   
in terms of  $N$  ?

# Hash Table Performance

- Hash table uses way less memory/space than INTSet or ENGSet, but now the running time is no longer constant  $O(1)$ , it becomes  $O(Q)$ 
  - where  $Q$  is the length of the longest list

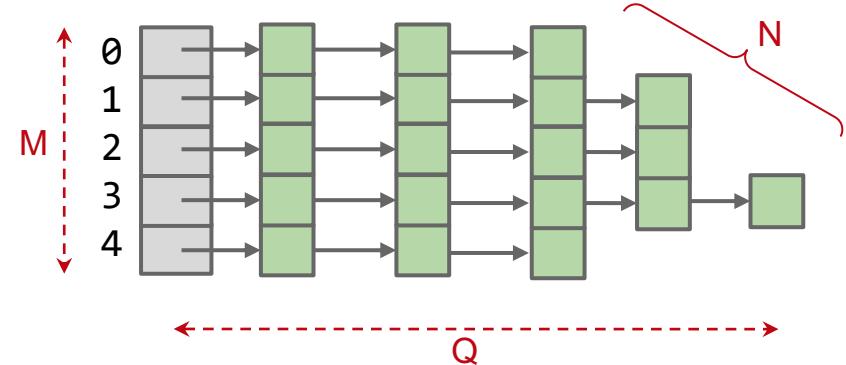


for this hash table that has 5 buckets, and stores  $N$  items, what is the order of growth of  $Q$  in terms of  $N$  ?

In the best case, the items are evenly distributed, the length of the longest list will be  $N/5$ .  
In the worst case, all items in a single list, it will be  $N$ .  
So,  $Q(N)$  is  $O(N)$ , which is not good

# Improving Hash Table

- Suppose we have:
  - A fixed number of buckets  $M$
  - An increasing number of items  $N$

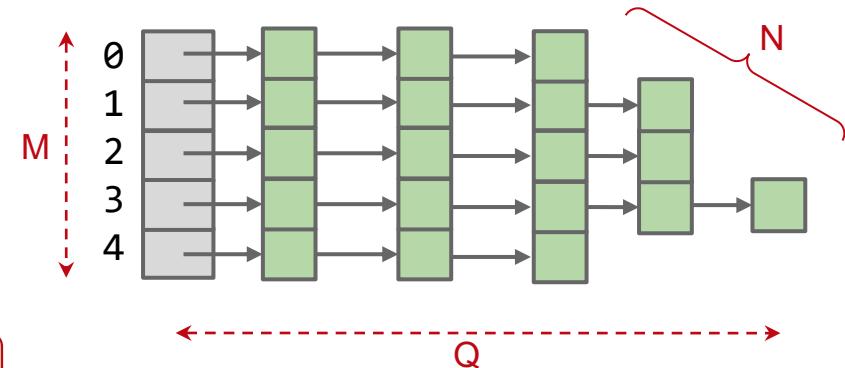


- Even if items are spread out evenly, lists are of length  $Q = N/M$ 
  - For  $M = 5$ , that means  $Q = O(N)$
  - Results in linear time operations, which is bad
- How can we improve our design to guarantee that  $N/M$  is  $O(1)$ ?

# Improving Hash Table

- Suppose we have:
  - A ~~fixed~~ number of buckets  $M$
  - An **increasing**
  - An increasing number of items  $N$

improve the ratio by growing  $M$  at the same rate of  $N$



- Even if items are spread out evenly, lists are of length  $Q = N/M$ 
  - For  $M = 5$ , that means  $Q = O(N)$
  - Results in linear time operations, which is bad
- How can we improve our design to guarantee that  $N/M$  is  $O(1)$ ?

# Improving Hash Table 2

---

- As long as  $M = O(N)$ , then  $O(N/M) = O(1)$
- Idea: When  $N/M$  is  $\geq 1.5$ , then double M
  - when it gets too full, double the buckets!
  - we call this process of increasing  $M$  **resizing**
  - $N/M$  is often called the **load factor**
  - it represents how full the hash table is
- This rule ensures that the average list is never more than 1.5 items long!

use this idea in your Lab 15: HAMap !

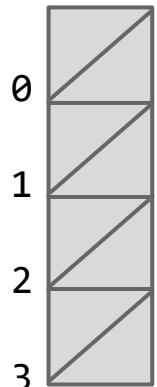
# Improving Hash Table Example

---

- Idea: When  $N/M$  is  $\geq 1.5$ , then double M

suppose we start with 4 empty buckets

$$N = 0 \quad M = 4 \quad N/M = 0$$



# Improving Hash Table Example

---

- Idea: When  $N/M \geq 1.5$ , then double M

$$N = 1 \quad M = 4 \quad N/M = 0.25$$

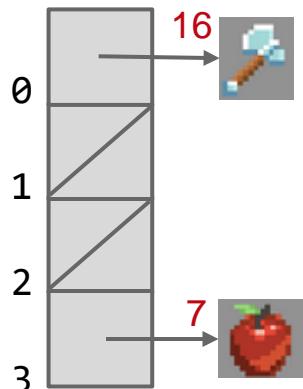


# Improving Hash Table Example

---

- Idea: When  $N/M \geq 1.5$ , then double M

$$N = 2 \quad M = 4 \quad N/M = 0.5$$

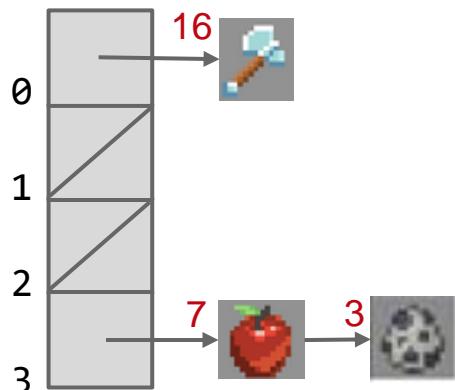


# Improving Hash Table Example

---

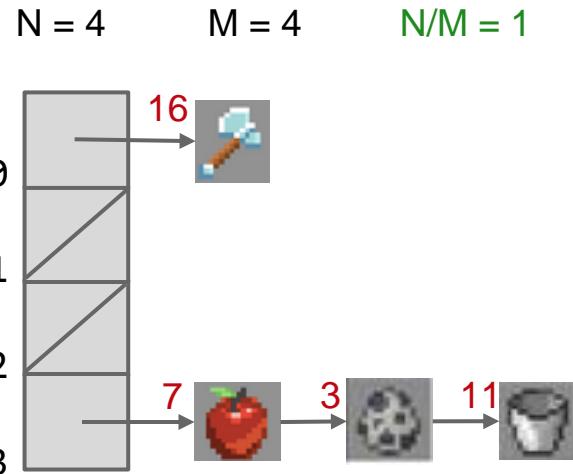
- Idea: When  $N/M \geq 1.5$ , then double M

$$N = 3 \quad M = 4 \quad N/M = 0.75$$



# Improving Hash Table Example ---

- Idea: When  $N/M \geq 1.5$ , then double M

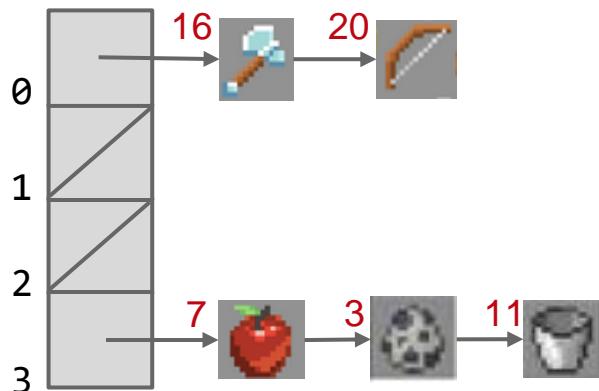


# Improving Hash Table Example

---

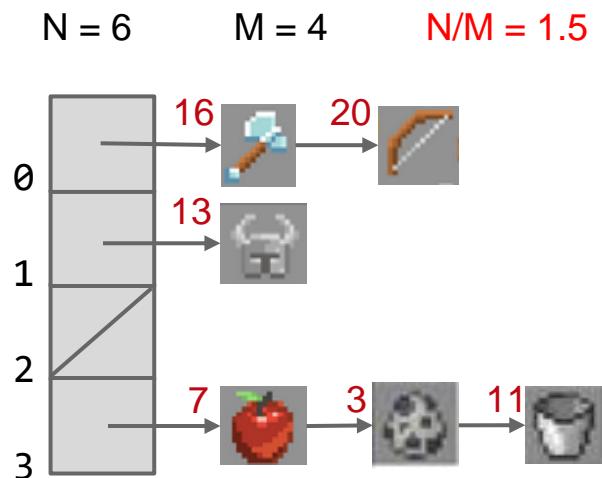
- Idea: When  $N/M \geq 1.5$ , then double M

$$N = 5 \quad M = 4 \quad N/M = 1.25$$



# Improving Hash Table Example

- Idea: When  $N/M \geq 1.5$ , then double M



N/M is too large,  
it's time to double!

# In-Class Quiz 2

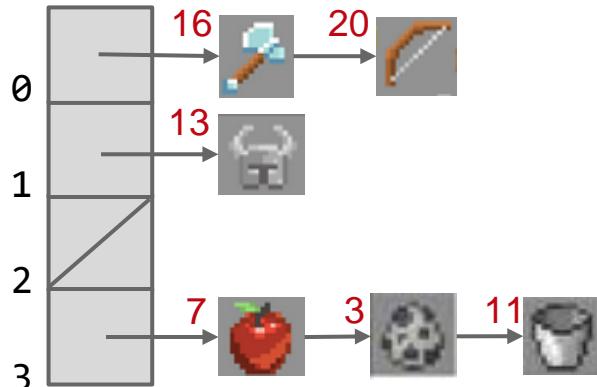
- Idea: When  $N/M \geq 1.5$ , then double M

where do we put this item  
after doubling?



?

$$N = 6 \quad M = 4 \quad N/M = 1.5$$



## N/M is too large, it's time to double!

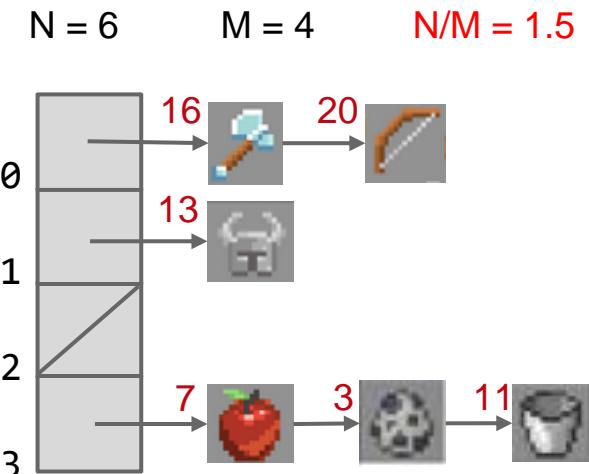
|   |   |
|---|---|
| 0 | ? |
| 1 | ? |
| 2 | ? |
| 3 | ? |
| 4 | ? |
| 5 | ? |
| 6 | ? |
| 7 | ? |

Select one:

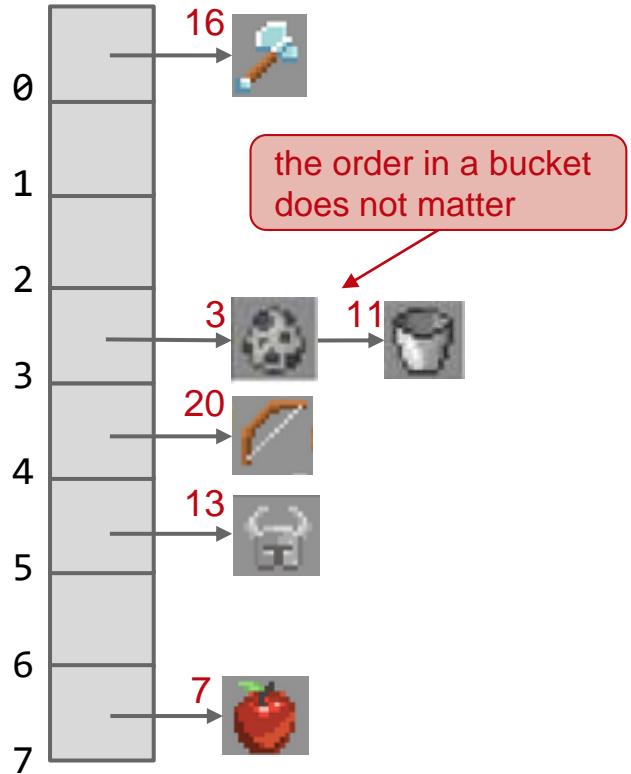
- 0
  - 4
  - 8
  - 20

# Improving Hash Table Example

- Idea: When  $N/M \geq 1.5$ , then double M



$N = 6$      $M = 8$      $N/M = 0.75$



# Resizing Hash Table Running Time

---

- As long as  $M = O(N)$ , then  $O(N/M) = O(1)$
- *Assuming items are evenly distributed*, lists will be approximately  $N/M$  items long, resulting in  $O(N/M)$  runtimes
  - Our doubling strategy ensures that  $N/M = O(1)$
  - Thus, worst case runtime for all operations is  $O(N/M) = O(1)$
  - unless that operation causes a resize
- What is the runtime of resizing?

# Resizing Hash Table Running Time

---

- As long as  $M = O(N)$ , then  $O(N/M) = O(1)$
- *Assuming items are evenly distributed*, lists will be approximately  $N/M$  items long, resulting in  $O(N/M)$  runtimes
  - Our doubling strategy ensures that  $N/M = O(1)$
  - Thus, worst case runtime for all operations is  $O(N/M) = O(1)$
  - unless that operation causes a resize
- Resizing takes  $O(N)$  time, because we have to redistribute all items!
  - Most add operations will be  $O(1)$ , but some will be  $O(N)$  time to resize
  - Similar to our ARList, as long as we resize by a multiplicative factor, the average runtime will still be  $O(1)$

## Even Distribution

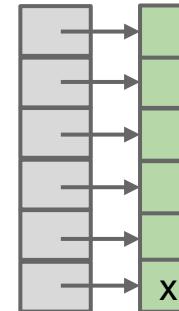
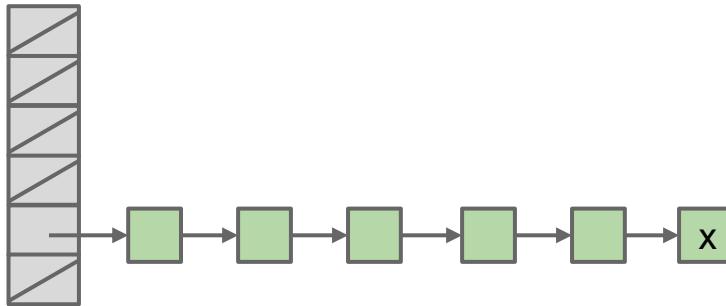
---

- Thus, Resizing Hash Table operations are on average constant time if:
  1. we double M to ensure constant average bucket length
  2. items are evenly distributed

# Even Distribution

- Thus, Resizing Hash Table operations are on average constant time if:
  1. we double M to ensure constant average bucket length
  2. items are evenly distributed
- Even distribution of item is critical for good hash table performance
  - o both tables below have load factor of  $N/M = 1$
  - o but left table is much worse, contains is  $O(N)$  for item x

we need to discuss how to ensure even distribution !



# Object's Hash Code

- Recall that all classes are subclass of Object

| Modifier and Type           | Method                                      | Description                                                                                                                                                            |
|-----------------------------|---------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| protected                   | <code>Object clone()</code>                 | Creates and returns a copy of this object.                                                                                                                             |
| boolean                     | <code>equals (Object obj)</code>            | Indicates whether some other object is "equal to" this one.                                                                                                            |
| protected void              | <code>finalize()</code>                     | <b>Deprecated.</b><br>The finalization mechanism is inherently problematic.                                                                                            |
| <code>Class&lt;?&gt;</code> | <code>getClass()</code>                     | Returns the runtime class of this Object.                                                                                                                              |
| int                         | <code>hashCode()</code>                     | >Returns a hash code value for the object.                                                                                                                             |
| void                        | <code>notify()</code>                       | Wakes up a single thread that is waiting on this object's monitor.                                                                                                     |
| void                        | <code>notifyAll()</code>                    | Wakes up all threads that are waiting on this object's monitor.                                                                                                        |
| <code>String</code>         | <code>toString()</code>                     | Returns a string representation of the object.                                                                                                                         |
| void                        | <code>wait()</code>                         | Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> .                                                     |
| void                        | <code>wait (long timeout)</code>            | Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed. |
| void                        | <code>wait (long timeout, int nanos)</code> | Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed. |

default implementation of  
hashCode returns the memory  
address of the object

# String's Hash Code

- Java's String overrides hashCode
  - for example:

```
System.out.println("a".hashCode());  
System.out.println("bee".hashCode());  
System.out.println("포옹".hashCode());  
System.out.println("kamala lifefully".hashCode());  
System.out.println("đậu hũ".hashCode());
```

→ 97  
→ 97410  
→ 1732557  
→ 1732557  
→ -2108180664

a collision !

a negative !

# Negative Hash Code in Java

- Suppose that the hash code of an item is -1
- Unfortunately,  $-1 \% 4 \rightarrow -1$ 
  - will result in index errors!
- Solution: use `Math.floorMod` instead

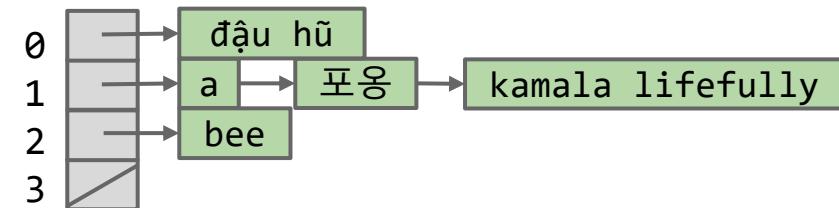
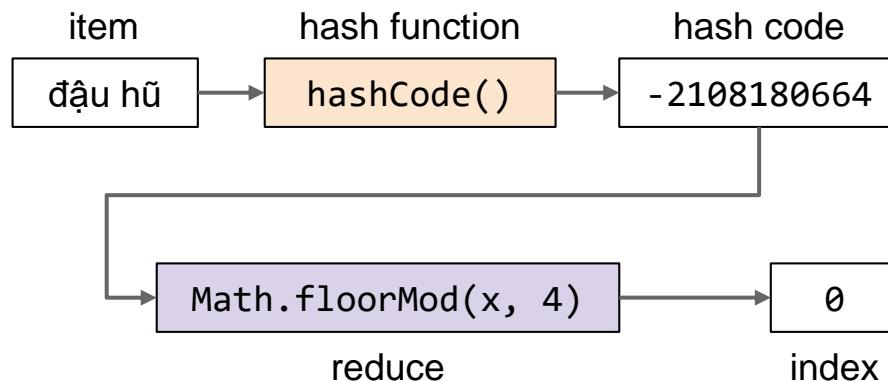


```
public class ModTest {  
    public static void main(String[] args) {  
        System.out.println(-1 % 4);  
        System.out.println(Math.floorMod(-1, 4));  
    }  
}
```

→ -1  
→ 3

# Our Hash Table Set Implementation HASet

- Here is our set implementation using hash tables in Java, called HASet:
  - item is converted by the **hashCode method** into an integer representation called a **hash code**
  - the hash code is then reduced to a **valid index**, using the `Math.floorMod` method



# HASet: Member Variables and Constructor

- Here is our hash-based implementation of set

```
/*
 * Hash-based Set
 */
public class HASet<T> {

    private static final int DEFAULT_CAPACITY = 16;

    private ArrayList<ArrayList<T>> buckets;
    private int size;

    private int reduce(T item) {
        return Math.floorMod(item.hashCode(), DEFAULT_CAPACITY);
    }

    public HASet() {
        buckets = new ArrayList<>();
        for (int i = 0; i < DEFAULT_CAPACITY; i++) {
            buckets.add(new ArrayList<>());
        }
        size = 0;
    }
}
```

as a rep of a (fixed) hash table,  
we use ArrayList of ArrayList

hash function and reduce

initializing the buckets

# HASet: Contains

---

- Recall that in a set, an item is either in a set or not

```
/*
 * Checks whether an item is inside the set.
 * @param item to be checked
 * @return true iff the set contains the item
 */
public boolean contains(T item) {
    if (item == null) {
        return false;
    }
    int index = reduce(item);
    return buckets.get(index).contains(item);
}
```

compute the index,  
go to the right bucket,  
and call ArrayList.contains

# HASet: Add

---

- Recall that in a set, we must have no duplicates

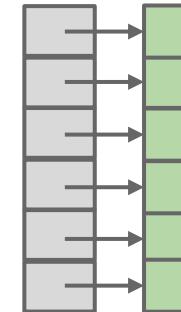
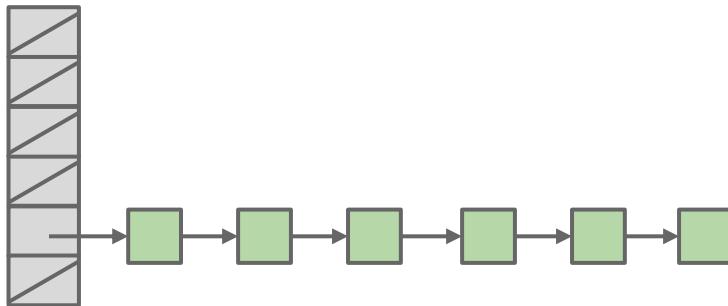
```
/**  
 * Adds an item into the set if it is not already inside.  
 * @param item to be added inside the set.  
 * @throws IllegalArgumentException if item is null.  
 */  
  
public void add(T item) {  
    if (item == null) {  
        throw new IllegalArgumentException();  
    }  
    if (contains(item)) {  
        return;  
    }  
    int index = reduce(item);  
    buckets.get(index).add(item);  
    size++;  
}
```

compute the index,  
go to the right bucket,  
and add the item,  
if it is not already in there

# Bad hashCode

---

- The following are examples of bad hash functions
  - return 0
  - just returning the first character of a word, e.g. "cat" → 3
  - adding chars together, e.g. "ab" collides with "ba"
- Recall that we want hash tables that look like the table on the right:



# String hashCode Method

- The following is the Java 8 hashCode for String:

```
@Override  
public int hashCode() {  
    int h = cachedHashCode;  
    if (h == 0 && this.length() > 0) {  
        for (int i = 0; i < this.length(); i++) {  
            h = 31 * h + this.charAt(i);  
        }  
        cachedHashCode = h;  
    }  
    return h;  
}
```

it cached the value, only compute if it is not computed before

it uses a small base

# Choosing a Base

---

- Java's hashCode() function for Strings:
  - $h(s) = s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + \dots + s_{n-1}$
- Our asciiToInt function for Strings:
  - $h(s) = s_0 \times 126^{n-1} + s_1 \times 126^{n-2} + \dots + s_{n-1}$
- Which one is better?
  - recall that overflow is problem for base 126
  - and in turns, that causes collision

## Base 126

---

- Major collision problem:
  - "Laughter is the best thing on the earth.".hashCode() yields 634199182
  - "Muesli is the best thing on the earth.".hashCode() yields 634199182
  - "Being heard is the best thing on the earth.".hashCode() yields 634199182
  - "Memes is the best thing on the earth.".hashCode() yields 634199182
- Any string that ends in the same last 32 characters has the same hash code !
  - Why? Because of overflow
  - This causes  $126^{32} = 126^{33} = 126^{34} = \dots = 0$
  - Thus, upper characters are all multiplied by zero

## Typical Base

---

- A typical hash code base is a **small prime**
- Why prime?
  - Never even: avoid the overflow issue on previous slide
  - Lower chance of resulting hash code having a bad relationship with the number of buckets, e.g. the hash code is a multiple of the number of buckets
- Why small?
  - Lower cost to compute

# Collection hashCode Method

- Suppose we have a collection of items, and each item has its own hashCode
  - somewhat similar to the one in String:

```
@Override  
public int hashCode() {  
    int hashCode = 1;  
    for (Object o : this) {  
        hashCode = hashCode * 31;  
        hashCode = hashCode + o.hashCode();  
    }  
    return hashCode;  
}
```

The diagram illustrates the logic of the hashCode method. It shows the code within a light blue box. Three red arrows point from three red callout boxes to specific parts of the code:

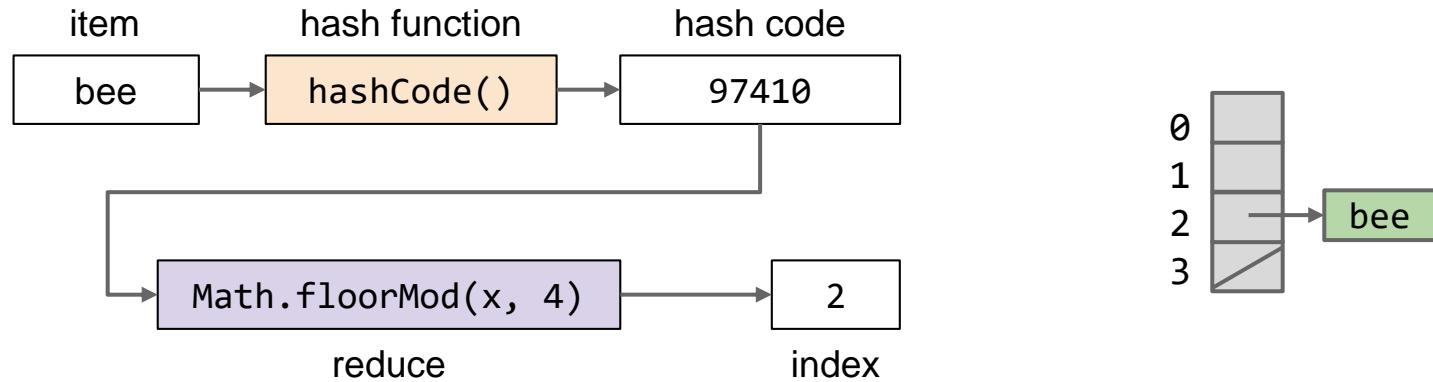
- An arrow points from the top box ("iterate over all items") to the loop iteration line: `for (Object o : this) {`.
- An arrow points from the middle box ("raise" to the "next power" with small prime) to the multiplication line: `hashCode = hashCode * 31;`.
- An arrow points from the bottom box ("add the hashCode of an item") to the addition line: `hashCode = hashCode + o.hashCode();`.

Below the light blue box is a red callout box containing the text: "we may only look at the first few items, since the Collection maybe very large".

we may only look at the first few items,  
since the Collection maybe very large

# Summary for Hash Table

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



# Higher Order Functions

---

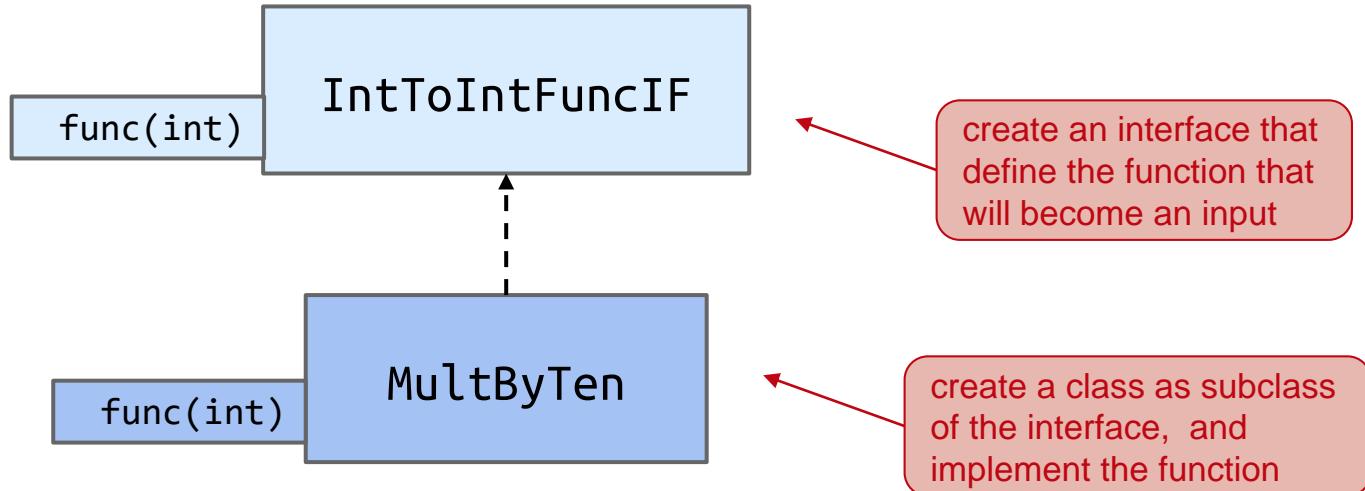
- Next, we will talk about Comparator which uses higher order function
  - **Higher order function** is a function that takes another function as *input*
    - for example, the function `applyTwice` below takes another function `func` :
- function `applyTwice(func, x) : func(func(x))`
- so, given function `multByTen(y) : 10 * y`  
we have that, `applyTwice(multByTen, 5) → 500`


$$\begin{aligned} &\text{multByTen(multByTen(5))} \\ &= \text{multByTen}(10 * 5) \\ &= 10 * 50 \\ &= 500 \end{aligned}$$

how do we do  
this in Java?

# Higher Order Functions in Java 7

- In Java 7 and earlier, variables cannot contain pointers to functions
- So, we use an interface instead:



- We also learn this design pattern because it is going to be used in our next topic, the Comparator

# Higher Order Functions by Interface (1)

---

1. Create an interface that declare the function which will be an input to another function

```
public interface IntToIntFuncIF {  
    int func(int x);  
}
```

represent an int to int function func

2. Create a class that is a subclass of that interface, and implement the function

```
public class MultByTen implements IntToIntFuncIF {  
  
    /** @return ten times the int argument */  
    public int func(int x) {  
        return 10 * x;  
    }  
  
}
```

implement it as  $10 * x$  function

# Higher Order Functions by Interface (2)

3. Implement the function `applyTwice` that takes the interface and call the function declared in that interface

```
public class HigherOrderFunc {  
  
    public static int applyTwice(IntToIntFuncIF f, int x) {  
        return f.func(f.func(x));  
    }  
  
    public static void main(String[] args) {  
        IntToIntFuncIF multByTen = new MultByTen();  
        System.out.println(applyTwice(multByTen, 5));  
    }  
}
```

so instead of passing a function,  
we pass an object that implements the  
function, and call the function (method)

→ 500

3. Call the function `applyTwice` by passing an object of a specific class that implements the function as guaranteed by the interface

# Higher Order Functions in Java 8

- Actually, you can do the following in the newer version:

```
import java.util.function.Function;

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

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

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

must use apply

but the one with interface is used in Comparator next ...

# Natural Order

---

- **Natural Order** is used to refer to the ordering implied by a Comparable's compareTo method, for example by **size** in our Dog objects :



"XiaXue", size: 20



"FooDog", size: 200



"HellHound", size: 4444

## Natural Order 2

---

- We don't always want to compare objects in the same way every time
  - for example, by *name*:



"FooDog", size: 200



"HellHound", size: 4444



"XiaXue", size: 20

# Additional Orders

---

- To do that in Java, we use the Higher Order Function technique: we create SizeComparator or NameComparator classes that implement Comparator interface
  - and fill in the method that is required called compare :

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

differentiate from  
compareTo of  
Comparable interface

same rule with compareTo (positive if o1 is bigger, ... )

- The methods comparing T objects must also take Comparator<T> arguments
  - for example, a comparison-based sorting method

# NameComparator for Dog

- As an example, here is how implement NameComparator for Dog objects:

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

use private static class that implements Comparator <Dog>, so we need to import java.util.Comparator

implement compare by calling compareTo of String, since we compare names alphabetically

use a getter to return an object of Comparator<Dog>

# Calling NameComparator for Dog Client

- And here is how we use the NameComparator for Dog objects:

```
import java.util.Comparator;

public class DogClient {
    public static void main(String[] args) {

        Dog d1 = new Dog( n: "XiaXue", w: 20);
        Dog d2 = new Dog( n: "FooDog", w: 200);
        Dog d3 = new Dog( n: "HellHound", w: 4444);

        Comparator<Dog> nameComp = Dog.getNameComparator();
        if (nameComp.compare(d1, d2) < 0) {
            d1.bark();
        }
        else {
            d2.bark();
        }
    }
}
```

first, need to instantiate the NameComparator by calling the getter

call compare, in this case, do decide if d1 comes before than d2 in the alphabet

this we can compare two objects in many ways!

## Part 2 : Concurrency

---

- In the second part of the lecture, we will
  - learn about message passing and shared memory models of concurrency
  - study concurrent processes and threads, time slicing, and the danger of race conditions

# Concurrency (1)

---

- Concurrency means **multiple** computations are happening **at the same time**
- Concurrency is *everywhere* in modern programming, whether we like it or not:
  - Multiple computers in a network
  - Multiple applications running on one computer
  - Multiple processors in a computer
    - today, often multiple processor *cores* on a single chip

## Concurrency (2)

---

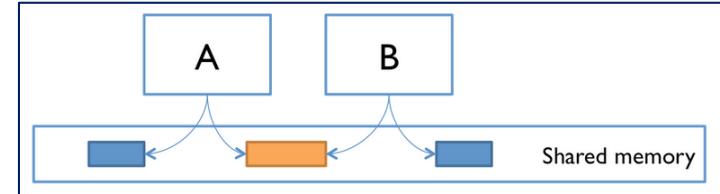
- In fact, concurrency is essential in modern programming:
  - Web sites must handle multiple simultaneous users
  - Mobile apps need to do some of their processing on servers "*in the cloud*"
  - Graphical user interfaces almost always require background work that does not interrupt the user
    - for example, IntelliJ compiles your Java code *while* you're still editing it!
- Being able to program with concurrency will still be important in the future
  - Processor clock speeds are no longer increasing
  - Instead, we're getting more cores with each new generation of chips
  - So in the future, in order to get a computation to run faster, we'll have to split up a computation into concurrent pieces

# Two Models for Concurrent Programming (1)

---

There are two common models for concurrent programming: shared memory and message passing

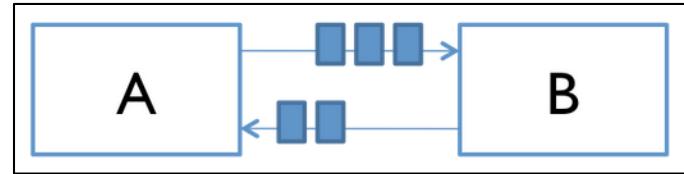
- **Shared memory.** In the shared memory model of concurrency, concurrent modules interact by *reading and writing shared objects in memory*
- Examples of the shared-memory model:
  - A and B might be two *processors* (or processor cores) in the same computer, sharing the same *physical memory*
  - A and B might be two *programs* running on the same computer, sharing a common *filesystem* with files they can read and write
  - A and B might be two *threads* in the same Java program (will explain what a thread is), sharing the same *Java objects*



# Two Models for Concurrent Programming (2)

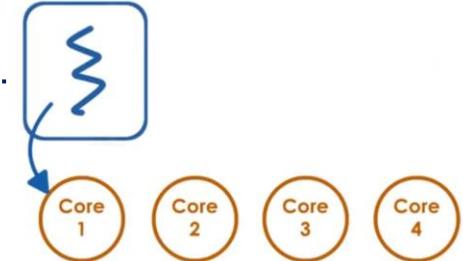
---

- **Message passing.** In the message-passing model, concurrent modules interact by sending messages to each other through *a communication channel*
- Modules send off messages, and incoming messages to each module are *queued* up for handling
- Examples include:
  - A and B might be two *computers* in a network, communicating by *network connections*
  - A and B might be *a web browser* and *a web server* — A opens a *connection* to B and asks for a *web page*, and B sends the *web page data* back to A
  - A and B might be an instant messaging *client* and *server*
  - A and B might be two *programs* running on the same computer whose input and output have been connected by a *pipe*, like `ls | grep` typed into a command prompt



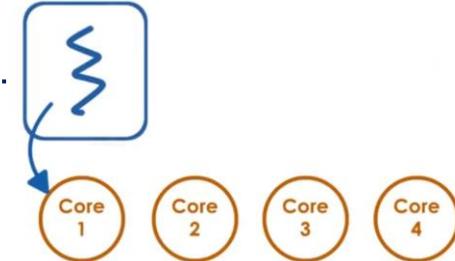
# Processes (1)

- The message-passing and shared-memory models are about *how concurrent modules communicate*
- The *concurrent modules* themselves come in two different kinds: processes and threads
- **Process.** A process is *an instance of a running program* that is isolated from other processes on the same machine
  - in particular, it has *its own private section* of the machine's memory
- The process abstraction is a *virtual computer*
  - it makes the program feel like it has the entire machine to itself — like a fresh computer has been created, with fresh memory, just to run that program



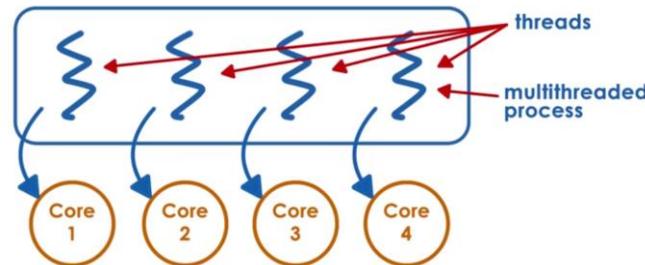
## Processes (2)

- Just like computers connected across a network, processes normally *share no memory* between them
  - A process *can't* access another process's memory or objects at all
    - sharing memory between processes is possible on most operating systems, but it needs special effort
  - By contrast, a new process is *automatically ready for message passing*, because it is created with standard input and output streams
    - which are the `System.out` and `System.in` streams you've used in Java



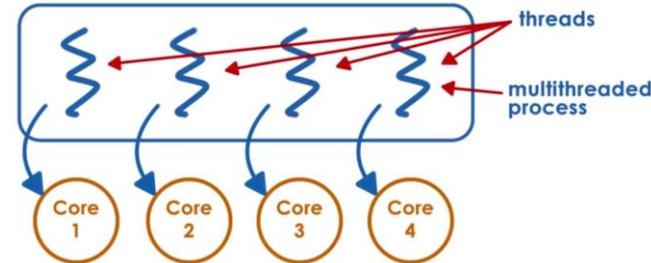
# Threads (1)

- **Thread.** A thread is *a locus of control inside a running program*
  - think of it as a place in the program that is being run,
  - plus the stack of method calls that led to that place
    - so the thread can go back up the stack when it reaches return statements
- Just as a process represents a virtual computer, the thread abstraction represents *a virtual processor*
  - making a new thread simulates making a fresh processor inside the virtual computer represented by the process
  - this new virtual processor *runs the same program and shares the same memory as other threads* in the process



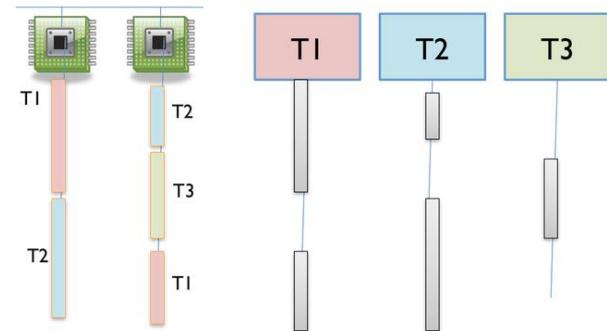
## Threads (2)

- Threads are *automatically ready for shared memory*, because threads share all the memory in the process
  - it takes special effort to get "thread-local" memory that's private to a single thread
  - it's also necessary to set up message-passing explicitly, by creating and using queue data structures
    - will discuss about that in future lecture
- Whenever you run a Java program, the program starts with one thread, which calls `main()` as its first step
  - this thread is referred to as the **main thread**



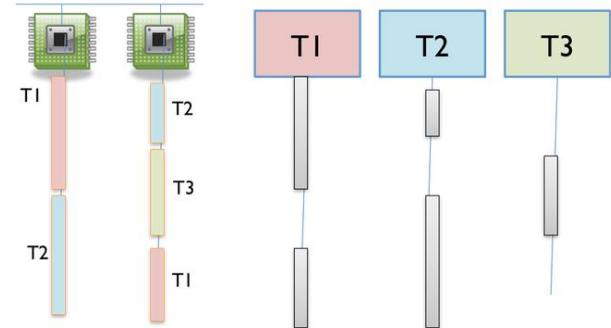
# Time-slicing (1)

- How can you have many concurrent threads with only one or two processors in my computer?
- When there are more threads than processors, concurrency is *simulated* by **time slicing**, which means that the processor switches between threads
- The figure above shows how *three* threads T1, T2, and T3 might be time-sliced on a machine that has only *two* actual processors



## Time-slicing (2)

- In the figure, time proceeds downward, so at first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3
- Thread T2 simply pauses, until its next time slice on the same processor or another processor
- On most systems, time slicing happens unpredictably and nondeterministically, meaning that a thread may be paused or resumed at any time



# Starting a thread in Java (1)

- You can start new threads by making an instance of Thread and telling it to start()
  - You provide code for the new thread to run by creating a class implementing Runnable
  - The first thing the new thread will do is call the run() method in this class
  - For example:

```
// ... in the main method:  
new Thread(new HelloRunnable()).start();  
  
// elsewhere in the code  
public class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello! It's me, a thread!");  
    }  
}
```

create an instance of HelloRunnable class, and pass that instance to the Thread constructor

## Starting a thread in Java (2)

---

- But a very common idiom is starting a thread with an *anonymous* Runnable, which eliminates the need to name the HelloRunnable class at all:

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

# Anonymous Class (1)

---

- An **anonymous class** is an implementation of an interface that has ***no class name*** of its own
- Usually when we implement an interface, we do so by declaring a class
  - For example, in the first part of lecture, the interface Comparator in the Java API:

```
/** A comparison function that imposes a total ordering on some objects.  
 * ... */  
  
public interface Comparator<T> {  
  
    /** Compares its two arguments for order.  
     * ...  
     * @return a negative integer, zero, or a positive integer if the first  
     *         argument is less than, equal to, or greater than the second */  
    public int compare(T o1, T o2);  
}
```

## Anonymous Class (2)

---

- We might declare:

```
/** Orders Strings by length (shorter first) and then lexicographically. */
public class StringLengthComparator implements Comparator<String> {

    @Override
    public int compare(String s1, String s2) {
        if (s1.length() == s2.length()) {
            return s1.compareTo(s2);
        }
        return s1.length() - s2.length();
    }
}
```

## Anonymous Class (3)

- One purpose of Comparator is for sorting, for example SortedSet keeps its items in a total order
- Without a Comparator, the SortedSet implementation uses the compareTo method provided by the objects in the set:

```
SortedSet<String> strings = new TreeSet<>();  
strings.addAll(Arrays.asList("yolanda", "zach", "alice", "bob"));  
// strings is { "alice", "bob", "yolanda", "zach" } ←
```

With a Comparator:

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

lexicographically

by length, then lexicographically

## Anonymous Class (4)

---

- If we only intend to use this comparator in this one place, we already know how to eliminate the variable:

```
// uses StringLengthComparator declared in previous slide  
SortedSet<String> strings = new TreeSet<>(new StringLengthComparator());
```

## Anonymous Class (5)

---

- An anonymous class declares an *unnamed* class that implements an interface and immediately creates the *one and only* instance of that class

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

## Anonymous Class (6)

---

- Advantages of anonymous class over named class:
  - if we're only using the comparator in this one piece of code, we've reduced its scope by using an anonymous class; with a named class, any other code could start using and depending on `StringLengthComparator`
  - a reader no longer has to search elsewhere for the details of the comparator; everything is right here
- Disadvantages:
  - If we need the same comparator more than once, we might be tempted to copy-and-paste; a named class is more DRY
  - If the implementation of the comparator is long, it interrupts the surrounding code, making it harder to understand; a named class is separated out as a modular piece
- So anonymous classes are good for ***short one-off*** implementations of a method

the Runnable we use to create new threads often meet these criteria perfectly

# Using an anonymous Runnable to start a thread

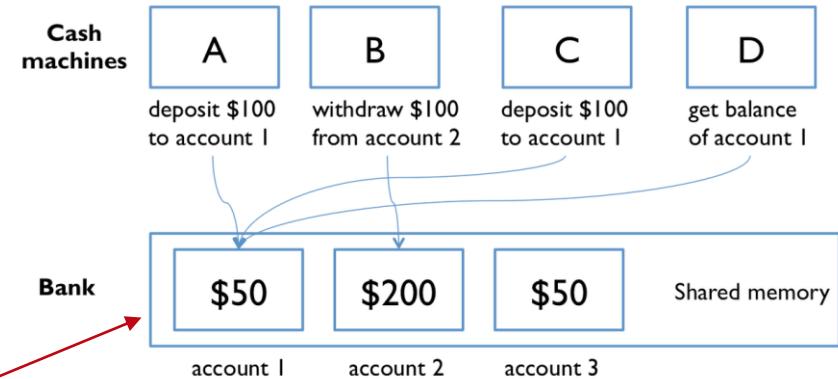
---

- To start a new thread, we create an anonymous implementation of Runnable which represents the code we want the new thread to start running:

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

# Shared Memory System (1)

- Let's look at an example of a shared memory system
- The point of this example is to show that concurrent programming is *hard*, because it can have subtle bugs
- Imagine that a bank has cash machines that use a *shared memory model*, so all the cash machines can *read* and **write the same account objects in memory**



# Shared Memory System (2)

---

- To illustrate what can go wrong, let's simplify the bank down to a single account, with a dollar balance stored in the balance variable, and two operations deposit and withdraw that simply add or remove a dollar:

```
// suppose all the cash machines share a single bank account  
  
private static int balance = 0;  
  
private static void deposit() {  
  
    balance = balance + 1;  
  
}  
  
private static void withdraw() {  
  
    balance = balance - 1;  
  
}
```

- Customers use the cash machines to do transactions like this:

```
deposit(); // put a dollar in  
  
withdraw(); // take it back out
```

# Shared Memory System (3)

- Each transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account *unchanged*
- Throughout the day, each cash machine in our network is processing a sequence of deposit/withdraw transactions:

```
// each ATM does a bunch of transactions that modify balance,
// but leave it unchanged afterward
public static void cashMachine() {
    new Thread(new Runnable() {
        public void run() {
            for (int i = 0; i < TRANSACTIONS_PER_MACHINE; i++) {
                deposit(); // put a dollar in
                withdraw(); // take it back out
            }
        }
    }).start();
}
```

## Shared Memory System (4)

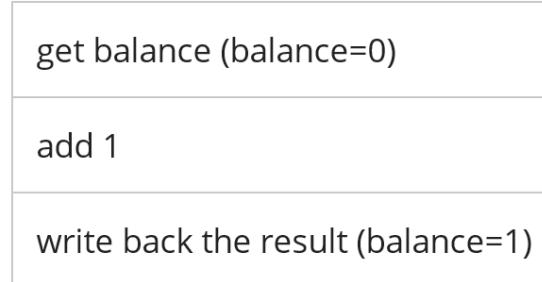
---

- So at the end of the day, regardless of how many cash machines were running, or how many transactions we processed,
  - we should *expect* the account balance to **still be 0**
- But if we run this code, we discover frequently that the balance at the end of the day is not 0
  - if more than one thread *created by* cashMachine() is running at the same time – say, on separate processors in the same computer – then balance **may not** be zero at the end of the day
  - *why not?*

# Interleaving (1)

---

- Here's one thing that can happen
- Suppose two cash machine threads, A and B, are *both working on a deposit at the same time*
- Here's how the deposit() step typically breaks down into *low-level processor instructions*:



## Interleaving (2)

---

- When A and B are running concurrently, these low-level instructions can **interleave** with each other, meaning that the actual sequence of execution can intersperse A's operations and B's operations arbitrarily
- Here is one possible interleaving:

| A                                   | B                                   |
|-------------------------------------|-------------------------------------|
| A get balance (balance=0)           |                                     |
| A add 1                             |                                     |
| A write back the result (balance=1) |                                     |
|                                     | B get balance (balance=1)           |
|                                     | B add 1                             |
|                                     | B write back the result (balance=2) |

- This interleaving is fine — we end up with balance 2, so both A and B successfully put in a dollar

# In-Class Quiz 3

---

- But what if the interleaving looked like this:

| A                         | B                         |
|---------------------------|---------------------------|
| A get balance (balance=0) |                           |
|                           | B get balance (balance=0) |
| A add 1                   |                           |
|                           | B add 1                   |
| A write back the result   |                           |
|                           | B write back the result   |

- What is the value of balance now ?
  - 0
  - 1
  - 2
  - 3

## Interleaving (3)

---

- But what if the interleaving looked like this:

| A                         | B                         |
|---------------------------|---------------------------|
| A get balance (balance=0) |                           |
|                           | B get balance (balance=0) |
| A add 1                   |                           |
|                           | B add 1                   |
| A write back the result   |                           |
|                           | B write back the result   |

- The balance is now — A's dollar was **lost!**
- A and B *both read the balance at the same time*, computed separate final balances, and then **raced** to store back the new balance — which failed to take the other's deposit into account

# Race Condition

---

- That is an example of a **race condition**
- A race condition means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the *relative timing* of events in concurrent computations A and B
  - when this happens, we say "**A is in a race with B**"
- Some interleavings of events may be OK, in the sense that they are consistent with what a single, non-concurrent process would produce, but other interleavings produce wrong answers — violating postconditions or invariants

# Tweaking the Code Won't Help (1)

---

- All these versions of the bank-account code exhibit the same race condition:

```
// version 1
private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}

// version 2
private static void deposit() {
    balance += 1;
}
private static void withdraw() {
    balance -= 1;
}
```

## Tweaking the Code Won't Help (2)

---

```
// version 3  
private static void deposit() {  
    ++balance;  
}  
private static void withdraw() {  
    --balance;  
}
```

## Tweaking the Code Won't Help (3)

---

- You can't tell just from looking at Java code how the processor is going to execute it
- You can't tell what the **atomic operations** (the indivisible steps of the computation) will be
  - it isn't atomic just because it's one line of Java
  - it doesn't touch balance only once just because the balance identifier occurs only once in the line
- The Java compiler, and in fact the processor itself, makes *no commitments* about what low-level operations it will generate from your code
  - in fact, a typical modern Java compiler produces exactly the same code for **all three** of those versions!
- The key lesson is that you *can't tell* by looking at an expression whether it will be safe from race conditions

## Reordering (1)

---

- It's even worse than that, in fact
- The race condition on the bank account balance can be explained in terms of different interleavings of sequential operations on different processors
- But in fact, when you're using multiple variables and multiple processors, you ***can't even count on*** changes to those variables *appearing in the same order*

## Reordering (2)

---

- Here's an example of a broken code
- Note that it uses a loop that continuously checks for a concurrent condition; this is called **busy waiting** and it is *not* a good pattern

```
private boolean ready = false;
private int answer = 0;
// computeAnswer runs in one thread
private void computeAnswer() {
    // ... calculate for a long time ...
    answer = 42;
    ready = true;
}
// useAnswer runs in a different thread
private void useAnswer() {
    while (!ready) {
        Thread.yield();
    }
    if (answer == 0) throw new RuntimeException("answer wasn't ready!");
}
```

## Reordering (3)

---

- We have two methods that are being run in different threads
  - computeAnswer does a long calculation, finally coming up with the answer 42, which it puts in the answer variable
  - then it sets the ready variable to true, in order to signal to the method running in the other thread, useAnswer, that the answer is ready for it to use
- Looking at the code, answer is set before ready is set,
  - so once useAnswer sees ready as true,
  - then it seems reasonable that it can assume that the answer will be 42, *right?*
    - *not so...*

## Reordering (4)

---

- The problem is that modern compilers and processors do a lot of things to make the code fast
  - one of those things is *making temporary copies of variables* like answer and ready in faster storage (processor registers or processor caches), and
  - *working with them temporarily* before eventually storing them back to their official location in memory
  - the storeback **may occur in a different order** than the variables were manipulated in your code

## Reordering (5)

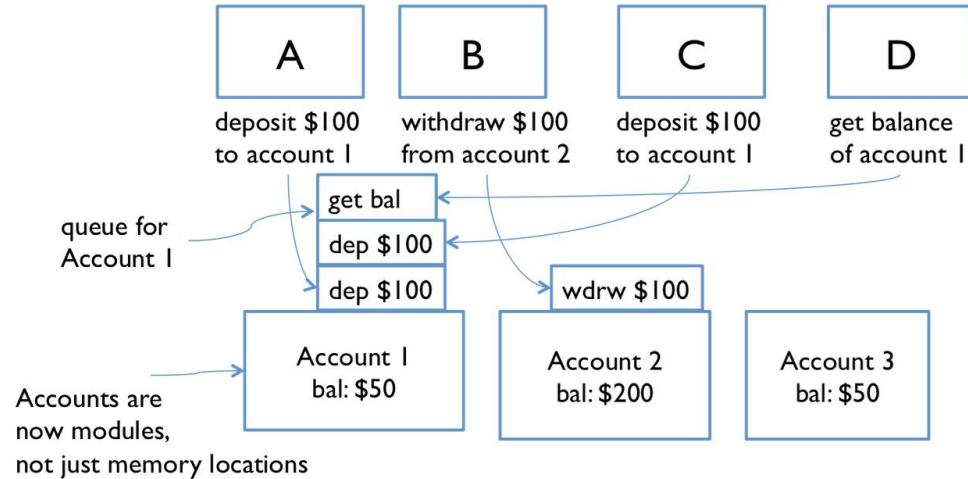
---

- Here's what might be going on under the covers (but expressed in Java syntax to make it clear)
- The processor is effectively creating two temporary variables, tmpr and tmpa, to manipulate the fields ready and answer:

```
private void computeAnswer() {  
    boolean tmpr = ready;  
    int tmpa = answer;  
    tmpa = 42;  
    tmpr = true;  
    ready = tmpr;  
        // <-- what happens if useAnswer() interleaves here?  
        // ready is set, but answer isn't!  
    answer = tmpa;  
}
```

# Message Passing Example (1)

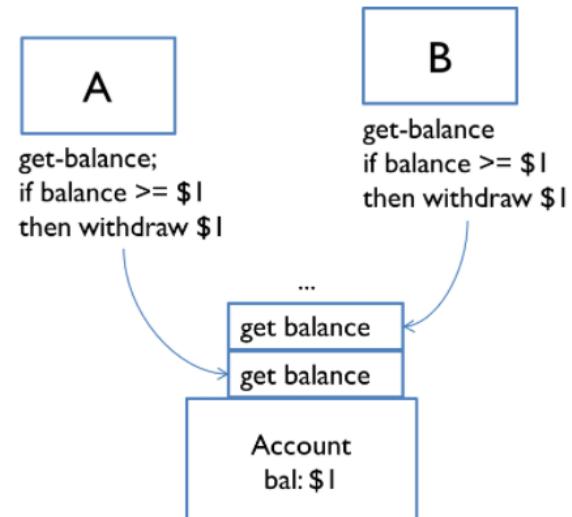
- Now let's look at the *message-passing approach* to our bank account example
- Now not only are the cash machine modules, but the accounts are modules, too
- Modules interact by sending messages to each other
- Incoming requests are placed in a queue to be handled one at a time
- The sender *doesn't stop working* while waiting for an answer to its request
  - it handles more requests from its own queue
  - the reply to its request eventually comes back as another message



## Message Passing Example (2)

- Unfortunately, message passing *doesn't* eliminate the possibility of *race conditions*
- Suppose each account supports get-balance and withdraw operations, with corresponding messages
  - Two users, at cash machines A and B, are both trying to withdraw a dollar from the same account
  - They check the balance first to make sure they never withdraw more than the account holds, because overdrafts trigger big bank penalties:

```
get-balance  
if balance >= 1 then withdraw 1
```



## Message Passing Example (3)

---

- The problem is again interleaving, but this time *interleaving of the messages* sent to the bank account, rather than the *instructions* executed by A and B
- If the account starts with a dollar in it, then what interleaving of messages will fool A and B into thinking they can both withdraw a dollar, thereby overdrawing the account
- One lesson here is that you need to carefully choose the operations of a message-passing model
  - withdraw-if-sufficient-funds would be a better operation than just withdraw

# Concurrency is Hard to Test and Debug (1)

---

- As we clearly see from these examples, concurrency is tricky, and here's the worst of it
  - it's **very hard to discover** race conditions using testing
  - and even once a test has found a bug, it may be **very hard to localize** it to the part of the program causing it
- Concurrency bugs exhibit **very poor reproducibility**
  - it's hard to make them happen the same way twice
  - interleaving of instructions or messages depends on the *relative timing of events* that are strongly influenced by the environment
  - delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc
  - each time you run a program containing a race condition, you may get different behavior

## Concurrency is Hard to Test and Debug (2)

---

- These kinds of bugs are **heisenbugs**, which are nondeterministic and hard to reproduce,
  - as opposed to a **bohrbug**, which shows up repeatedly whenever you look at it
  - almost all bugs in sequential programming are bohrbugs
- A heisenbug may even *disappear* when you try to look at it with `println` or debugger!
  - the reason is that printing and debugging are so much slower than other operations, often 100-1000x slower, that they dramatically change the timing of operations, and the interleaving

# Concurrency is Hard to Test and Debug (3)

- Inserting a simple print statement into the cashMachine():

```
private static void cashMachine() {  
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; i++) {  
        deposit(); // put a dollar in  
        withdraw(); // take it back out  
        System.out.println(balance); // makes the bug disappear!  
    }  
}
```

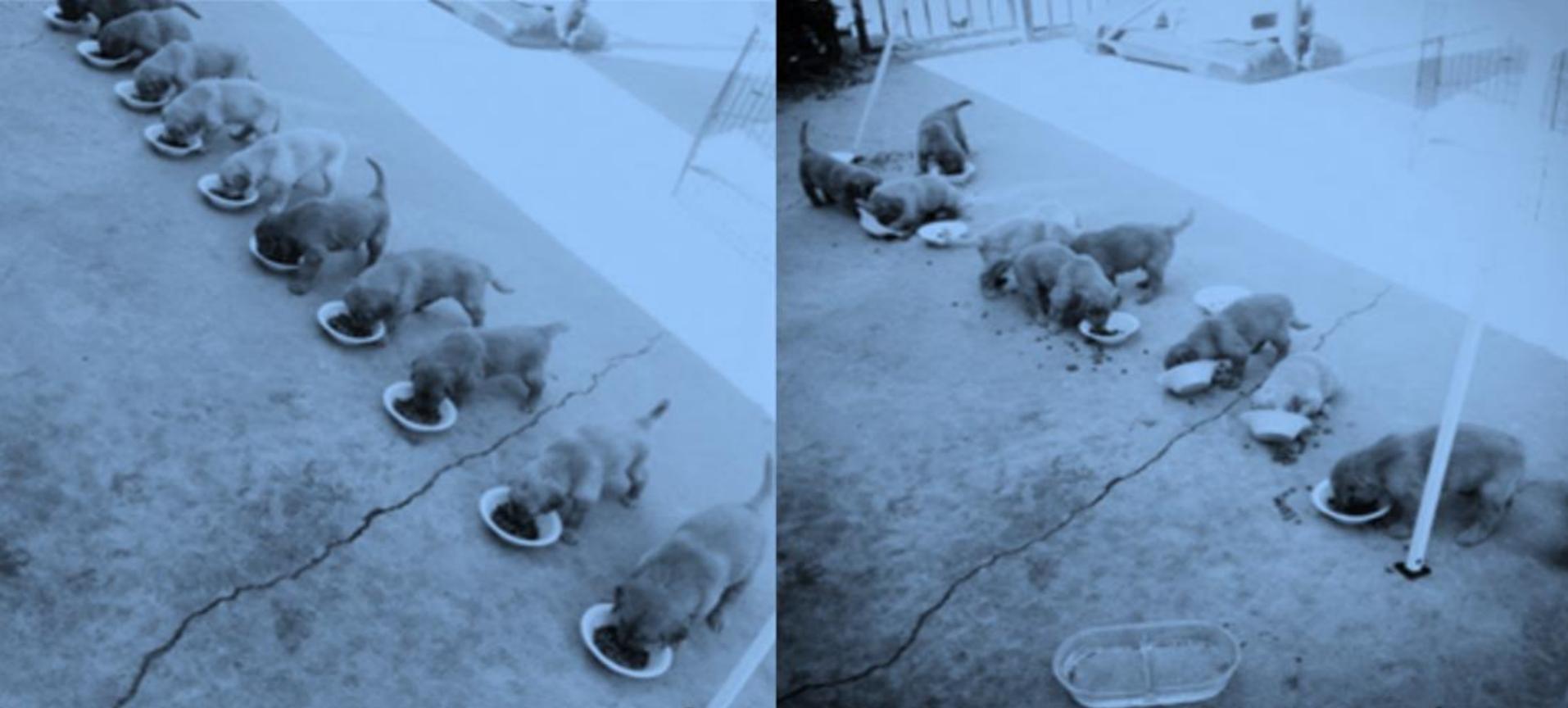
... and suddenly the balance is always 0, as desired, and the bug appears to disappear

- but it's only *masked*, **not** truly fixed: a change in timing somewhere else in the program may suddenly make the bug come back
- Concurrency is hard to get right
  - but, we'll see principled ways to design concurrent programs so that they are safer from these kinds of bugs *next week* !

# Thank you for your attention !

---

- In this lecture, you have learned:
  - to make your data structure can be compared in multiple ways
  - to create a resizing hash table that enables you to add and search data in constant average time
  - to create good hash codes
  - about concurrency, shared-memory and message-passing paradigms, processes and threads, and race conditions
- Please continue to Lecture Quiz 12 and Lab 12:
  - to do Lab Exercise 12.1 - 12.4, and
  - to do Exercise 12.1 - 12.3



# Advanced Object-Oriented Programming

CPT204 – Lecture 13  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming

### Lecture 13

**Priority Queue, Thread Safety,  
Locks, Synchronization**

# Welcome !

---

- Welcome to Lecture 13 !
- In this lecture we are going to
  - learn about priority queues
  - learn about making variables threadsafe
  - learn to use a lock to protect shared mutable data
  - learn to recognize deadlock and know strategies to prevent it

# Part 1 : Priority Queue

---

- Priority queue is an abstract data structure that supports the operations :
  - add an item: `add(T item)`
  - delete smallest item: `T delMin()`
- For example:

```
add("A")
add("B")
add("C")
delMin()      →      "A"
add("D")
delMin()      →      "B"
```

# Application Example

- Problem : Find the *largest M* items in a stream `InputStream` of  $N$  items
  - Fraud detection : isolate transactions
  - File maintenance : find biggest files or directories
- Constraint : Not enough memory to store  $N$  items

$N$  huge,  $M$  large

```
MinPQ<Transaction> pq = new MinPQ<Transaction>();
while (InputStream.hasNextLine()) {
    String line = InputStream.readLine();
    Transaction item = new Transaction(line);
    pq.add(item);
    if (pq.size() > M)
        pq.delMin();
}
```

# Applications of Priority Queues

---

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

# MinPQ Interface

- Minimum Priority Queue supports fast removal of the smallest item in the data structure and fast item addition
- To have a notion of smallest, either we implement the item to be comparable, or explicitly specify the priority of an item with a comparable value, for example an integer
- Interface MinPQ with comparable items:

Lab 13

your course project  
Part A : Explicit  
Minimum Priority  
Queue ExpMinPQ

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

# Implementing MinPQ

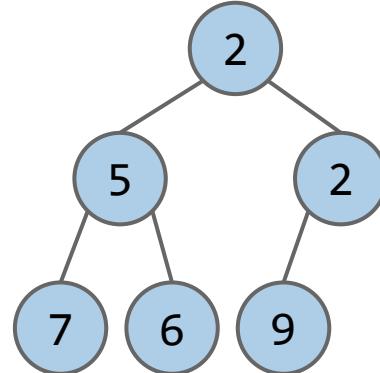
---

- Implemented naively using an **array**, we can achieve constant-time item addition but linear-time smallest item removal
  - On the other hand, using an **ordered array**, constant-time removal but linear-time addition
- Our goal is then to have an implementation that can achieve fast running-time for *both* operations
  - Binary heap is suitable for this

# Binary Heap

---

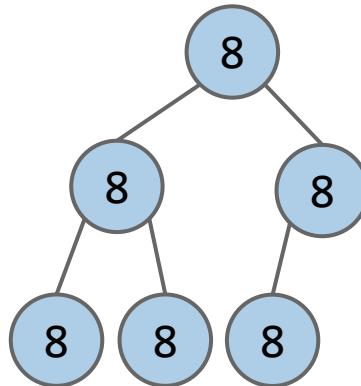
- **Binary heap** is a *binary tree* that is complete and satisfies the heap property :
  - **Complete**: balanced binary tree, could be left-aligned at the bottom level
  - **Heap property**: parent's item/priority is smaller or equal to the children's
- For example :



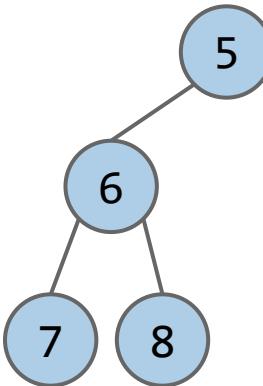
# In-Class Quiz 1

---

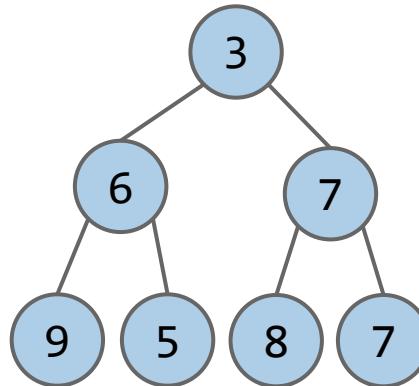
- Which of the following are binary heaps ?



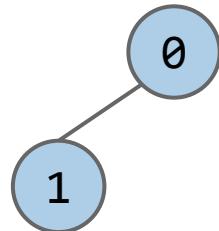
(i)



(ii)



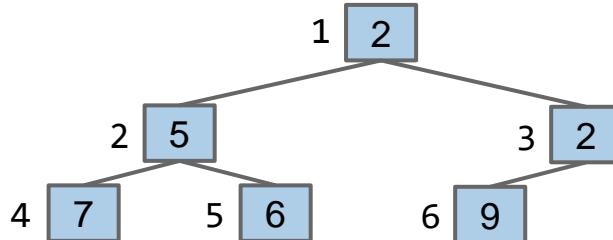
(iii)



(iv)

# Binary Heap Representation

- If we represent the binary heap by an array or an ArrayList, set the indices to start at 1, and order the nodes containing the item and the priority by the heap-property :
  - there is no explicit links needed between parent and the children
  - we can use the indices to access an item's parent or children
  - parent of an item at index  $k$  is  $k/2$
  - children of an item at index  $k$  is  $2*k$  and  $2*k + 1$
- For example:



T[] heap

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| - | 2 | 5 | 2 | 7 | 6 | 9 |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

size = 6  
capacity = 7  
heap.length = 8

# Comparable Items and Swap

---

- Since the items are comparable, we can create a helper method to decide whether an item at index  $i$  is *greater than* the item at index  $j$

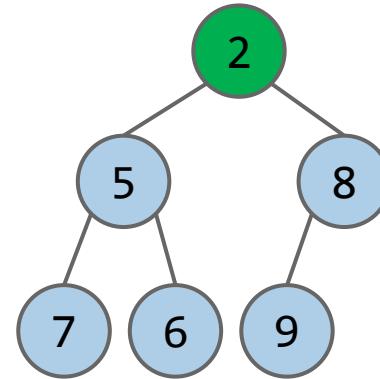
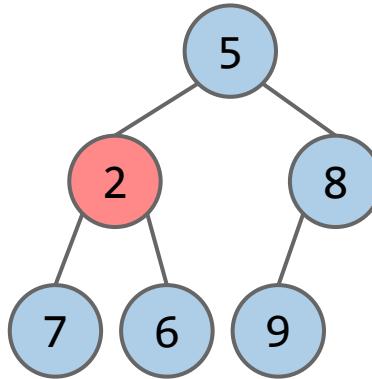
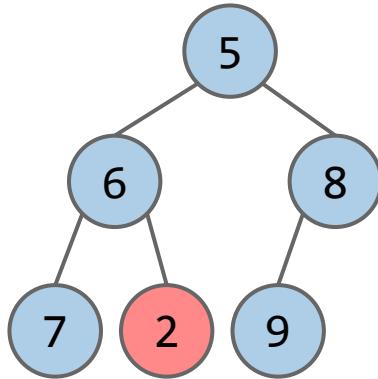
```
private boolean greater(int i, int j) {  
    return ((Comparable<T>) heap[i]).compareTo(heap[j]) > 0;  
}
```

- In the next slides, we will implement operations that swap (the reference to) the items

```
private void swap(int i, int j) {  
    T temp = heap[i];  
    heap[i] = heap[j];  
    heap[j] = temp;  
}
```

# Restoring Heap Property : Swim

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



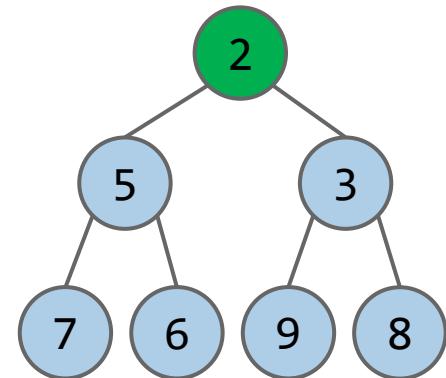
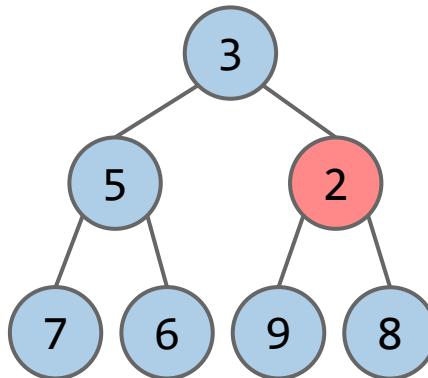
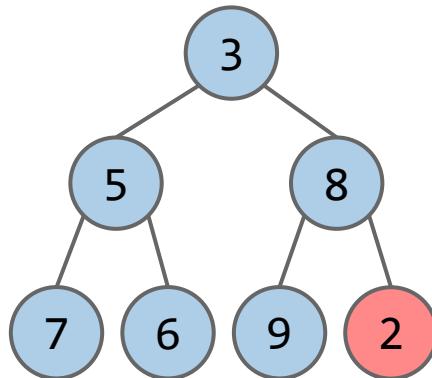
```
private void swim(int k) {  
    while (k > 1 && greater(k/2, k)) {  
        swap(k, k/2);  
        k = k/2;  
    }  
}
```

swap child with the parent,  
until heap property is restored

# Item Addition

---

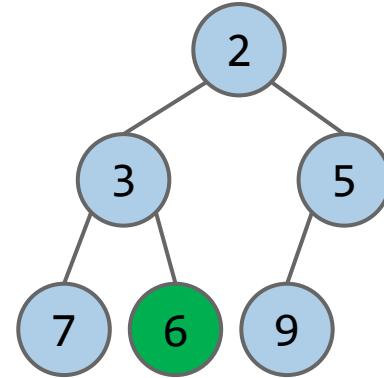
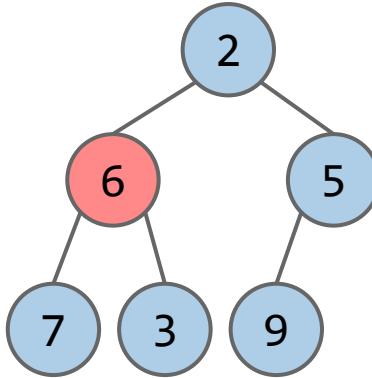
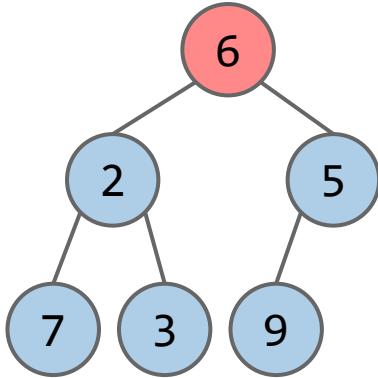
- Item addition:
  - add the new item at the end
  - then swim it : swap it with its parent repeatedly until the heap-property is restored



implement in Lab 13

# Restoring Heap Property : Sink

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

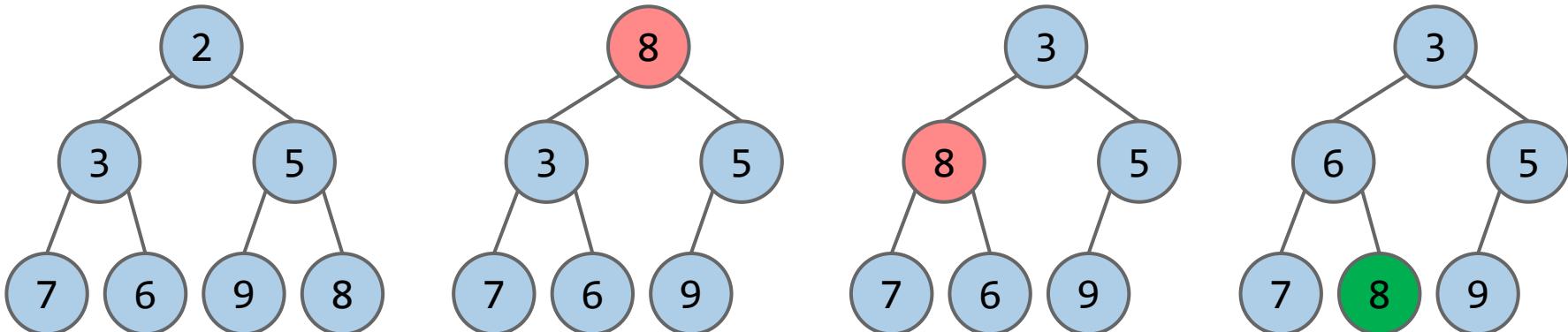


```
private void sink(int k) {  
    while (2*k <= size) {  
        int j = 2*k;  
        if (j < size && greater(j, j+1)) j++;  
        if (!greater(k, j)) break;  
        swap(k, j);  
        k = j;  
    }  
}
```

swap parent with the **smaller** child,  
until heap property is restored

# Minimum Item Removal

- Delete min:
  - replace the minimum item at the root with the last item
  - then sink it : swap it with the smaller children repeatedly until the heap-property is restored



implement in Lab 13

# Running Analysis of Binary Heap

---

How many comparisons?

- Item addition:  $O(\text{height of a binary tree}) = O(\log N)$
- Minimum item removal:  $O(2 \times \text{height of a binary tree}) = O(\log N)$

| Representation | add         | delMin      |
|----------------|-------------|-------------|
| array          | $O(1)$      | $O(N)$      |
| ordered array  | $O(N)$      | $O(1)$      |
| binary heap    | $O(\log N)$ | $O(\log N)$ |

## Part 2 : Thread Safety

---

- Last week we learn about race conditions
  - multiple threads share the same mutable variable without coordinating what they're doing
  - unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations
- This week we learn how to make variable access safe in shared-memory concurrency

# Thread Safety

---

- There are basically four ways to make variable access safe in shared-memory concurrency:
  - **Confinement** : don't share variables or data between threads
  - **Immutability** : make the shared variables unre assignable or the shared data immutable
    - we've talked a lot about immutability already, but there are some additional requirements for concurrent programming that we'll talk about in this lecture
  - **Threadsafe data type** : encapsulate the shared data in an existing threadsafe data type that does the coordination for you
  - **Synchronization** : use synchronization to keep the threads from accessing shared variables or data at the same time
    - synchronization is what you need to build your own threadsafe data type

# What Threadsafe Means

---

- A data type or static method is **threadsafe** if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code
  - "*behaves correctly*" means satisfying its specification and preserving its rep invariant
  - "*regardless of how threads are executed*" means threads might be on multiple processors or timesliced on the same processor
  - "*without additional coordination*" means that the data type can't put preconditions on its caller related to timing, like "you can't call `get()` while `set()` is in progress"

# Iterator Is Not Threadsafe

---

- For example : Iterator  
It is **not threadsafe** when used with a mutable collection
  - Iterator's specification says that you can't modify a collection at the same time as you're iterating over it
    - except using the iterator's own remove method
  - That's a timing-related precondition put on the caller, and Iterator makes no guarantee to behave correctly if you violate it

## Strategy 1: Confinement (1)

---

- Thread **confinement** is a simple idea: you avoid races on *reassignable references* and *mutable data* by keeping that data confined to a single thread
  - don't give any other threads the ability to read or write the data directly
- Since shared mutable data is the root cause of a race condition, confinement solves it by ***not sharing*** the mutable data

## Strategy 1: Confinement (2)

---

- *Local variables* are always thread confined
  - a local variable is stored in the stack, and each thread has its own stack
  - there may be multiple invocations of a method running at a time (in different threads or even at different levels of a single thread's stack, if the method is recursive),
  - but each of those invocations has its own *private copy* of the variable, so the variable itself is confined
- But be careful — the variable is thread confined, but if it's an object reference, you also *need to check the object it points to*
  - if the object is mutable, then we want to check that the object is confined as well — there *can't* be references to it that are reachable from any other thread

# Confinement Example

- We will see that confinement is what makes the accesses to n, i, and result safe in code like this:

```
public class Factorial {  
    /* Computes n! and prints it on standard output.  
     * @param n must be >= 0 */  
  
    private static void computeFact(final int n) {  
        BigInteger result = BigInteger.valueOf(1);  
        for (int i = 1; i <= n; i++) {  
            System.out.println("working on fact " + n);  
            result = result.multiply(BigInteger.valueOf(i));  
        }  
        System.out.println("fact(" + n + ") = " + result);  
    }  
  
    public static void main(String[] args) {  
        new Thread(new Runnable() { // create a thread using an  
            public void run() { // anonymous Runnable  
                computeFact(99);  
            }  
        }).start();  
        computeFact(100);  
    }  
}
```

This code starts the thread for computeFact(99) with an anonymous Runnable

# In-Class Quiz 2

---

- How many threads are running when the program runs?

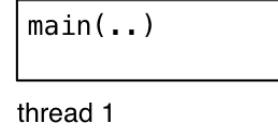
```
public class Factorial {  
    /* Computes n! and prints it on standard output.  
     * @param n must be >= 0 */  
  
    private static void computeFact(final int n) {  
        BigInteger result = BigInteger.valueOf(1);  
        for (int i = 1; i <= n; i++) {  
            System.out.println("working on fact " + n);  
            result = result.multiply(BigInteger.valueOf(i));  
        }  
        System.out.println("fact(" + n + ") = " + result);  
    }  
  
    public static void main(String[] args) {  
        new Thread(new Runnable() { // create a thread using an  
            public void run() { // anonymous Runnable  
                computeFact(99);  
            }  
        }).start();  
        computeFact(100);  
    }  
}
```

|                           |
|---------------------------|
| <input type="radio"/> 1   |
| <input type="radio"/> 2   |
| <input type="radio"/> 100 |
| <input type="radio"/> 199 |

# Confinement Example (1)

---

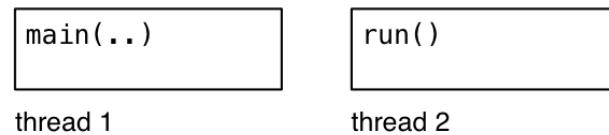
- When we start the program, we start with one thread running `main`



## Confinement Example (2)

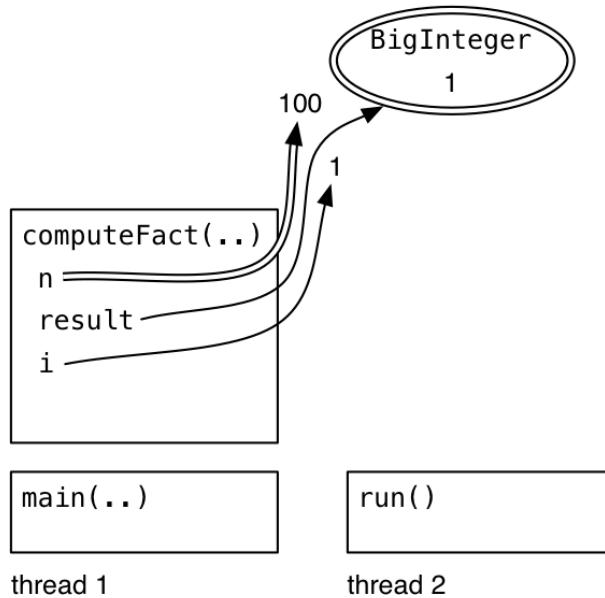
---

- When we start the program, we start with one thread running `main`
- `main` creates a second thread using the anonymous Runnable idiom, and starts that thread



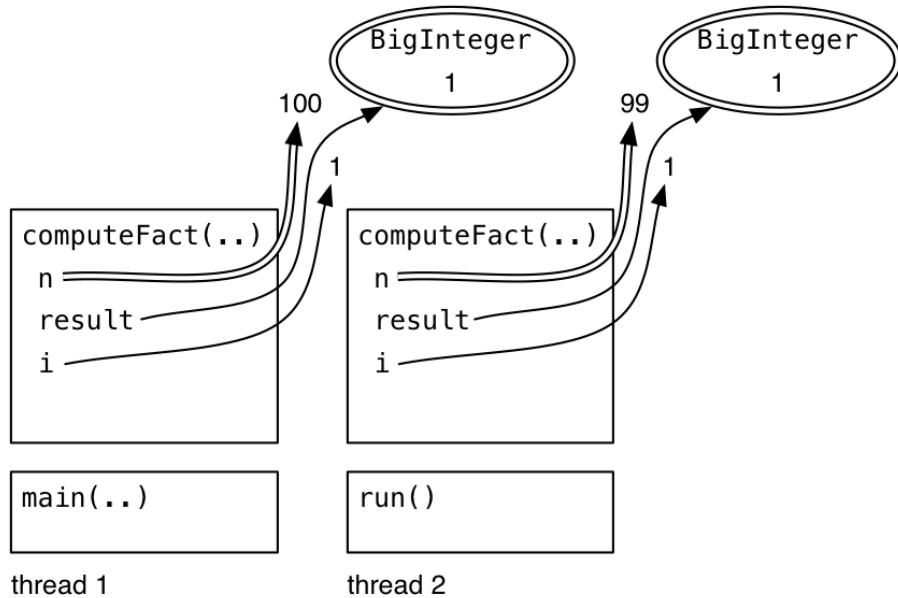
# Confinement Example (3)

- When we start the program, we start with one thread running main
- main creates a second thread using the anonymous Runnable idiom, and starts that thread
- At this point, we have two concurrent threads of execution
  - their interleaving is unknown!
  - but *one possibility* for the next thing that happens is that thread 1 enters computeFact



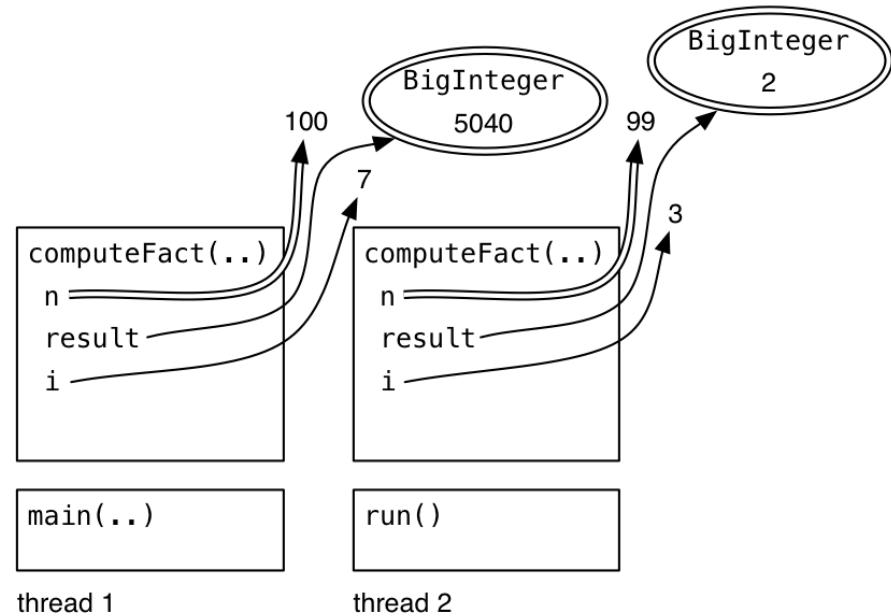
## Confinement Example (4)

- Then, the next thing that *might* happen is that thread 2 also enters computeFact
- At this point, we see how confinement helps with thread safety
  - Each execution of computeFact has *its own* n, i, and result variables, confined to that thread
  - The data they point to are also confined, and immutable
  - If the BigInteger objects were not confined – if they were aliased from multiple threads – then we would need to rely on their immutability to guarantee thread safety  
(next strategy, strategy #2)



# Confinement Example (5)

- The computeFact computations proceed **independently**, updating their respective variables



# Avoid Global Variables (1)

---

- Unlike local variables, ***static variables*** are **not automatically thread confined**
- If you have static variables in your program, then you have to make an argument that only one thread will ever use them, and you have to document that fact clearly; even better, you should eliminate the static variables entirely, for example:

```
// This class has a race condition in it.
public class PinballSimulator {
    private static PinballSimulator simulator = null;
    // invariant: there should never be more than one PinballSimulator object created
    private PinballSimulator() {
        System.out.println("created a PinballSimulator object");
    }
    // factory method that returns the sole PinballSimulator object,
    // creating it if it doesn't exist
    public static PinballSimulator getInstance() {
        if (simulator == null) {
            simulator = new PinballSimulator();
        }
        return simulator;
    }
}
```

## Avoid Global Variables (2)

---

- This class has a race in the `getInstance()` method
  - two threads could call it at the same time and end up creating two copies of the `PinballSimulator` object, which we don't want
- To fix this race using the thread confinement approach
  - you would specify that only a certain thread (maybe the “pinball simulation thread”) is allowed to call `PinballSimulator.getInstance()`

# Avoid Global Variables (3)

---

- In general, static variables are very risky for concurrency
  - They might be hiding behind a harmless function that seems to have no side-effects or mutations
  - Consider this example:

```
// is this method threadsafe?  
/**  
 * @param x integer to test for primeness; requires x > 1  
 * @return true if x is prime with high probability  
 */  
  
public static boolean isPrime(int x) {  
    if (cache.containsKey(x)) return cache.get(x);  
    boolean answer = BigInteger.valueOf(x).isProbablePrime(100);  
    cache.put(x, answer);  
    return answer;  
}  
  
private static Map<Integer, Boolean> cache = new HashMap<>();
```

# Avoid Global Variables (4)

---

- This function stores the answers from previous calls in case they're requested again
  - this technique is called **memoization**, and it's a sensible optimization for slow functions like exact primality testing
- But now the `isPrime` method is not safe to call from multiple threads, and its clients may not even realize it
- The reason is that the `HashMap` referenced by the static variable `cache` is shared by all calls to `isPrime()`, and `HashMap` is **not** threadsafe
- If multiple threads mutate the map at the same time, by calling `cache.put()`, then the map can become corrupted in the same way that the bank account became corrupted in the last lecture
- If you're lucky, the corruption may cause an exception deep in the hash map, like a `NullPointerException` or `IndexOutOfBoundsException`
  - but it also may just quietly give wrong answers, as we saw in the bank account example

## Strategy 2: Immutability (1)

---

- Our second way of achieving thread safety is by using **unre assignable references** and **immutable data types**
  - immutability tackles the shared-mutable-data cause of a race condition and solves it simply by making the shared state **not mutable**
- Final variables are unre assignable/immutable references, so a variable declared final is safe to access from multiple threads
  - you can only read the variable, not write it
  - be careful, because this safety applies only to the variable itself, and we still have to argue that the object the variable points to is immutable

## Strategy 2: Immutability (2)

---

- Immutable objects are usually also threadsafe
- We say "usually" here because our current definition of immutability is too loose for concurrent programming
  - We've said that a type is immutable if an object of the type always represents the same abstract value for its entire lifetime
  - But that actually allows the type the freedom to mutate its rep, as long as those mutations are invisible to clients
  - We saw an example of this notion when we looked at an immutable list that cached its length in a mutable field the first time the length was requested by a client
- For concurrency, though, this kind of hidden mutation is **not safe**
  - An immutable data type that uses some mutation will have to make itself threadsafe using locks (the same technique required of mutable data types we will discuss later)

# Threadsafe Immutability

---

- So in order to be confident that an immutable data type is threadsafe without locks, we need a *stronger* definition of immutability
- A type is **threadsafe immutable** if it has:
  - no mutator methods
  - all fields declared private and final
  - no representation exposure
  - no mutation whatsoever of mutable objects in the rep
- If you follow these rules, then you can be confident that your immutable type will also be threadsafe
- In Java Tutorials, read [A Strategy for Defining Immutable Objects](#)

## Strategy 3: Using Threadsafe Data Types (1)

---

- Our third major strategy for achieving thread safety is to store shared mutable data in **existing threadsafe data types**
- When a data type in the Java library is threadsafe, its documentation will *explicitly* state that fact
- For example, here's what StringBuffer says:
  - *[StringBuffer is] A threadsafe, mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.*
- StringBuffers are **safe** for use by multiple threads
  - the methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved

## Strategy 3: Using Threadsafe Data Types (2)

---

- This is in contrast to `StringBuilder`:
  - *[`StringBuilder` is] A mutable sequence of characters. This class provides an API compatible with `StringBuffer`, but with no guarantee of synchronization.*  
*This class is designed for use as a drop-in replacement for `StringBuffer` in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to `StringBuffer` as it will be faster under most implementations.*

## Strategy 3: Using Threadsafe Data Types (3)

---

- It's become common in the Java API to find two mutable data types that do the same thing, one threadsafe and the other not
  - the reason is what this quote indicates: threadsafe data types usually incur a *performance penalty* compared to an unsafe type
- It's deeply unfortunate that `StringBuffer` and `StringBuilder` are named so similarly, without any indication in the name that thread safety is the crucial difference between them
  - It's also unfortunate that they *don't share a common interface*, so you can't simply *swap* in one implementation for the other for the times when you need thread safety
  - The Java collection interfaces do much better in this respect, as we'll see next

# Threading Collections (1)

---

- The collection interfaces in Java — List, Set, Map — have basic implementations that are **not** threadsafe
  - The implementations of these that you've been used to using, namely ArrayList, HashMap, and HashSet, cannot be used safely from more than one thread
- Fortunately, just like the Collections API provides wrapper methods that make collections immutable, it provides another set of wrapper methods to make collections threadsafe, while ***still mutable***
- These wrappers effectively make each method of the collection *atomic* with respect to the other methods
  - an **atomic action** effectively happens all at once — it *doesn't interleave* its internal operations with those of other actions, and none of the effects of the action are visible to other threads until the entire action is complete, so it *never looks partially done*

# Threading Collections (2)

---

- Now we see a way to fix the `isPrime()` method we had earlier in the slides:

```
private static Map<Integer, Boolean> cache =  
    Collections.synchronizedMap(new HashMap<>());
```

- Don't circumvent the wrapper**

- make sure to *throw away references* to the underlying non-threadsafe collection, and access it only through the synchronized wrapper
- that happens automatically in the line of code above, since the new `HashMap` is passed only to `synchronizedMap()` and never stored anywhere else
- we saw this same warning with the unmodifiable wrappers: the underlying collection is still mutable, and code with a reference to it can circumvent immutability

# Threading Collections (3)

---

- Iterators are still not threadsafe
  - Even though method calls on the collection itself (`get()`, `put()`, `add()`, etc.) are now *threadsafe*, iterators created from the collection are still *not threadsafe*
  - So you **can't** use `iterator()`,
  - or the enhanced for loop syntax:

```
for (String s: list) { ... } // not threadsafe
```

even if `list` is a synchronized list wrapper
- The solution to this iteration problem will be to *acquire the collection's lock* when you need to iterate over it, which we'll talk about in the next strategy

## Threading Collections (4)

---

- Finally, **atomic operations aren't enough to prevent races**: the way that you use the synchronized collection can still have a race condition
- Consider this code, which checks whether a list has at least one element and then gets that element:

```
if ( ! list.isEmpty() ) {  
    String s = list.get(0);  
    ...  
}
```

- Even if you make `list` into a synchronized list, this code still may have a race condition, because another thread *may remove* the element ***between*** the `isEmpty()` call and the `get()` call

## Threadingsafe Collections (5)

- Even the `isPrime()` method still has potential races:

```
if (cache.containsKey(x)) return cache.get(x);  
boolean answer = BigInteger.valueOf(x).isProbablePrime(100);  
cache.put(x, answer);
```

- The synchronized map ensures that `containsKey()`, `get()`, and `put()` are now atomic, so using them from multiple threads won't damage the rep invariant of the map
  - But those three operations can now *interleave* in arbitrary ways with each other, which might break the invariant that `isPrime` needs from the cache: if the cache maps an integer  $x$  to a value  $b$ , then  $x$  is prime if and only if  $b$  is true
  - If the cache ever fails this invariant, then we might return the wrong result

## Threading Collections (6)

---

- So we have to argue that the races between `containsKey()`, `get()`, and `put()` don't threaten this invariant :
  1. Suppose `containsKey(x)` returns true, but then another thread mutates the cache before the `get(x)` call; this is not harmful because we never remove items from the cache – once it contains a result for  $x$ , it will continue to do so
  2. Alternatively, suppose `containsKey(x)` returns false, but another thread mutates the cache before `put(x, ...)`; It may end up that the two threads both test the primeness of the same  $x$  at the same time, and both will race to call `put()` with the answer; But both of them should call `put(x, answer)` with the same value for `answer`, so it doesn't matter which one wins the race – the result will be the same
- The need to make these kinds of careful arguments about safety — even when you're using threadsafe data types — is the main reason that concurrency is hard

## Strategy 4: Synchronization

---

- **Synchronization:** prevent threads from accessing the shared data at the same time
  - this is what we use to implement a threadsafe type
  - we want to learn to use synchronization to implement our own data type that is safe for shared-memory concurrency
    - the correctness of a concurrent program should not depend on accidents of timing

# Locks (1)

---

- Since race conditions caused by concurrent manipulation of shared mutable data are disastrous bugs — hard to discover, hard to reproduce, hard to debug — we need a way for concurrent modules that share memory to **synchronize** with each other
- **Locks** are one synchronization technique
  - a lock is an abstraction that *allows at most one thread to own it at a time*
  - holding a lock is how one thread tells other threads:  
"I'm working with this thing, ***don't*** touch it right now."

## Locks (2)

---

- Locks have two operations:
  - **acquire** allows a thread to take ownership of a lock
    - if a thread tries to acquire a lock currently owned by another thread, it **blocks** until the other thread releases the lock
    - at that point, it will *compete* with any other threads that are trying to acquire the lock
    - at most one thread can own the lock at a time
  - **release** relinquishes ownership of the lock, allowing another thread to take ownership of it
- Using a lock also tells the compiler and processor that you're using shared memory concurrently, so that *registers* and *caches* will be flushed out to shared storage
  - this avoids the problem of **reordering**, ensuring that the owner of a lock is always looking at up-to-date data

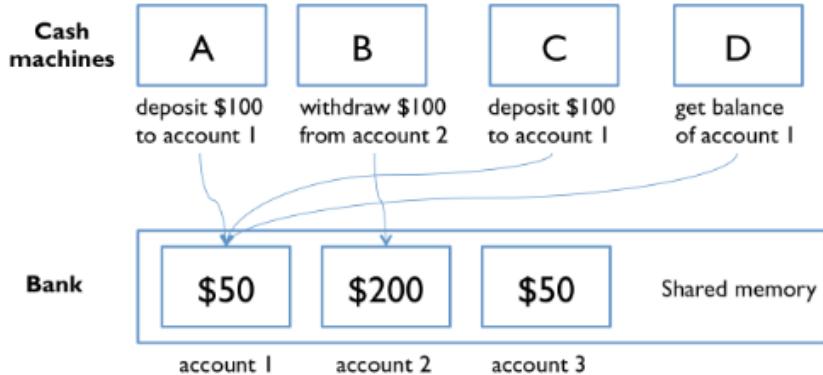
# Blocking

---

- **Blocking** means that a thread waits (without doing further work) until an event occurs
- An  $\text{acquire}(\ell)$  on thread 1 will block if another thread, say thread 2, is holding lock  $\ell$ 
  - the event it waits for is thread 2 performing  $\text{release}(\ell)$
  - at that point, if thread 1 can acquire  $\ell$ , it continues running its code, with ownership of the lock
  - it is possible that another thread, say thread 3, was also blocked on  $\text{acquire}(\ell)$
  - if so, either thread 1 or 3 (the winner is nondeterministic) will take the lock  $\ell$  and continue
  - the other will continue to block, waiting for  $\text{release}(\ell)$  again

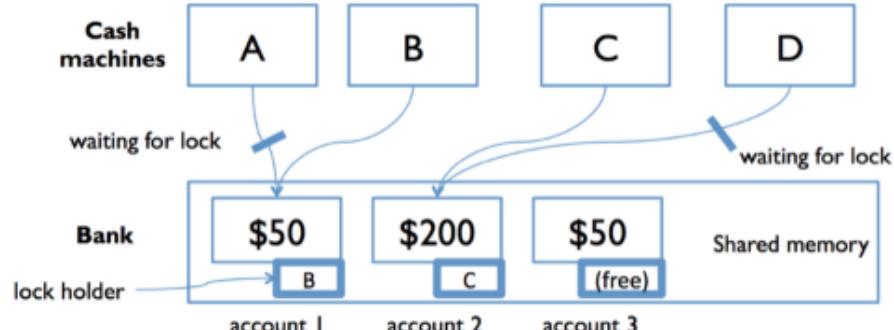
# Bank Account Example (1)

- Recall our first example of shared memory concurrency was *a bank with cash machines*
- The bank has several cash machines, all of which can read and write the same account objects in memory
- Of course, without any coordination between concurrent reads and writes to the account balances, things went horribly wrong



## Bank Account Example (2)

- To solve this problem with locks, we can add a lock that protects *each* bank account
  - now, before they can access or update an account balance, cash machines must first acquire the lock on that account
- In the figure, both A and B are trying to access account 1
  - suppose B acquires the lock first
  - then A must wait to read and write the balance until B finishes and releases the lock
  - this ensures that A and B are synchronized, but another cash machine C is able to run *independently* on a different account (because that account is protected by a different lock)



# Deadlock (1)

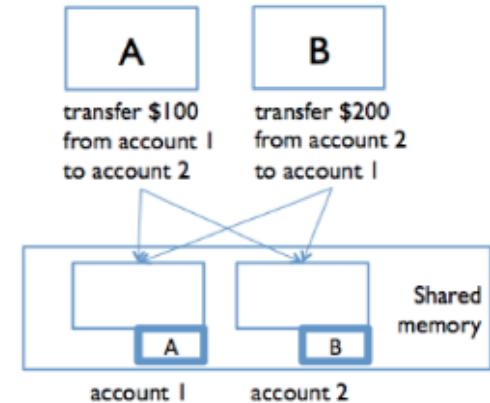
---

- When used properly and carefully, locks can prevent *race conditions*
- But then another problem rears its ugly head
  - Because the use of locks requires threads to wait (acquire blocks when another thread is holding the lock),
  - it's possible to get into a situation where two threads are waiting *for each other* — and hence ***neither can make progress***

## Deadlock (2)

---

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



## Deadlock (3)

---

- **Deadlock** occurs when concurrent modules are stuck waiting for each other to do something
- A deadlock may involve more than two modules, e.g., A may be waiting for B, which is waiting for C, which is waiting for A
  - none of them can make progress
  - the essential feature of deadlock is a **cycle of dependencies**

## Deadlock (4)

---

- You can also have deadlock without using any locks
- For example, a message-passing system, say a client and a server, can experience deadlock when message buffers fill up
  - if the client fills up the server's buffer with requests, and then blocks waiting to add another request,
  - the server may then fill up the client's buffer with results and then block itself
  - so the client is waiting for the server, and the server waiting for the client, and neither can make progress until the other one does
  - again, deadlock ensues
- In the Java Tutorials, read [Deadlock](#)

# Developing a Threadsafe Abstract Data Type

---

- Let's see how to use synchronization to implement a threadsafe ADT
- Suppose we're building a multi-user editor, like Google Docs, that allows multiple people to connect to it and edit it at the same time
  - We'll need a mutable data type to represent the text in the document
  - Here's the interface; basically it represents a **string with insert and delete operations**:

```
/** An EditBuffer represents a threadsafe mutable
 * string of characters in a text editor. */
public interface EditBuffer {
    /**
     * Modifies this by inserting a string.
     * @param pos position to insert at
     *           (requires 0 <= pos <= current buffer length)
     * @param ins string to insert
     */
    public void insert(int pos, String ins);
```

# EditBuffer Interface

```
/*
 * Modifies this by deleting a substring
 * @param pos starting position of substring to delete
 *           (requires 0 <= pos <= current buffer length)
 * @param len length of substring to delete
 *           (requires 0 <= len <= current buffer length - pos)
 */
public void delete(int pos, int len);

/**
 * @return length of text sequence in this edit buffer
 */
public int length();

/**
 * @return content of this edit buffer
 */
public String toString();
}
```

# String Rep

---

- A very simple rep for this data type would just be *a string*:

```
public class SimpleBuffer implements EditBuffer {  
    private String text;  
    // Rep invariant:  
    //   text != null  
    // Abstraction function:  
    //   represents the sequence text[0],...,text[text.length()-1]
```

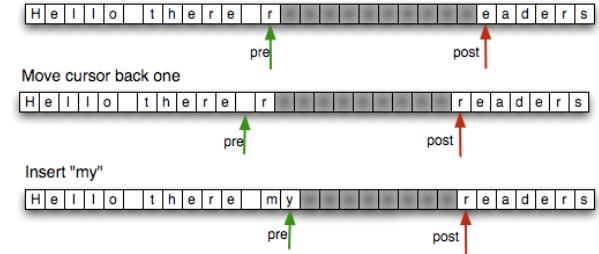
# Character Array Rep

---

- The downside of this rep is that every time we do an insert or delete, we have to copy the entire string into a new string
  - that gets expensive
- Another rep we could use would be **a character array**, with space at the end
  - that's fine if the user is just typing new text at the *end* of the document (we don't have to copy anything),
  - but if the user is typing at the *beginning* of the document, then we're copying the entire document with every keystroke

# Gap Buffer Rep (1)

- A more interesting rep, which is used by many text editors in practice, is called a **gap buffer**
  - It is a character array with extra space in it, but instead of having all the extra space at the end, the extra space is a gap that can appear anywhere in the buffer
  - Whenever an insert or delete operation needs to be done, the datatype first moves the gap to the location of the operation, and then does the insert or delete
  - If the gap is already there, then nothing needs to be copied — an insert just consumes part of the gap, and a delete just enlarges the gap!
  - Gap buffers are particularly well-suited to representing a string that is being edited by a user with a cursor, since inserts and deletes tend to be focused around the cursor, so the gap rarely moves



## Gap Buffer Rep (2)

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

- In a multi user scenario, we would want multiple gaps
  - one for each user's cursor, but we'll use a single gap for now

# Steps to Develop The Data Type (1)

---

Recall our recipe for designing and implementing an ADT:

1. **Specify**: define the operations (method signatures and specs)
  - o We did that in the EditBuffer interface
2. **Test**: develop test cases for the operations
  - o Build a test suite with a testing strategy based on partitioning the parameter space of the operations
3. **Rep**: choose a rep
  - o We chose two of them for EditBuffer, and this is often a good idea :

# Steps to Develop The Data Type (2)

---

## a) Implement a simple, brute-force rep first

- It's easier to write, you're more likely to get it right, and it will *validate your test cases* and *your specification* so you can fix problems in them before you move on to the harder implementation
- This is why we implemented SimpleBuffer before moving on to GapBuffer
- Don't throw away your simple version, either — keep it around so that you have something to test and compare against in case things go wrong with the more complex one

## Steps to Develop The Data Type (3)

---

- b) Write down the rep invariant and abstraction function, and implement checkRep()
  - o checkRep() asserts the rep invariant at the end of every constructor, producer, and mutator method
  - o It's typically *not necessary* to call it at the end of *an observer*, since the rep hasn't changed
  - o In fact, assertions can be very useful for testing complex implementations, so it's not a bad idea to also assert the postcondition at the end of a complex method

## Steps to Develop The Data Type (4)

---

- In all these steps, we're working entirely **single-threaded** at first
  - *Multithreaded* clients should be in the back of our minds at all times while we're writing specs and choosing reps
    - we'll see later that careful choice of operations may be necessary to avoid race conditions in the clients of your datatype
  - But get it working, and thoroughly tested, in a *sequential, single-threaded* environment first

## Steps to Develop The Data Type (5)

---

Now we're ready for the next step:

4. **Synchronize**: make an argument that your rep is threadsafe
  - o write it down explicitly as a comment in your class, right by the rep invariant, so that a maintainer knows how you designed thread safety into the class
  - o this part of the lecture is about how to do step 4

# Steps to Develop The Data Type (6)

---

And then the extra step we hinted at above:

## 5. Iterate

- You may find that your choice of operations makes it hard to write a threadsafe type with the guarantees clients require
- You might discover this in step 1, or in step 2 when you write tests, or in steps 3 or 4 when you implement
- If that's the case, go back and refine the set of operations your ADT provides

# Locking (1)

---

- **Locks** are so commonly-used that Java provides them as *a built-in language feature*
- In Java, *every* object has a lock implicitly associated with it — a String, an array, an ArrayList, and every class you create, all of their object instances have a lock
  - Even a humble Object has a lock, so bare Objects are often used for explicit locking:

```
Object lock = new Object();
```

## Locking (2)

---

- You *can't* call acquire and release on Java's intrinsic locks, however
- Instead you use the **synchronized** statement to *acquire* the lock **for the duration of a statement block**:

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

- Synchronized regions like this provide **mutual exclusion**: only one thread at a time can be in a synchronized region guarded by a given object's lock
  - in other words, you are back in sequential programming world, with only one thread running at a time, at least with respect to other synchronized regions that refer to the same object

# Locks Guard Access to Data (1)

---

- Locks are used to **guard** a shared data variable, like the balance shown in the previous slide
  - if all accesses to a data variable are guarded (surrounded by a synchronized block) by the same lock object, then those accesses will be guaranteed to be *atomic* — uninterrupted by other threads
- Locks only provide mutual exclusion with other threads that acquire the **same** lock
  - all accesses to a data variable must be guarded by the same lock
  - you might guard an entire collection of variables behind a single lock, but all modules must agree on which lock they will all acquire and release
- Because every object in Java has a lock implicitly associated with it, you might think that simply owning an object's lock would prevent other threads from accessing that object
  - that is **not** the case

## Locks Guard Access to Data (2)

---

- When a thread t acquires an object's lock using `synchronized (obj) { ... }` it does one thing and one thing only:  
prevents other threads from entering **their own synchronized(exp) block**,  
where exp refers to the same object as obj
  - until thread t finishes its synchronized block
  - that's it
- Even while t is in its synchronized block, another thread can dangerously mutate the object,
  - simply by neglecting to use synchronized itself
- In order to use an object lock for synchronization,
  - you have to **explicitly** and carefully guard every such access with an appropriate synchronized block

# Monitor Pattern (1)

---

- When you are writing methods of a class, the most convenient lock is the object instance itself, i.e. `this`; as a simple approach, we can guard the entire rep of a class by wrapping all accesses to the rep inside `synchronized (this)`

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

## Monitor Pattern (2)

---

```
public void delete(int pos, int len) {  
    synchronized (this) {  
        text = text.substring(0, pos) + text.substring(pos+len);  
        checkRep();  
    }  
}  
public int length() {  
    synchronized (this) {  
        return text.length();  
    }  
}  
public String toString() {  
    synchronized (this) {  
        return text;  
    }  
}  
}
```

# Monitor Pattern (3)

---

- Note the very careful discipline here
  - *every* method is guarded with the lock — even apparently small and trivial ones like `length()` and `toString()`
  - this is because reads must be guarded as well as writes — if reads are left unguarded, then they may be able to see the rep in a partially-modified state
- This approach is called **the monitor pattern**
  - A monitor is a class whose methods are mutually exclusive, so that only one thread can be inside an instance of the class at a time
- Java provides some syntactic sugar for the monitor pattern
  - If you add the keyword **synchronized** to *a method signature*, then Java will act *as if* you wrote `synchronized (this)` around the method body

## Monitor Pattern (4)

---

- So the code below is an equivalent way to implement the synchronized SimpleBuffer:

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

# Monitor Pattern (5)

```
public synchronized void delete(int pos, int len) {  
    text = text.substring(0, pos) + text.substring(pos+len);  
    checkRep();  
}  
  
public synchronized int length() {  
    return text.length();  
}  
  
public synchronized String toString() {  
    return text;  
}  
}
```

- Notice that the SimpleBuffer constructor doesn't have a synchronized keyword
- Java actually forbids it, syntactically, because an object under construction is expected to be confined to a single thread until it has returned from its constructor
  - so synchronizing constructors should be *unnecessary*

# Monitor Pattern (6)

---

In the Java Tutorials, read:

- [Synchronized Methods](#)
- [Intrinsic Locks and Synchronization](#)

# Thread Safety Argument with Synchronization (1)

- Now that we're protecting SimpleBuffer's rep with a lock, we can write a better thread safety argument:

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    // Rep invariant:
    //   text != null
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
    // Thread safety argument:
    //   all accesses to text happen within SimpleBuffer methods,
    //   which are all guarded by SimpleBuffer's lock
```

## Thread Safety Argument with Synchronization (2)

---

- The same argument works for GapBuffer, if we use the monitor pattern to synchronize all its methods
- Note that the encapsulation of the class, the absence of rep exposure, is very important for making this argument
  - If text were public:

```
public String text;
```

then clients outside SimpleBuffer would be able to read and write it without knowing that they should first acquire the lock,  
and SimpleBuffer would *no longer be threadsafe*

# Locking Discipline

---

- A locking discipline is a strategy for ensuring that synchronized code is threadsafe
- We must satisfy two conditions:
  1. Every shared mutable variable must be guarded by some lock
    - The data may not be read or written except inside a synchronized block that acquires that lock
  2. If an invariant involves multiple shared mutable variables (which might even be in different objects), then all the variables involved must be guarded by the same lock
    - Once a thread acquires the lock, the invariant must be reestablished before releasing the lock
- The monitor pattern as used here satisfies both rules
  - All the shared mutable data in the rep — which the rep invariant depends on — are guarded by the same lock

# Atomic Operations (1)

- Consider a find-and-replace operation on the EditBuffer datatype:

```
/** Modifies buf by replacing the first occurrence of s with t.  
 * If s not found in buf, then has no effect.  
 * @returns true if and only if a replacement was made  
 */  
  
public static boolean findReplace(EditBuffer buf, String s, String t) {  
    int i = buf.toString().indexOf(s);  
    if (i == -1) {  
        return false;  
    }  
    buf.delete(i, s.length());  
    buf.insert(i, t);  
    return true;  
}
```

notice that `findReplace` is a static method

## Atomic Operations (2)

---

- This method makes three different calls to buf — to convert it to a string in order to search for s, to delete the old text, and then to insert t in its place
  - Even though each of these calls individually is atomic, the `findReplace` method as a whole is **not** threadsafe,
  - because other threads might mutate the buffer while `findReplace` is working, causing it to delete the wrong region or put the replacement back in the wrong place
- To prevent this, `findReplace` needs to synchronize with all other clients of buf

# Giving clients access to a lock (1)

---

- It's sometimes useful to make your datatype's lock available to clients, so that they can use it to implement higher-level atomic operations using your datatype
- So one approach to the problem with `findReplace` is to document that clients can use the `EditBuffer`'s lock to synchronize with each other:

```
/** An EditBuffer represents a threadsafe mutable string of characters
 * in a text editor. Clients may synchronize with each other using the
 * EditBuffer object itself. */
public interface EditBuffer {  
    ...  
}
```

## Giving clients access to a lock (2)

- And then `findReplace` can synchronize on `buf`:

pass the corresponding  
EditBuffer object-instance

```
public static boolean findReplace(EditBuffer buf, String s, String t) {  
    synchronized (buf) {  
        int i = buf.toString().indexOf(s);  
        if (i == -1) {  
            return false;  
        }  
        buf.delete(i, s.length());  
        buf.insert(i, t);  
        return true;  
    }  
}
```

- The effect of this is to **enlarge** the synchronization region that the monitor pattern already put around the individual `toString`, `delete`, and `insert` methods, into a single atomic region that ensures that all three methods are executed without interference from other threads

# Sprinkling synchronized everywhere? (1)

---

- So is thread safety simply a matter of putting the synchronized keyword on every method in your program?
  - Unfortunately **not**
- First, you actually don't want to synchronize methods haphazardly
  - synchronization imposes **a large cost** on your program
  - making a synchronized method call may take significantly longer, because of the need to acquire a lock (and flush caches and communicate with other processors)
  - Java leaves many of its mutable data types unsynchronized by default exactly for these performance reasons
  - *when you don't need synchronization, don't use it*

## Sprinkling synchronized everywhere? (2)

---

- Second, it's not actually sufficient to add synchronized everywhere
  - dropping synchronized onto a method without thinking means that you're acquiring a lock without thinking about which lock it is, or about whether it's the right lock for guarding the shared data access you're about to do
  - suppose we had tried to solve `findReplace`'s synchronization problem simply by dropping synchronized onto its declaration:

```
public static synchronized boolean findReplace(EditBuffer buf, ... ) {
```

## Sprinkling synchronized everywhere? (3)

---

- This **wouldn't** do what we want
  - it would indeed acquire a lock — because `findReplace` is a **static method**, it would acquire a **static lock for the whole class** that `findReplace` happens to be in, rather than an instance object lock
  - as a result, *only one thread could call findReplace at a time — even if other threads want to operate on different buffers*, which should be safe, they'd still be blocked until the single lock was free
  - so we'd suffer a significant loss in performance, because only one user of our massive multiuser editor would be allowed to do a find-and-replace at a time, even if they're all editing different documents

## Sprinkling synchronized everywhere? (4)

---

- Worse, however, it wouldn't provide useful protection, because other code that touches the document probably **wouldn't be acquiring the same lock**
  - it **wouldn't** actually eliminate our race conditions
- The synchronized keyword is *not a panacea*
  - thread safety requires a discipline — using confinement, immutability, or locks to protect shared data
  - and that discipline needs to be written down, or maintainers won't know what it is

# Designing a Data Type for Concurrency (1)

---

- `findReplace`'s problem can be interpreted another way: that the `EditBuffer` interface really isn't that friendly to multiple simultaneous clients
  - it relies on integer indexes to specify insert and delete locations, which are extremely brittle to other mutations
  - if somebody else inserts or deletes before the index position, then the index becomes invalid

## Designing a Data Type for Concurrency (2)

---

- So if we're designing a data type specifically for use in a concurrent system, we need to think about providing operations that have better-defined semantics when they are interleaved
- For example, it might be better to pair EditBuffer with a Position data type representing a cursor position in the buffer, or even a Selection data type representing a selected range
- Once obtained, a Position could hold its location in the text against the wash of insertions and deletions around it, until the client was ready to use that Position
- If some other thread deleted all the text around the Position, then the Position would be able to inform a subsequent client about what had happened (perhaps with an exception), and allow the client to decide what to do
- These kinds of considerations come into play when designing a data type for concurrency

# Designing a Data Type for Concurrency (3)

---

- As another example, consider the ConcurrentHashMap interface in Java
  - this interface extends the existing Map interface, adding a few key methods that are commonly needed as atomic operations on a shared mutable map, e.g.:
  - `map.putIfAbsent(key, value)` is an atomic version of

```
if ( ! map.containsKey(key)) map.put(key, value);
```
  - `map.replace(key, value)` is an atomic version of

```
if (map.containsKey(key)) map.put(key, value);
```

# Deadlock rears its ugly head (1)

---

- The synchronization approach to thread safety is powerful, but (unlike confinement and immutability) it introduces blocking into the program
  - threads must sometimes wait for other threads to get out of synchronized regions before they can proceed
  - and blocking raises the possibility of deadlock
- As we saw in the previous few slides before, deadlock happens when threads acquire multiple locks at the same time, and two threads end up blocked while holding locks that they are each waiting for the other to release
  - the monitor pattern *unfortunately makes this fairly easy to do*

# Deadlock rears its ugly head (2)

- Here's an example: suppose we're modeling the social network in Hogwarts:

```
public class Wizard {  
    private final String name;  
    private final Set<Wizard> friends;  
    // Rep invariant:  
    //     friend links are bidirectional:  
    //     for all f in friends, f.friends contains this  
    // Thread safety argument:  
    //     threadsafe by monitor pattern: all accesses to rep  
    //     are guarded by this object's lock  
  
    public Wizard(String name) {  
        this.name = name;  
        this.friends = new HashSet<Wizard>();  
    }  
}
```



## Deadlock rears its ugly head (3)

---

```
public synchronized boolean isFriendsWith(Wizard that) {  
    return this.friends.contains(that);  
}  
  
public synchronized void friend(Wizard that) {  
    if (friends.add(that)) {  
        that.friend(this);  
    }  
}  
  
public synchronized void defriend(Wizard that) {  
    if (friends.remove(that)) {  
        that.defriend(this);  
    }  
}
```

## Deadlock rears its ugly head (4)



- Like Facebook, this social network is *bidirectional*:  
if  $x$  is friends with  $y$ , then  $y$  is friends with  $x$ 
  - the friend() and defriend() methods enforce that invariant by modifying the reps of both objects,
  - which because they use the monitor pattern means acquiring the locks to both objects as well
- Let's create a couple of wizards:

```
Wizard harry = new Wizard("Harry Potter");
Wizard snape = new Wizard("Severus Snape");
```

# In-Class Quiz 3

---

- And then think about what happens when two independent threads are repeatedly running:

```
// thread A                                // thread B
while(true) {
    harry.friend(snape);
    harry.defriend(snape);
}
while(true) {
    snape.friend(harry);
    snape.defriend(harry);
}
```

- Deadlock will happen when:
  - Thread A acquires lock on harry, Thread B acquires lock on snape
  - Thread A acquires lock on snape, Thread B acquires lock on harry
  - Thread A acquires lock on harry, Thread B acquires lock on harry
  - Thread A acquires lock on snape, Thread B acquires lock on snape

# Deadlock rears its ugly head (5)

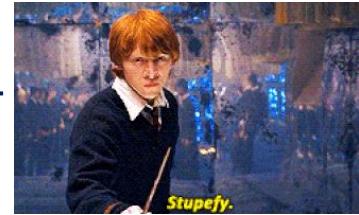
- And then think about what happens when two independent threads are repeatedly running:

```
// thread A                                // thread B
while(true) {
    harry.friend(snape);
    harry.defriend(snape);
}
while(true) {
    snape.friend(harry);
    snape.defriend(harry);
}
```

- We will deadlock very rapidly :
  - Suppose thread A is about to execute `harry.friend(snape)`, and thread B is about to execute `snape.friend(harry)`
  - Thread A acquires the lock on `harry` (because the `friend` method is synchronized)
  - Then thread B acquires the lock on `snape` (for the same reason)
  - They both update their individual reps independently, and then try to call `friend()` on the other object — which requires them to acquire the lock on the other object

# Deadlock rears its ugly head (6)

---



- So A is holding Harry and waiting for Snape, and B is holding Snape and waiting for Harry
- Both threads are stuck in `friend()`, so neither one will ever manage to exit the synchronized region and release the lock to the other
- This is a classic deadly embrace: the program was *stupefied*
- The essence of the problem is acquiring multiple locks, and holding some of the locks while waiting for another lock to become free
- Notice that it is *possible* for thread A and thread B to interleave such that deadlock does not occur: perhaps thread A acquires and releases both locks before thread B has enough time to acquire the first one
  - If the locks involved in a deadlock are also involved in a race condition — and very often they are — then the deadlock will be just *as difficult to reproduce or debug*

# Deadlock Solution 1: Lock Ordering (1)

---

- One way to prevent deadlock is to put **an ordering on the locks** that need to be acquired simultaneously, and ensuring that all code acquires the locks in that order
- In our Hogwarts social network example, we might always acquire the locks on the Wizard objects in *alphabetical* order by the wizard's name
  - since thread A and thread B are both going to need the locks for Harry and Snape, they would both acquire them in that order: Harry's lock first, then Snape's
  - if thread A gets Harry's lock before B does, it will also get Snape's lock before B does, because B can't proceed until A releases Harry's lock again
  - the ordering on the locks forces an ordering on the threads acquiring them, so there's no way to produce a cycle in the waiting-for graph

# Deadlock Solution 1: Lock Ordering (2)

---

- Here's what the code might look like:

```
public void friend(Wizard that) {  
    Wizard first, second;  
    if (this.name.compareTo(that.name) < 0) {  
        first = this; second = that;  
    } else {  
        first = that; second = this;  
    }  
    synchronized (first) {  
        synchronized (second) {  
            if (friends.add(that)) {  
                that.friend(this);  
            }  
        }  
    }  
}
```

# Deadlock Solution 1: Lock Ordering (3)

---

- Note that the decision to order the locks alphabetically by the person's name would work fine for this wizarding world, but it wouldn't work in a real life social network
  - Why not? What would be better to use for lock ordering than the name?
- Although lock ordering is useful (particularly in code like operating system kernels), it has a number of drawbacks in practice
- First, it's **not modular** — the code has to know about all the locks in the system, or at least in its subsystem
- Second, it may be difficult or impossible for the code **to know exactly which of those locks it will need** before it even acquires the first one
  - it may need to do some computation to figure it out
  - think about doing a *depth-first search* on the social network graph, for example — how would you know which nodes need to be locked, before you've even started looking for them?

## Deadlock Solution 2: Coarse-grained Locking (1)

---

- A more common approach than lock ordering, particularly for application programming (as opposed to operating system or device driver programming), is to use **coarser locking** — use a single lock to guard many object instances, or even a whole subsystem of a program
- For example, we might have *a single lock* for an entire social network, and have all the operations on any of its constituent parts *synchronize on that lock*

## Deadlock Solution 2: Coarse-grained Locking (2)

---

- In the code below, all Wizards belong to a Castle, and we just use that Castle object's lock to synchronize:

```
public class Wizard {  
    private final Castle castle;  
    private final String name;  
    private final Set<Wizard> friends;  
  
    ...  
    public void friend(Wizard that) {  
        synchronized (castle) {  
            if (this.friends.add(that)) {  
                that.friend(this);  
            }  
        }  
    }  
}
```

## Deadlock Solution 2: Coarse-grained Locking (3)

---

- Coarse-grained locks can have a significant performance penalty
- If you guard a large pile of mutable data with a single lock, then you're giving up the ability to access any of that data concurrently
- In the worst case, having a single lock protecting everything, your program might be essentially *sequential* — only one thread is allowed to make progress at a time

# Goals of Concurrent Program Design (1)

---

- Building concurrent software is challenging
- We can break the issues into two general classes: when we ask whether a concurrent program is safe from bugs, we care about two properties:
  - **Safety.** Does the concurrent program satisfy its invariants and its specifications?
    - Races in accessing mutable data threaten safety  
Safety asks the question: can you prove that *some bad thing never happens?*
  - **Liveness.** Does the program keep running and eventually do what you want, or does it get stuck somewhere waiting forever for events that will never happen?
    - Can you prove that *some good thing eventually happens?*

## Goals of Concurrent Program Design (2)

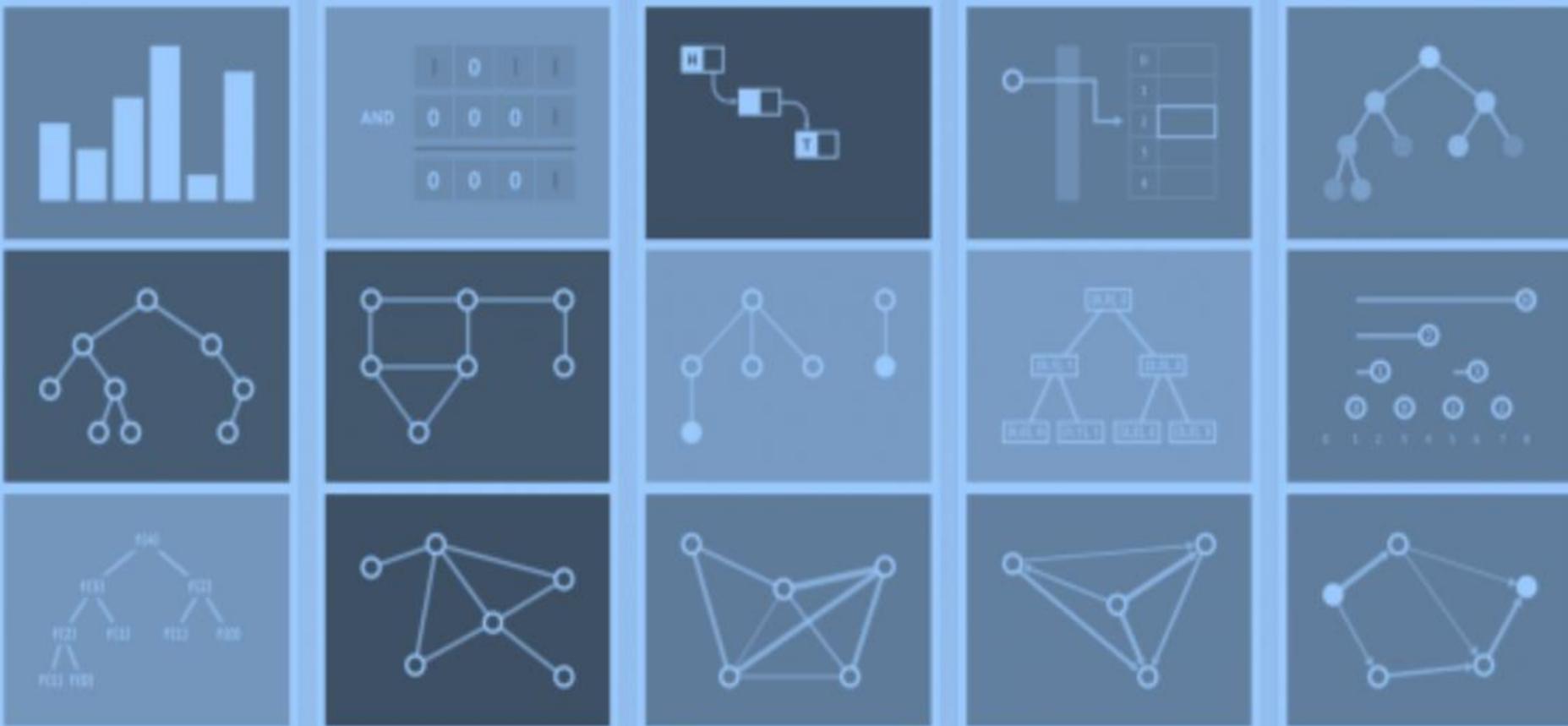
---

- Deadlocks threaten liveness
- Liveness may also require **fairness**, which means that concurrent modules are given processing capacity to make progress on their computations
- Fairness is mostly a matter for the operating system's thread scheduler, but you can influence it (for good or for ill) by setting thread priorities

# Thank you for your attention !

---

- In this lecture, you have learned:
    - about the priority queue and binary heap data structure
    - to achieve safety from race conditions on shared mutable data by confinement, immutability, threadsafe data types, and synchronization
    - how to use lock and synchronized methods
    - about deadlock and how to prevent it
  - Please continue to work on **Lecture Quiz 13, Lab 13 :**
    - to do Lab Exercise 13.1 - 13.4
- and **Lab 14 Task A, B**



# Advanced Object-Oriented Programming

CPT204 – Lecture 14  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming

### Lecture 14

# Review and Conclusion

# Welcome !

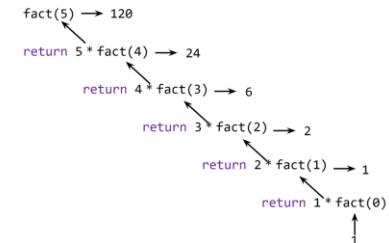
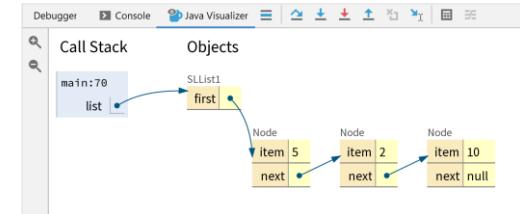
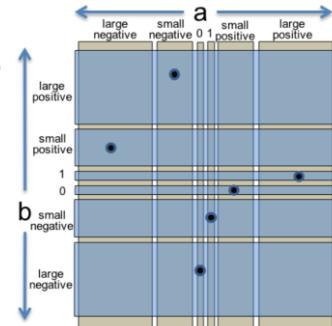
---

- Welcome to Lecture 14 !
- In this lecture we are going to
  - review the data structures
  - review the advanced object-oriented techniques that we have learned this semester
  - conclude the course :D



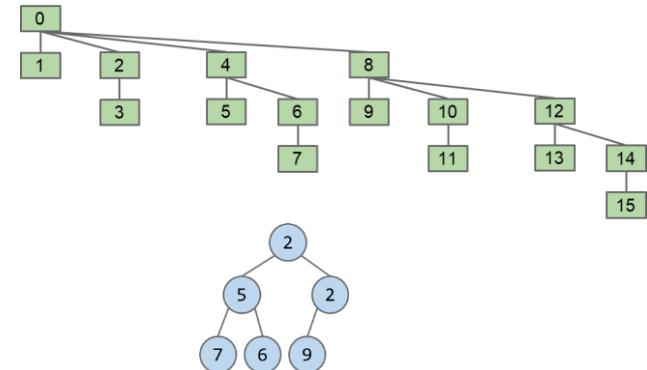
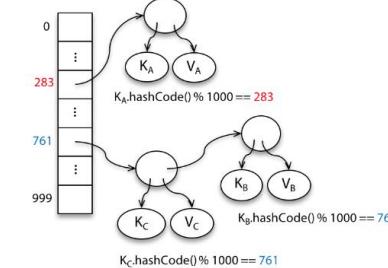
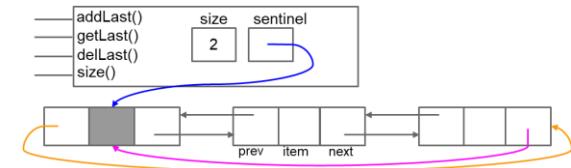
# Checking, Testing and Debugging, Problem Solving

- Checking:
  - Static checking, dynamic checking, no checking
- Testing and Debugging:
  - Unit testing, integration testing
  - Partitioning, boundary/corner cases
  - Black box, white box
  - Coverage, automated regression, visualization tools
- Problem solving:
  - Iterative
  - Recursive



# Generic Data Structures

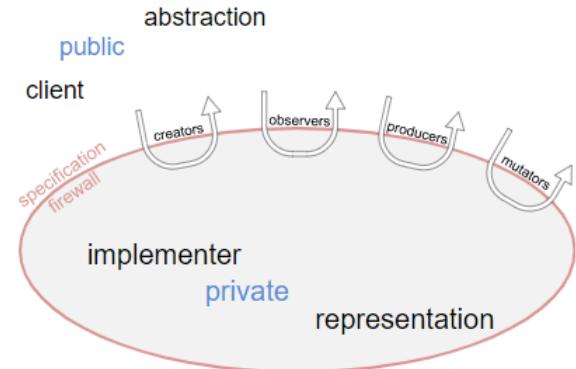
- List and Deque
  - Linked-list-based: Recursive, Singly, Doubly, Sentinel, Circular
  - Array-based: ArrayList, Circular Array, Resizing Array
- Map
  - HashMap, HAMap
- Set
  - HashSet, ARSet, HASet
- Disjoint sets
  - Weighted Quick Union
- Priority Queue
  - Binary Heap with Array, ArrayList



# Abstract Data Type (ADT)

- Defined by its operations

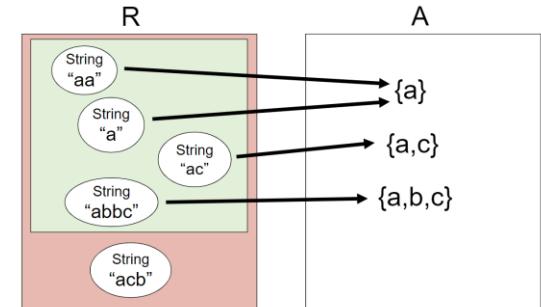
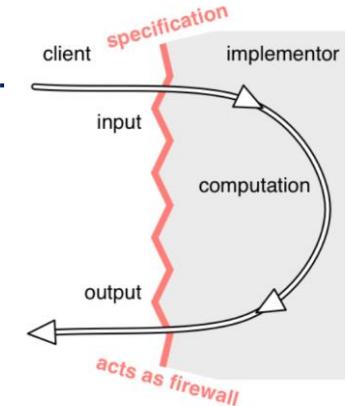
| ADT concept        | Ways to do it in Java                              | Examples                                                                                      |
|--------------------|----------------------------------------------------|-----------------------------------------------------------------------------------------------|
| Abstract data type | Class<br>Interface + class(es)<br>Enum             | <code>String</code><br><code>List</code> and <code>ArrayList</code><br><code>DayOfWeek</code> |
| Creator operation  | Constructor<br>Static (factory) method<br>Constant | <code>ArrayList()</code><br><code>List.of()</code><br><code>BigInteger.ZERO</code>            |
| Observer operation | Instance method<br>Static method                   | <code>List.get()</code><br><code>Collections.max()</code>                                     |
| Producer operation | Instance method<br>Static method                   | <code>String.trim()</code><br><code>Collections.unmodifiableList()</code>                     |
| Mutator operation  | Instance method<br>Static method                   | <code>List.add()</code><br><code>Collections.copy()</code>                                    |
| Representation     | <code>private</code> fields                        |                                                                                               |



- Representation independent, can change the representation

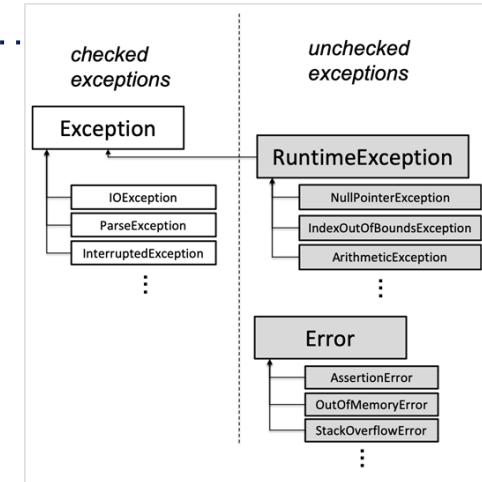
# Specifications and Representations

- Specifications
  - Method signatures, Javadoc
  - Preconditions and postconditions
    - deterministic, not undetermined
    - declarative, not operational
    - not too strong, not too weak
- Representations
  - Rep values, abstract values
  - Abstraction functions, representation invariants
  - Safe from representation exposure
    - defensive copying, deep copy, copy constructor



# Error Handling

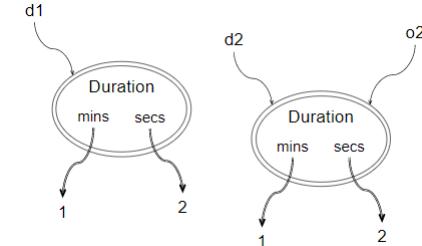
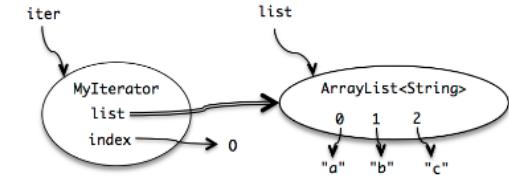
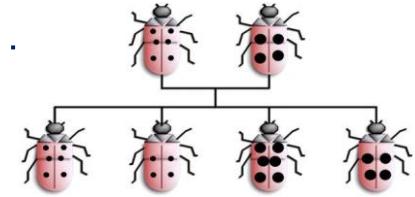
- Special values
  - null, -1
- Checked Exception
  - signal special cases, hard to prevent
- Unchecked Exception
  - signal special cases, easy to prevent
  - prevent bugs in the user input
- Assertion
  - prevent bugs in your algorithm or implementation, as you write your code
    - remove or deactivate after



```
assert x >= 0 : "x is " + x;
```

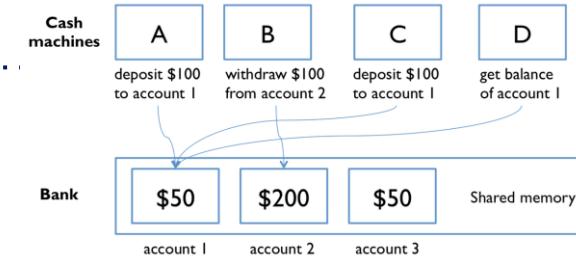
# Advanced OOP Techniques

- Immutability
- Inheritance and Polymorphism
  - Overloading, overriding
  - Dynamic method selection
  - Interface, default method
- Inner class, anonymous class
- Iterable, Iterator, enhanced for loop
- Comparable, Comparator, Equality
  - Referential, object equality
  - Observational, behavioral equality
- HashCode

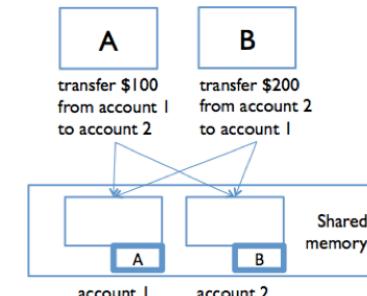


# Multithreading and Deadlock

- Thread in Java
  - Thread, start(), Runnable, run()
- Interleaving
  - Race condition, reordering
- Thread-safe
  - Confinement, Immutability
  - Threadsafe data types
  - Synchronization
    - Class/Object Lock, Mutual Exclusion, Monitor Pattern
- Deadlock
  - Lock ordering, Coarse-grained locking



| A                                   | B                                   |
|-------------------------------------|-------------------------------------|
| A get balance (balance=0)           |                                     |
|                                     | B get balance (balance=0)           |
| A add 1                             |                                     |
|                                     | B add 1                             |
| A write back the result (balance=1) |                                     |
|                                     | B write back the result (balance=1) |



# Lab Test and Final Exam using LMO Autograder

---

- The autograder may be busy and reject to grade and give feedback
  - you may try again after some time
- However, your submission is **recorded** and **not lost**
- I will regrade every submission later
  - the earlier submission will be graded

# Thank you for your attention !

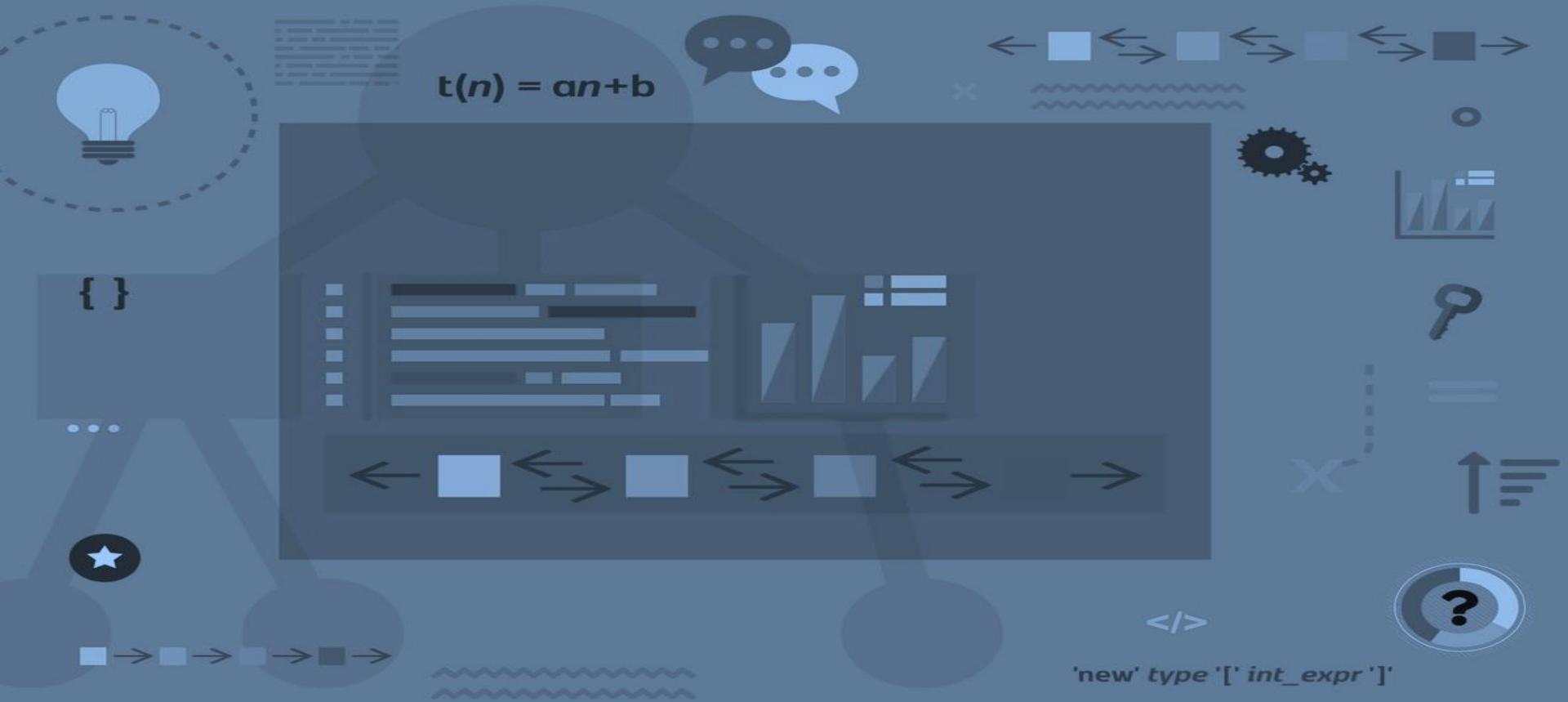
---

- Please check your assigned lab test and final exam lab seat
  - before the lab rooms are closed by MITS

- Wish you all the best for your lab test and final exam !



- Please continue to **Mock Final Exam**
  - and ask me questions



# Advanced Object-Oriented Programming

CPT204 – Lab 0  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

# CPT204 Advanced Object-Oriented Programming

## Lab 0

### Lab Setup

# Welcome !

---

- Welcome to Lab 0 !
  - Please read Lecture 0 first, if you have not done so
- In this lab, we are going to guide you to
  - A. Install Java
  - B. Install IntelliJ
  - C. Install IntelliJ Plugins
- Skip the Install Java step if you have done so
- You could use another IDE, but the one supported in the course and used in the lab will only be IntelliJ

## A. Install Java (1)

---

1. Go to <https://www.oracle.com/java/technologies/javase-downloads.html>
2. Click JDK Download

**Java SE 15**

Java SE 15.0.2 is the latest release for the Java SE Platform

- Documentation
- Installation Instructions
- Release Notes
- Oracle License
  - Binary License
  - Documentation License
- Java SE Licensing Information User Manual
  - Includes Third Party Licenses
- Certified System Configurations
- Readme

**Oracle JDK**

 [JDK Download](#)

 [Documentation Download](#)

## A. Install Java (2)

---

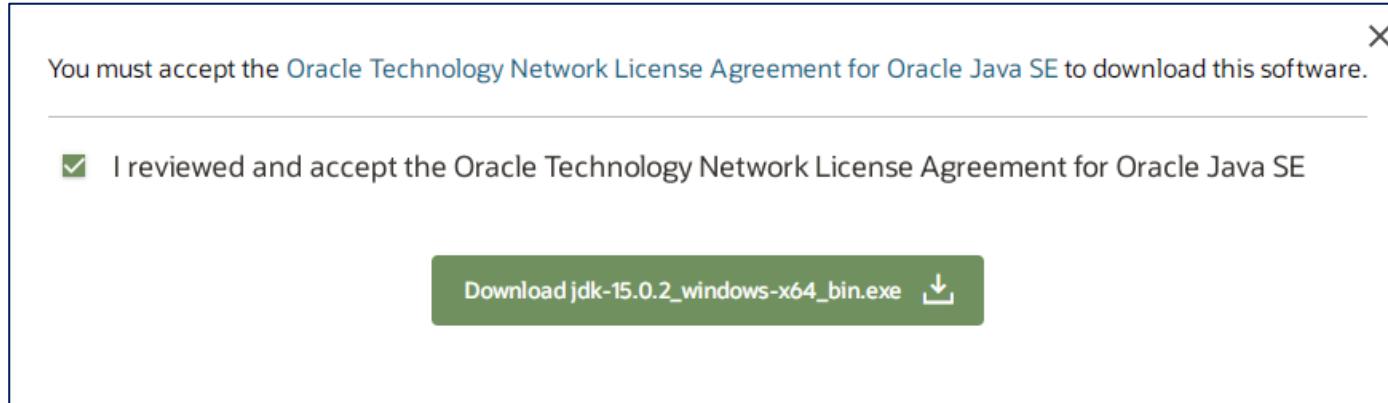
### 3. Pick the installer for your operating system

| Java SE Development Kit 15.0.2                                                                     |           |                                                                                                                                         |
|----------------------------------------------------------------------------------------------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------|
| This software is licensed under the Oracle Technology Network License Agreement for Oracle Java SE |           |                                                                                                                                         |
| Product / File Description                                                                         | File Size | Download                                                                                                                                |
| Linux ARM 64 RPM Package                                                                           | 141.82 MB |  <a href="#">jdk-15.0.2_linux-aarch64_bin.rpm</a>    |
| Linux ARM 64 Compressed Archive                                                                    | 157 MB    |  <a href="#">jdk-15.0.2_linux-aarch64_bin.tar.gz</a> |
| Linux x64 Debian Package                                                                           | 154.81 MB |  <a href="#">jdk-15.0.2_linux-x64_bin.deb</a>        |
| Linux x64 RPM Package                                                                              | 162.03 MB |  <a href="#">jdk-15.0.2_linux-x64_bin.rpm</a>        |
| Linux x64 Compressed Archive                                                                       | 179.35 MB |  <a href="#">jdk-15.0.2_linux-x64_bin.tar.gz</a>     |
| macOS Installer                                                                                    | 175.93 MB |  <a href="#">jdk-15.0.2_osx-x64_bin.dmg</a>          |
| macOS Compressed Archive                                                                           | 176.51 MB |  <a href="#">jdk-15.0.2_osx-x64_bin.tar.gz</a>       |
| Windows x64 Installer                                                                              | 159.71 MB |  <a href="#">jdk-15.0.2_windows-x64_bin.exe</a>      |
| Windows x64 Compressed Archive                                                                     | 179.28 MB |  <a href="#">jdk-15.0.2_windows-x64_bin.zip</a>      |

## A. Install Java (3)

---

### 4. Tick and download

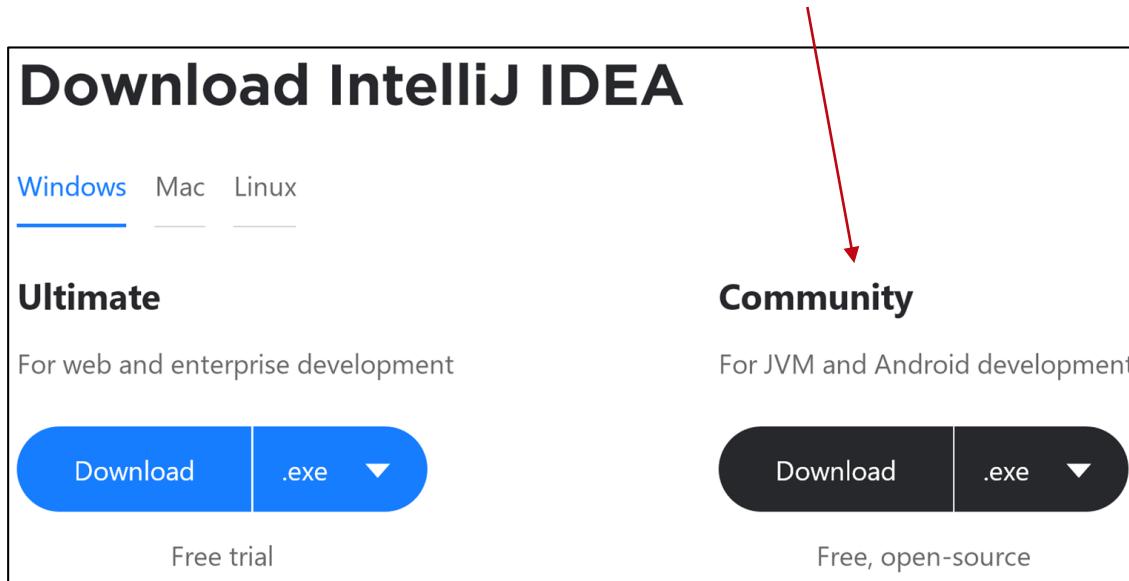


### 5. Run the installer file and select all the default options

## B. Install IntelliJ (1)

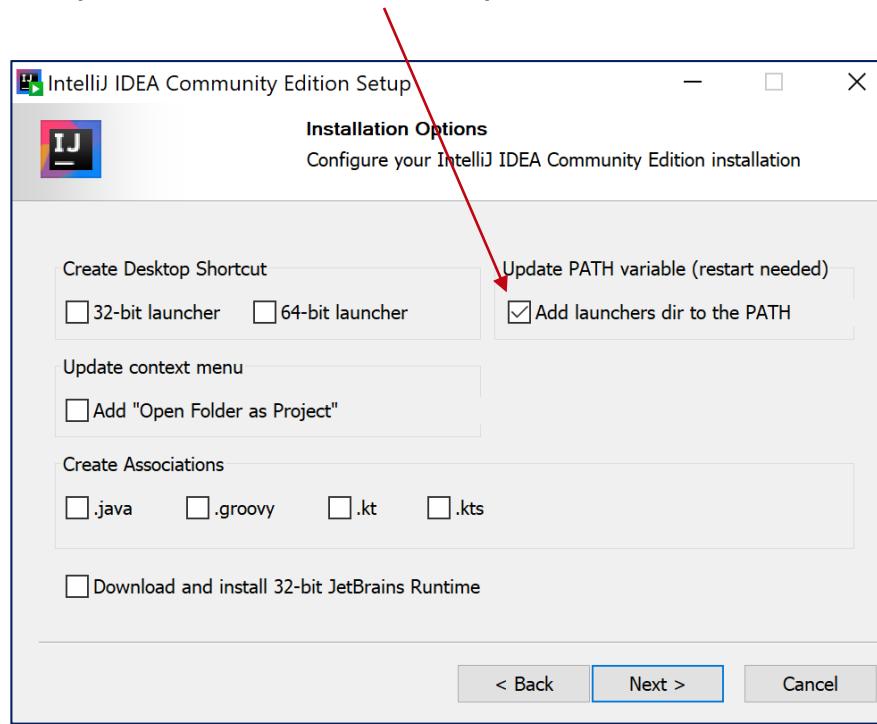
---

1. Go to <https://www.jetbrains.com/idea/download/>
2. Choose your operating system, download the Community Edition



## B. Install IntelliJ (2)

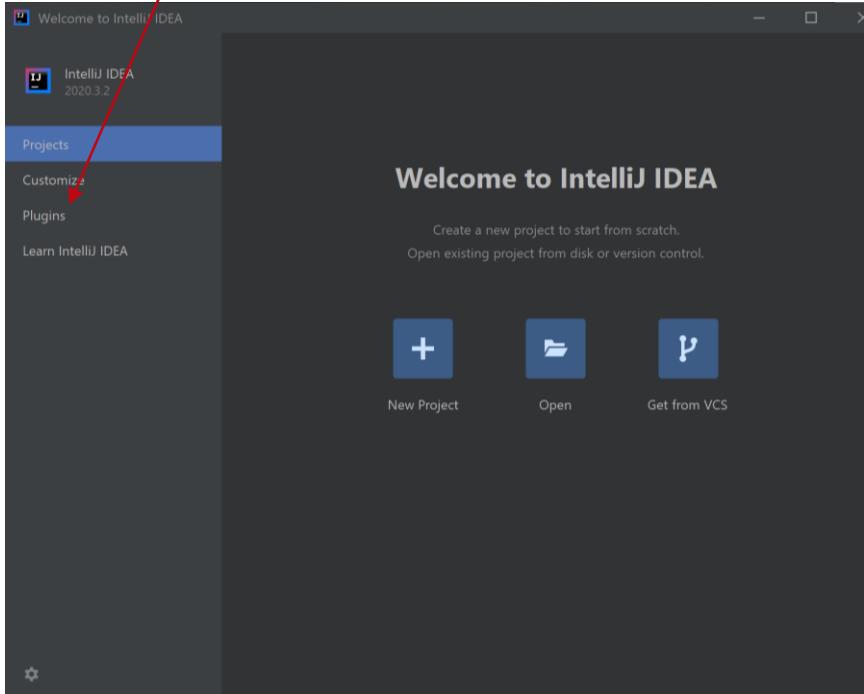
3. Run the installer file, select all the default options, create shortcut if you want, check Update PATH variable



## C. Install IntelliJ Plugins (1)

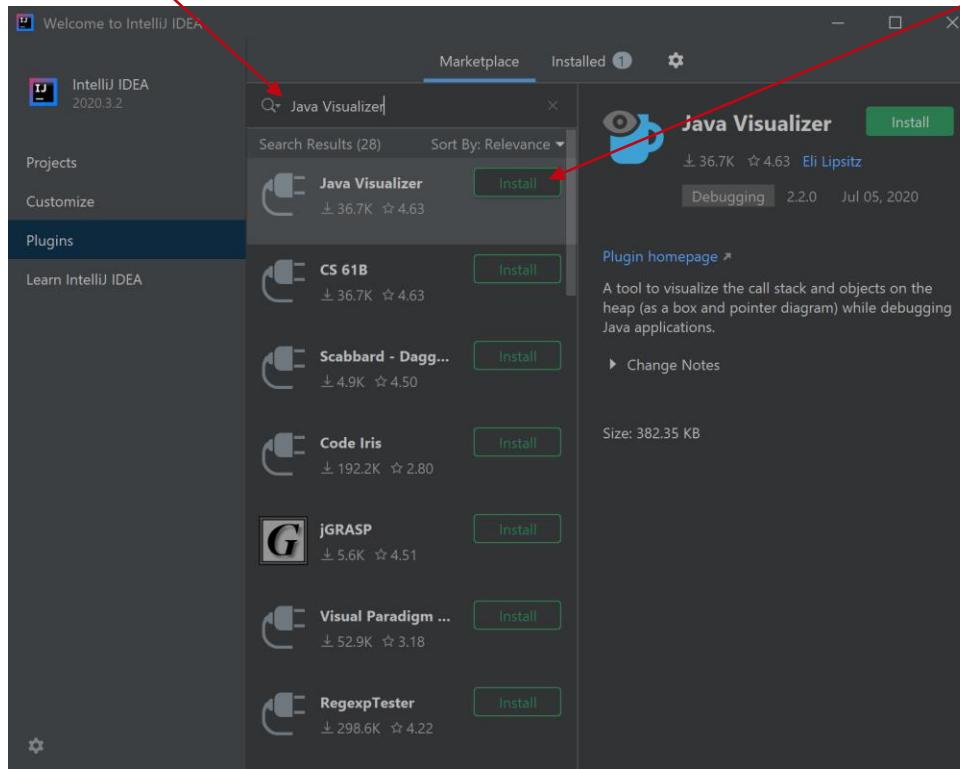
---

1. Run IntelliJ, accept JetBrains Privacy Policy and Data Sharing
2. In Welcome window, click Plugins



## C. Install IntelliJ Plugins (2)

3. In Marketplace, type Java Visualizer in the search bar, then click Install



# Done !

---

- You are now ready to learn and try out the materials for Week 1
- Please continue to read Lecture 1



# Advanced Object-Oriented Programming

CPT204 – Lab 1  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

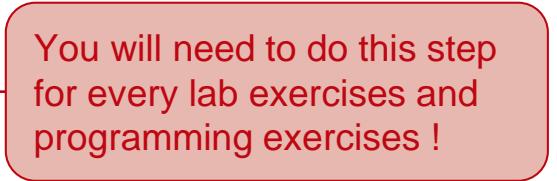
# CPT204 Advanced Object-Oriented Programming

## Lab 1

**Basic Java Review,  
Checking 1, Testing 1**

# Welcome !

---

- Welcome to Lab 1 !
  - Please read Lecture 1 first, if you have not done so
- In this lab, we are going to guide you how to
  1. Import files into a project
  2. Submit lab exercises to Learning Mall Quiz
  3. Submit programming exercises to Learning Mall Quiz

You will need to do this step  
for every lab exercises and  
programming exercises !
- We are also going give hints to Lab Exercise 1.1

# Import Lab 1 into a Project (1)

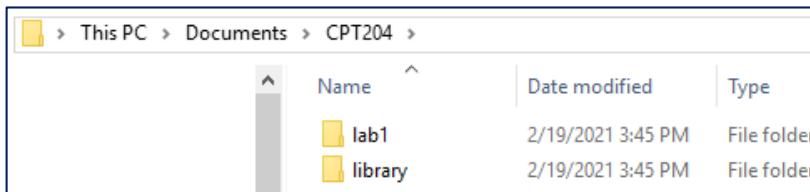
---

We are going to guide you to import the files of Lab 1 into an IntelliJ project  
*(repeat these steps for the subsequent labs !)*

1. Download from Learning Mall the zip files library and lab1 ;



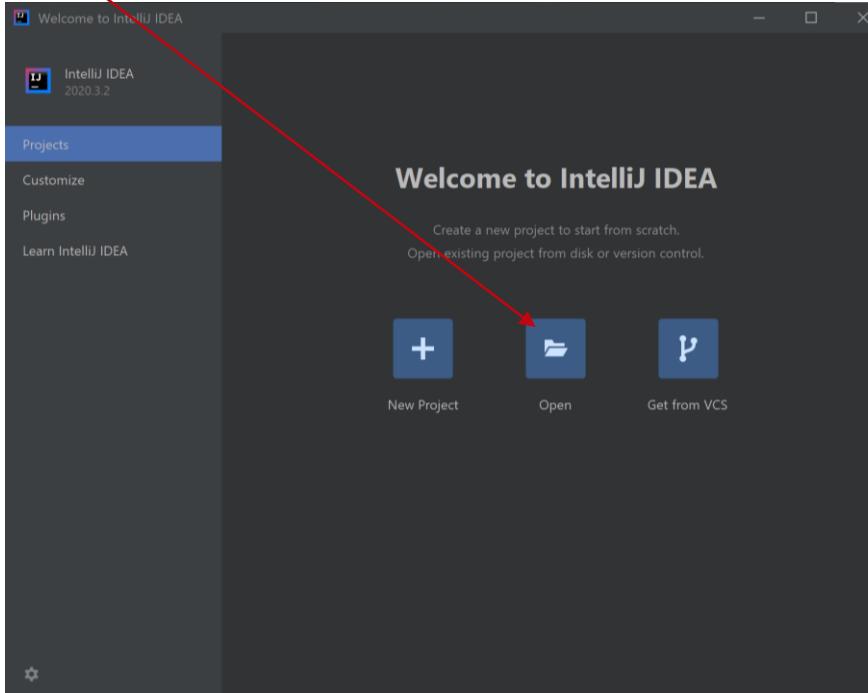
Create a folder for the CPT204 lab files, e.g. CPT204, in your computer;  
Unzip to get these two folders:



# Import Lab 1 into a Project (2)

---

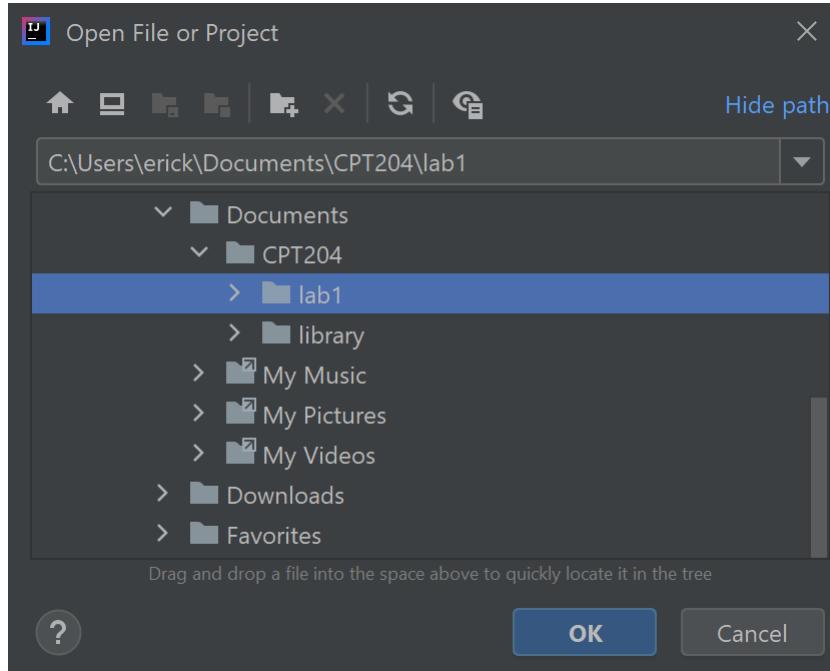
2. Open IntelliJ; click Open



# Import Lab 1 into a Project (3)

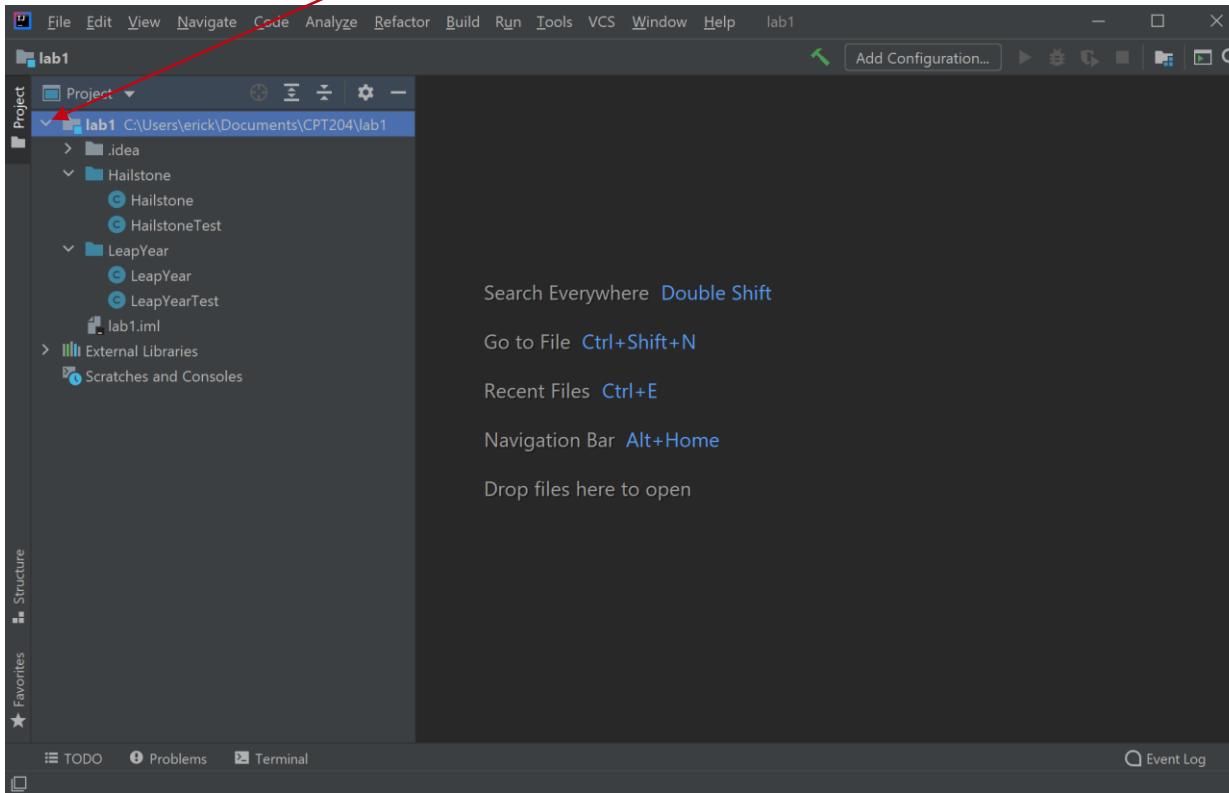
---

3. Find your lab1 folder, and click OK



# Import Lab 1 into a Project (4)

4. Your project is opened; Click the little triangle to show the source files



# Import Lab 1 into a Project (5)

5. Click the source files, it shows errors in one of the source code; Click ...

The screenshot shows an IDE interface with a dark theme. The top menu bar includes File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help, and lab1 - HailstoneTest.java. The left sidebar has a Project view showing a hierarchy: lab1 > Hailstone > HailstoneTest. The HailstoneTest.java file is open in the editor. The code contains several annotations and method definitions:

```
import ...  
public class HailstoneTest {  
    @Test  
    public void testHailstone() {  
        List<Integer> expected = Arrays.asList(5, 16, 8, 4, 2, 1);  
        List<Integer> actual = Hailstone.hailstone( n: 5);  
        assertEquals(expected, actual);  
    }  
  
    @Test  
    public void testMaxHailstone() {  
        int expectedMax = 16;  
        assertEquals(expectedMax, Hailstone.maxHailstone(Hailstone.hailstone( n: 5)));  
    }  
}
```

Red arrows point from the text "Click the source files, it shows errors in one of the source code;" to the project tree and the code editor. A red box highlights the ellipsis (...) in the import statement.

# Import Lab 1 into a Project (6)

---

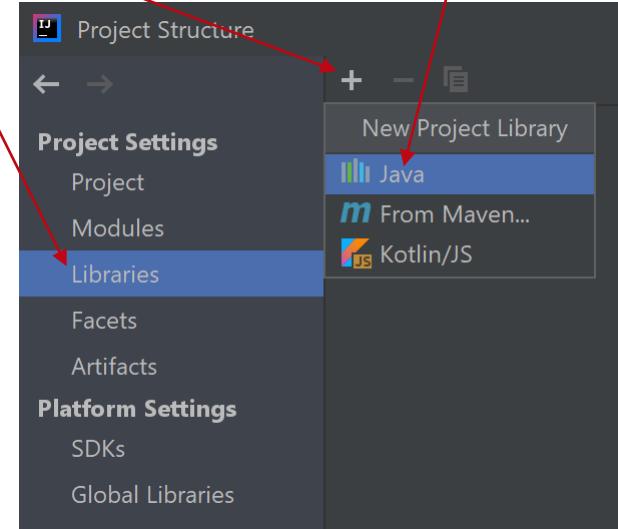
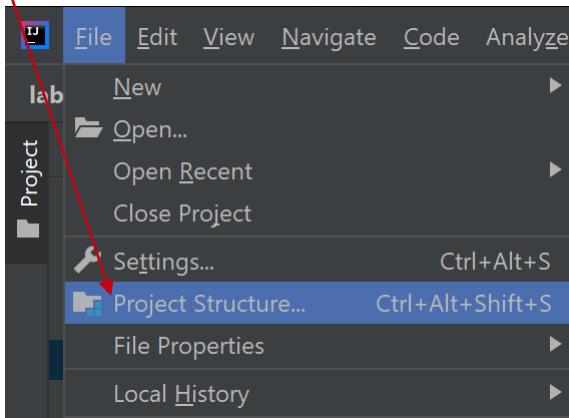
6. Hover over junit; see that Java cannot resolve it; this is because we have not added the necessary libraries, which we will be doing next

The screenshot shows a Java code editor with two files open: `Hailstone.java` and `HailstoneTest.java`. The cursor is positioned over the word `junit` in the `HailstoneTest.java` file. A tooltip appears, stating "Cannot resolve symbol 'junit'". Below the code editor, there is a toolbar with options: "Add 'JUnit4' to classpath" (highlighted in blue), "Alt+Shift+Enter", "More actions...", and "Alt+Enter".

```
1 import java.util.Arrays;
2 import java.util.List;
3 import org.junit.Test;
4 import static org.junit.Assert.*;
5
6 public class Hailstone {
7     @Test
8     public void testHailstone() {
9         List<Integer> expected = Arrays.asList(5, 16, 8, 4, 2, 1);
10        List<Integer> actual = Hailstone.hailstone(n: 5);
11        assertEquals(expected, actual);
12    }
13}
14
15}
```

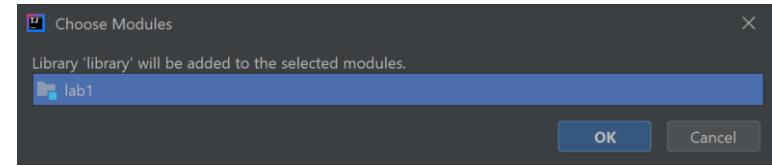
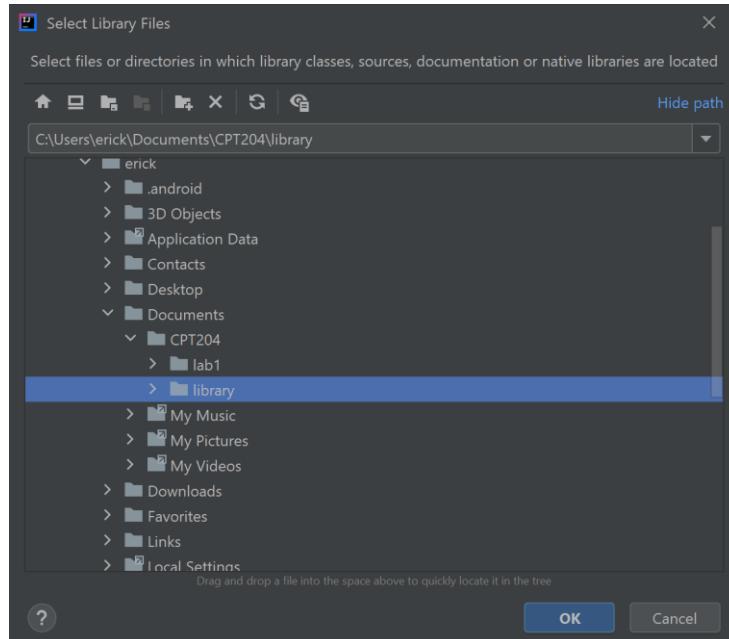
# Import Lab 1 into a Project (7)

7. Click File -> Project Structure; Click Libraries -> Plus sign; Click Java



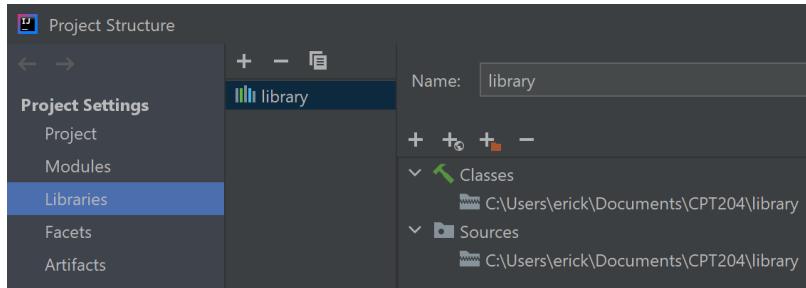
# Import Lab 1 into a Project (8)

- Find your library folder, and click OK; Click OK again



# Import Lab 1 into a Project (9)

- Finally, you should see the library in the Libraries of Project Structure ;



the errors disappear ; and we are now ready to use it in the next section!

The screenshot shows an IDE editor with two tabs: 'Hailstone.java' and 'HailstoneTest.java'. The code in Hailstone.java is:

```
1 import java.util.Arrays;
2 import java.util.List;
3 import org.junit.Test;
4 import static org.junit.Assert.*;
5 @Retention(RetentionPolicy.RUNTIME)
6 @Target({ElementType.METHOD})
7 public interface Test
8     extends annotation.Annotation
9 {
10     @Test
11     public void testHailstone()
12         List<Integer> expected = Arrays.asList(5, 16, 8, 4, 2, 1);
13         List<Integer> actual = Hailstone.hailstone( n: 5);
14         assertEquals(expected, actual);
15 }
```

A code completion dropdown is open over the word 'library' in the code, listing 'library' and 'org.junit'.

# The First Lab Exercise

---

- We now continue with the Lab Exercise
- We will first use IntelliJ to write and test our codes
  - and then you need to submit your code to Learning Mall
- In the next slide, you will see Lab Exercise 1.1
- Lab Exercise slides will consist of
  - Description of the problem
  - One or more test cases
  - Hints and skeleton code (optional)
- After that, we will guide you to solve and submit the Lab Exercise 1.1

## Lab Exercise 1.1 Max Hailstone

---

- Complete the code to compute the maximum element of a hailstone sequence
- The input is an integer  $n > 0$  ,  
and the output is the maximum element in that sequence
- Test case:
  - `maxHailstone(5) → 16`

# Lab Exercise 1.1 Max Hailstone

---

- Skeleton code:

```
/**  
 * Compute the largest element in a hailstone sequence.  
 * For example, maxHailstone(5) = 16.  
 * @param n starting number of the sequence. Assume n > 0.  
 * @return the largest element the sequence.  
 */  
public static int maxHailstone(int n) {  
  
}
```

# Our First Lab Exercise (1)

1. Finish the previous step Import Lab1 into a Project; Open HailstoneTest.java

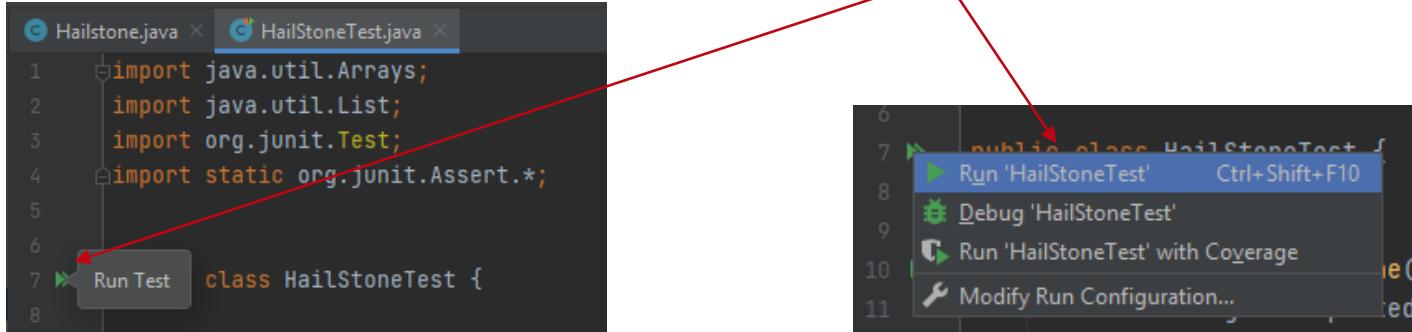
The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Bar:** File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help, lab1 - HailStoneTest.java - Administrator.
- Project Tree (Project View):** lab1 > Hailstone > HailStoneTest. The Hailstone and HailStoneTest nodes are selected.
- Code Editor (HailStoneTest.java):**

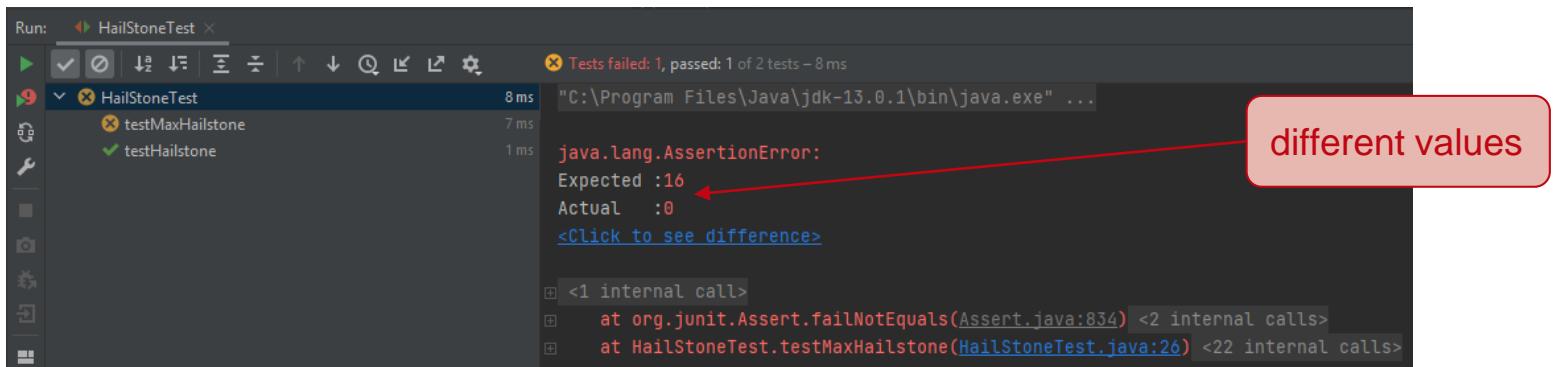
```
1 import java.util.Arrays;
2 import java.util.List;
3 import org.junit.Test;
4 import static org.junit.Assert.*;
5
6
7 public class HailStoneTest {
8
9     @Test
10    public void testHailstone() {
11        List<Integer> expected = Arrays.asList(5, 16, 8, 4, 2, 1);
12        List<Integer> actual = Hailstone.hailstone( 5 );
13        assertEquals(expected, actual);
14
15    }
16
17    @Test
18    public void testMaxHailstone() {
19        int expectedMax = 16;
20        assertEquals(expectedMax, Hailstone.maxHailstone( 5 ));
21
22    }
23
24
25
26    }
27
28 }
```
- Toolbars:** Standard Java development tools like Run, Stop, Refresh, and others.
- Sidebar:** Structure, External Libraries, Scratches and Consoles.

# Our First Lab Exercise (2)

2. Let's first run all the already written tests by clicking Green Arrow and Run



testMaxHailstone failed because we have *not* complete the code



# Our First Lab Exercise (3)

---

3. Add **more** test cases, as discussed in Lecture 1

For example,  $\text{hailstone}(3) = [3 \ 10 \ 5 \ 16 \ 8 \ 4 \ 2 \ 1]$  and its max element = 16

```
@Test
public void testHailstone() {
    List<Integer> expected = Arrays.asList(5, 16, 8, 4, 2, 1);
    List<Integer> actual = Hailstone.hailstone( n: 5);
    assertEquals(expected, actual);

    expected = Arrays.asList(3, 10, 5, 16, 8, 4, 2, 1);
    actual = Hailstone.hailstone( n: 3);
    assertEquals(expected, actual);

}

@Test
public void testMaxHailstone() {
    int expectedMax = 16;
    assertEquals(expectedMax, Hailstone.maxHailstone( n: 5));

    expectedMax = 16;
    assertEquals(expectedMax, Hailstone.maxHailstone( n: 3));
}
```

add your own  
test cases !

# Our First Lab Exercise (4)

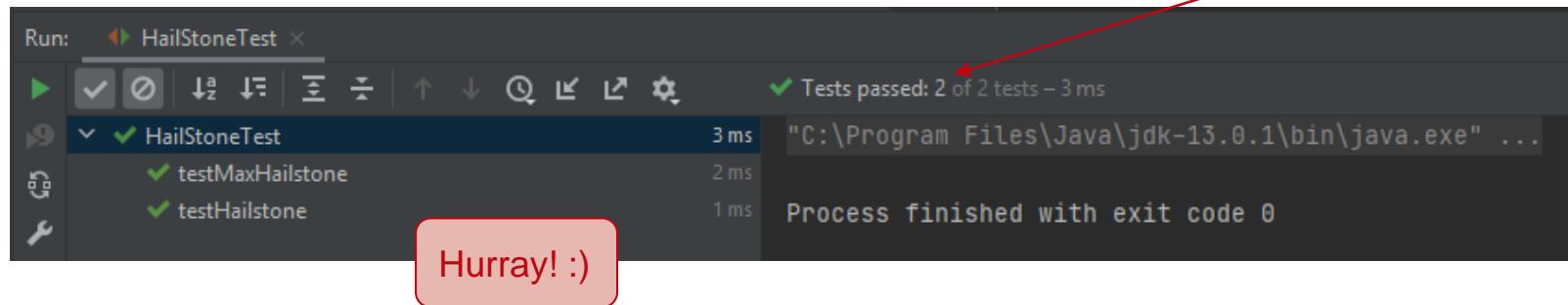
---

## 4. Open Hailstone.java

The method `hailstone` has been completed for you

Now complete the method `maxHailstone`,

and then check againsts your own set of test cases until all tests are passed



## Our First Lab Exercise (5)

---

5. Continue to test and debug your code using IntelliJ & JUnit until all of your own test cases are passed  
(create more test cases if necessary)
  
6. After you are satisfied with your code, submit into Learning Mall, click the quiz Lab Exercise 1.1



# Our First Lab Exercise (6)

7. You will see the following screen, click Attempt quiz now

Lab Exercise 1.1 Max Hailstone

Complete the code to compute the maximum element of a hailstone sequence.  
The input is an integer  $n > 0$  and the output is the maximum element in that sequence.



Test cases :

```
maxHailstone(5) → 16
```

Attempts allowed: 1

This quiz will close on Friday, 12 March 2021, 11:15.

Attempt quiz now

note that only one attempt is allowed for a lab exercise

and it has a deadline !

# Our First Lab Exercise (7)

8. Copy paste your code from IntelliJ into the box, and then click Check

| Test                                 | Result |
|--------------------------------------|--------|
| System.out.println(maxHailstone(5)); | 16     |

**Answer:** (penalty regime: 0 %)

Reset answer

```
1 /**
2  * Compute the largest element in a hailstone sequence.
3  * For example, maxHailstone(5) = 16.
4  * @param n starting number of the sequence. Assume n > 0.
5  * @return the largest element in the sequence.
6  */
7 public static int maxHailstone(int n) {
8
9
10 }
11 }
```

Check

pay attention to the curly braces { }  
missing or extra curly braces are  
common errors!

# Our First Lab Exercise (8)

9. You have to pass **all** test cases: given and hidden ones, to earn marks

```
1 /**
2  * Compute the largest element in a hailstone sequence.
3  * For example, maxHailstone(5) = 16.
4  * @param n starting number of the sequence. Assume n > 0.
5  * @return the largest element the sequence.
6 */
7 public static int maxHailstone(int n) {
8
9     return 16;
10}
```

Check

|   | Test                                 | Expected | Got |   |
|---|--------------------------------------|----------|-----|---|
| ✓ | System.out.println(maxHailstone(5)); | 16       | 16  | ✓ |

Your code failed one or more hidden tests.  
Your code must pass all tests to earn any marks. Try again.

Incorrect  
Marks for this submission: 0.00/100.00.

you cannot see this,  
but the code gave incorrect  
output to some hidden test cases

|   | Test                                   | Expected | Got |   |
|---|----------------------------------------|----------|-----|---|
| ✓ | System.out.println(maxHailstone(5));   | 16       | 16  | ✓ |
| ✓ | System.out.println(maxHailstone(12));  | 16       | 16  | ✓ |
| ✗ | System.out.println(maxHailstone(1));   | 1        | 16  | ✗ |
| ✗ | System.out.println(maxHailstone(256)); | 256      | 16  | ✗ |
| ✗ | System.out.println(maxHailstone(2));   | 2        | 16  | ✗ |

Your code failed one or more hidden tests.  
Your code must pass all tests to earn any marks. Try again.

# Our First Lab Exercise (9)

10. After passing, click Finish attempt , and then click Submit all and finish

The screenshot shows a programming exercise interface. At the top, there is a table with one row containing a green checkmark icon, the word "Test", and the code "System.out.println(maxHailstone(5));". To the right of this are columns for "Expected" (16) and "Got" (16), also with a green checkmark icon. Below the table, a green banner displays the message "Passed all tests!" with a green checkmark icon. At the bottom of the interface, the text "Correct" is displayed in green, followed by "Marks for this submission: 100.00/100.00". A blue button at the bottom left contains the text "Finish attempt ...". A red arrow points from the text "Finish attempt ..." to this button.

The screenshot shows a "Summary of attempt" page. It has a table with three columns: "Question", "Status", and "Marks". There is one row in the table with the question number "1", status "Correct", and marks "100.00". Below the table is a blue button labeled "Return to attempt". At the bottom of the page, a message states "This attempt must be submitted by Friday, 12 March 2021, 11:15." A red arrow points from the text "Submit all and finish" to a blue button at the bottom right of the page.

**WARNING:** Hint to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 1.1 Max Hailstone Hints

---

- You can simply reuse the code to generate the list of hailstone numbers, and find its maximum number
- Or, more efficiently, you can just store the current max while iterating
  - make sure you initialize the variable storing max correctly

# The First Programming Exercise

---

- We now continue with the Programming Exercise
  - Just like the Lab Exercise, use IntelliJ to write and test your code first
  - After it is well-tested, submit your code to Learning Mall
- 
- In the next slide, you will see the Exercise 1.1
  - Programming Exercise slides will consist of
    - Description of the problem
    - One or more test cases
    - Hints and skeleton code (optional)
- 
- After that, we will guide you to submit the Exercise 1.1
    - You will have to solve it yourself

## Exercise 1.1 Leap Year

---

- Leap years are years where an extra day is added to the end of the shortest month, February.
- Write a Java method with signature `boolean isLeapYear(int year)` that returns `true` if and only if `year` is a leap year.
- Test cases:
  - `isLeapYear(2020) → true`
  - `isLeapYear(2019) → false`

# Exercise 1.1 Leap Year

---

- Skeleton code:

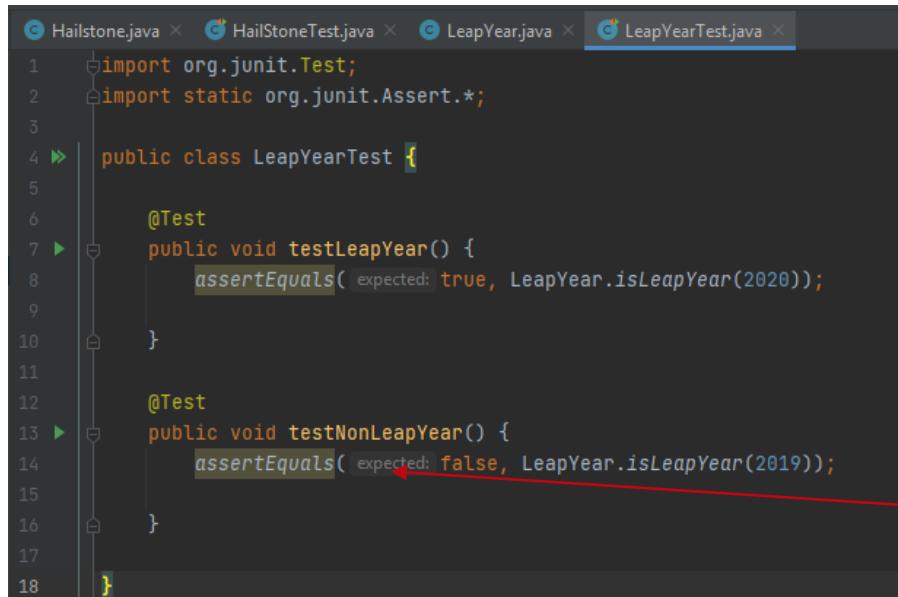
```
/*
 * Checks if an input year is a leap year.
 * @param year is the input year
 * Requires year to be a valid year
 * @return true iff year is a leap year
 */
public static boolean isLeapYear(int year) {

    return true;
}
```

# Our First Programming Exercise (1)

1. It is recommended that you finish Lab Exercise 1.1 first

Open LeapYearTest. java, write the JUnit import statements, and write the test cases, for example:



```
1 import org.junit.Test;
2 import static org.junit.Assert.*;
3
4 public class LeapYearTest {
5
6     @Test
7     public void testLeapYear() {
8         assertEquals( expected: true, LeapYear.isLeapYear(2020));
9     }
10
11    @Test
12    public void testNonLeapYear() {
13        assertEquals( expected: false, LeapYear.isLeapYear(2019));
14    }
15
16 }
17
18 }
```

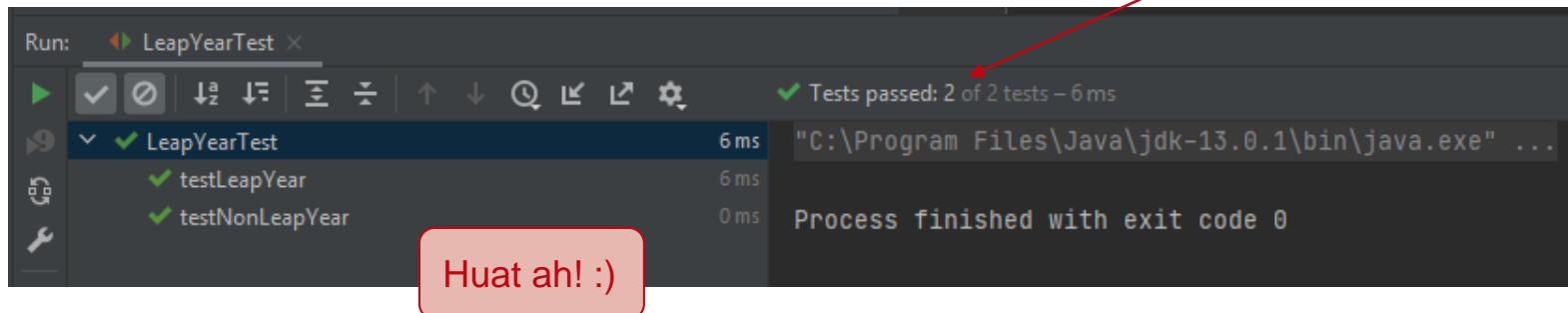
text automatically added by IntelliJ,  
to remind you what argument it is,  
do not type in "expected:"

Test cases above are **not enough**, find other cases of leap/nonleap year !!

# Our First Programming Exercise (2)

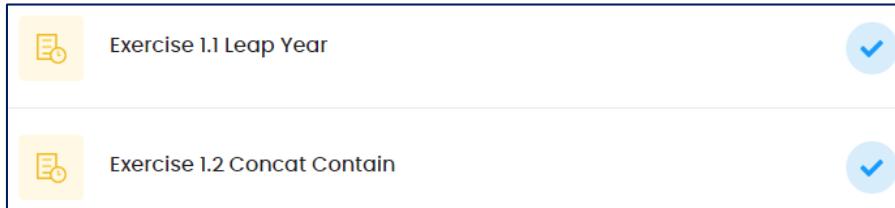
2. Open LeapYear.java, write your code

Continue to write, test, debug your code in IntelliJ until it passes your good set of test cases



A screenshot of the IntelliJ IDEA interface during a test run. The 'Run' tool window is open, showing the 'LeapYearTest' tab. Inside, there are two test methods: 'testLeapYear' and 'testNonLeapYear', both of which have passed, indicated by green checkmarks. The execution time for each is listed as 6 ms. The command used for the run is shown as "C:\Program Files\Java\jdk-13.0.1\bin\java.exe" ... . At the bottom right of the interface, the message "Process finished with exit code 0" is displayed. A red arrow points from the text "Huat ah! :)" in a red box at the bottom left towards the green checkmark indicating test success.

3. After you are satisfied with your code, submit it to Learning Mall the same way as submitting Lab Exercise 1.1 before



A screenshot of the Learning Mall submission interface. It shows two exercises listed: 'Exercise 1.1 Leap Year' and 'Exercise 1.2 Concat Contain'. Both exercises have been successfully submitted, as indicated by a blue checkmark icon next to each exercise name. The background of the interface is light blue.

# Our First Programming Exercise (3)

## 4. Note that programming exercises carry penalty for repeated mistakes

Leap years are years where an extra day is added to the end of the shortest month, February.

Write a Java method with signature `boolean isLeapYear(int year)` that returns true if and only if `year` is a leap year.

For example:

| Test                                                                                  | Result |
|---------------------------------------------------------------------------------------|--------|
| <code>boolean ans = isLeapYear(2020);</code><br><code>System.out.println(ans);</code> | true   |
| <code>boolean ans = isLeapYear(2019);</code><br><code>System.out.println(ans);</code> | false  |

Answer: (penalty regime: 0, 15, 30, ... %)

Reset answer

```
1 /**
2 * Checks if an input year is a leap year.
3 * @param year is the input year
4 * Requires year to be a valid year
5 * @return true iff year is a leap year
6 */
7 public static boolean isLeapYear(int year) {
8
9     return false;
10 }
```

Check

note that the **first Check** has no penalty if your code is incorrect, but the **second Check** and subsequent checks will be penalized by accumulative 15% thus, do **not** test your code here, but **test in IntelliJ+JUnit** !

# Our First Programming Exercise (4)

For example, if your 1<sup>st</sup>, 2<sup>nd</sup> and 3<sup>rd</sup> checks are incorrect,  
but your 4<sup>th</sup> check is correct,  
you will be penalized by 15%+15%

| Test                                                                    | Expected | Got     |
|-------------------------------------------------------------------------|----------|---------|
| <pre>boolean ans = isLeapYear(2020);<br/>System.out.println(ans);</pre> | true     | true ✓  |
| <pre>boolean ans = isLeapYear(2019);<br/>System.out.println(ans);</pre> | false    | false ✓ |

Passed all tests! ✓

Correct  
Marks for this submission: 100.00/100.00. Accounting for previous tries, this gives 70.00/100.00.

the same setting is used for the CW / Lab Exam!

don't test your code here,  
test your code in IntelliJ !

## Exercise 1.2 Concat Contain

---

- Complete the method int concatContain(String source, String target).
- Given two *non-empty* strings source and target, it could be possible to **concatenate** the string source **with itself a number of times**, so that the string target can be **contained** in it.
- For example, source "ab" concatenated 2 times "ab"+"ab"+"ab" into "ababab" contains target "baba".
- Return the smallest number of times you concatenate source so that it contains target; and if it is not possible for target to be contained in concatenated source strings, return -1.
- You must **not** use StringBuilder or Regular Expression methods: append, matches, replaceAll.
- Test cases:
  - concatContain("ab", "baba") → 2
  - concatContain("ab", "abcde") → -1

## Exercise 1.2 Concat Contain

---

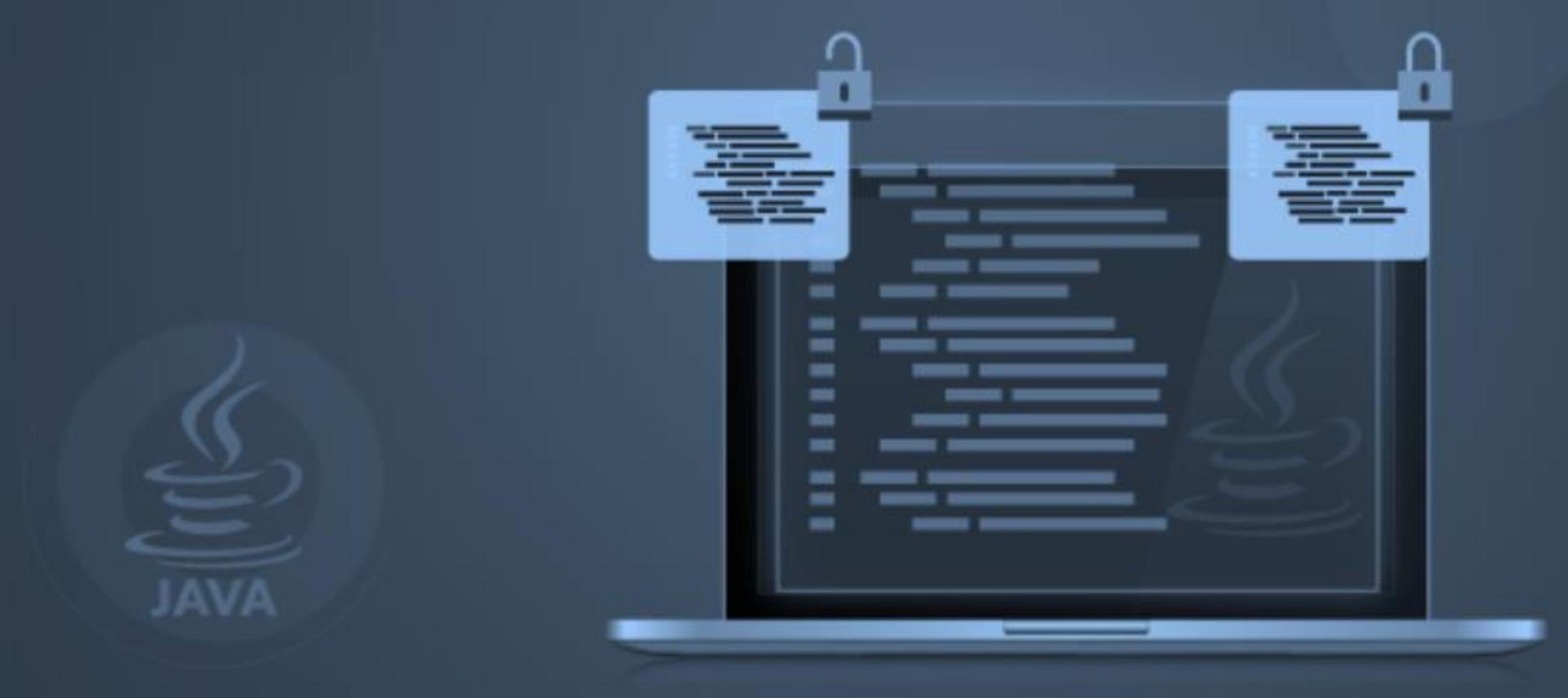
- Skeleton code:

```
/**  
 * Compute the smallest number of times source is concatenated with itself  
 * so that the resulting string contains target.  
 * For example, For example, source "ab" concatenated 2 times "ab"+"ab"+"ab" into "ababab"  
 * contains target "baba".  
 * @param source a non-empty string to be concatenated.  
 * @param target a non-empty string that can be contained in repeatedly concatenated source.  
 * @return the smallest number of times of the concatenation.  
 */  
public static int concatContain(String source, String target) {  
  
    return 0;  
}
```

# Thank you for your attention !

---

- In this lab, you have learned:
  - Creating test cases
  - Test-driven Programming
  - Reviewing basic Java and Lists
  - Using IntelliJ and JUnit
  - Submitting your code to Learning Mall



# Advanced Object-Oriented Programming

CPT204 – Lab 2  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming Lab 2

**Checking 2, Testing 2,  
List, Map**

# Welcome !

---

- Welcome to Lab 2 !
  - We are going continue practising with testing while reviewing list and map
- You will find in this lab
  1. Lab Exercise 2.1, 2.2, and their hints
  2. Exercise 2.1 - 2.4
- Download **lab2** zip file from Learning Mall
- Don't forget to import the **lab2** files and the library into an IntelliJ project
  - Read **lab1** again for reference

## Lab Exercise 2.1 MaxStretch

---

- We define the **stretch** of a value in a list to be the number of elements between that two leftmost and rightmost values in that list, inclusive  
For example, the stretch of 2 in [5, 2, 2, 5, 2] is 4 (from **2**, 2, 5, **2**)

A single value in a list has a stretch of 1

Write a method that returns the *maximum* stretch found in the input list

- Test case 1:

`maxStretch([8, 5, 1, 2, 3, 4, 5, 10]) = 6` (from **5**, 1, 2, 3, 4, **5**)

- Test case 2:

`maxStretch([2, 7, 1, 2, 3, 7]) = 5` (from **7**, 1, 2, 3, **7**)

# Lab Exercise 2.1 MaxStretch

---

- Skeleton code:

```
/**  
 * Find the largest stretch in a list.  
 * For example, maxStretch([8, 5, 1, 2, 3, 4, 5, 10]) = 6.  
 * @param list is a list of integers.  
 * @return the largest stretch in list.  
 */  
public static int maxStretch(List<Integer> list) {  
  
}
```

# Continue with Test-Driven Programming

We use the same approach as last week, before starting to write the maxStretch method, write the test code for it first !

- Open MaxStretchTest.java, create and add more test cases, for example:

```
@Test  
public void testOverlapMaxStretch() {  
    List<Integer> list = Arrays.asList(2, 3, 2, 3, 3, 2, 2, 3);  
    assertEquals( expected: 7, Stretch.maxStretch(list));  
}  
  
@Test  
public void testEmptyList() {  
    List<Integer> list = Arrays.asList();  
    assertEquals( expected: 0, Stretch.maxStretch(list));  
}  
  
@Test  
public void testSingletonList() {  
    List<Integer> list = Arrays.asList(55555);  
    assertEquals( expected: 1, Stretch.maxStretch(list));  
}  
  
@Test  
public void testOneWholeStretch() {  
    List<Integer> list = Arrays.asList(4, 4, 4, 4);  
    assertEquals( expected: 4, Stretch.maxStretch(list));  
}
```

use descriptive name  
for your test

include corner cases!

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 2.1 MaxStretch Hints

---

- One way to solve this is by using two nested loops
  - two pointers: one from left to right, one from right to left
- Use a variable, say max, to keep track maximum so far
- The outer loop goes through every element starting from the beginning
- The inner loop goes from the end of the list, up until the outer loop index
  - stops when finding the same element pointed by the outer loop
- After inner loop finishes, compute the stretch by the difference of inner and outer loop indices
  - update max if larger stretch is found
- After outer loop finishes, the max stretch is found
  - return it

## Lab Exercise 2.2 EvenAppend

---

- Given an input of a list of strings, write a method to build a result string as follows: when a string appears the 2nd, 4th, 6th, etc. time in the list, append the string to the result.

Return the empty string if no string appears a 2nd time.

- Test case 1:

evenAppend(["a", "b", "a"]) → "a"

- Test case 2:

evenAppend(["a", "b", "b", "a", "a"]) → "ba"

## Lab Exercise 2.2 EvenAppend

---

- Skeleton code:

```
/**  
 * Append words that appear the 2nd, 4th, 6th, etc. time in a list.  
 * For example, evenAppend(["a", "b", "b", "a", "a"]) → "ba".  
 * @param list is a list of words.  
 * @return a concatenation of even appearing words.  
 */  
public static String evenAppend(List<String> list) {  
  
}
```

# Continue with Test-Driven Programming

---

We use the same approach as last week, write the test code first

- Open EvenAppendTest.java, create and add more test cases, for example:

```
@Test
public void testManyEven() {
    List<String> list = Arrays.asList("a", "b", "b", "b", "a", "c", "a", "a", "a", "b", "a", "b", "c");
    assertEquals( expected: "baabac", EvenAppend.evenAppend(list));
}

@Test
public void testEmptyList() {
    List<String> list = Arrays.asList();
    assertEquals( expected: "", EvenAppend.evenAppend(list));
}

@Test
public void testSingletonList() {
    List<String> list = Arrays.asList("one");
    assertEquals( expected: "", EvenAppend.evenAppend(list));
}

@Test
public void testOverlapEven() {
    List<String> list = Arrays.asList("xxx", "xxx", "y", "yy", "xx", "xxx", "zz", "yy", "zz", "xx", "y");
    assertEquals( expected: "xxxxyyzzxy", EvenAppend.evenAppend(list));
}
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 2.2 EvenAppend Hints

---

- Create a map, with String as key, and Integer as value
- Create a string res to store the result
- Iterate over all Strings in the list
  - If it is not in the map yet, store it, with value 1
  - Else (it is in the map)
    - check the value
      - if it is odd (the next one would make it even occurrence)  
append the key (string) into res
      - else do nothing
- increment the value
- Return res

# Week 2 Online Programming Exercises

---

- Start with creating a good set of test cases first !
  - Include the **corner cases**, such as empty list for countRuns; a list with one element for isPartitionable; empty list and list with one/two non-empty strings for sameFirstLetter and matchSwap; as well as when it contains empty strings
- Use IntelliJ & JUnit to write and test your code

## Exercise 2.1 Count Runs

---

- We define a **runs** in a list is a series of 2 or more adjacent elements of the same value
  - Write a method to return the number of runs in the input list
- 
- Test case 1:  
`countRuns([1, 2, 2, 2, 3]) = 1` (which is 2, 2, 2)
  - Test case 2:  
`countRuns([1, 1, 2, 3, 4, 5, 5]) = 2` (which is 1, 1; and 5, 5)

## Exercise 2.1 Count Runs

---

- Skeleton code:

```
/**  
 * Count the number of runs in a list.  
 * For example, countRuns([1, 2, 2, 2, 3]) = 1.  
 * @param list is a list of integers.  
 * @return the number of runs in list.  
 */  
public static int countRuns(List<Integer> list) {  
  
}
```

## Exercise 2.2 Partitionable

---

- We define a list to be **partitionable**, if there is a place in between two indices in that list where the *sum* of the numbers on one side is *equal to* the *sum* of the numbers on the other side
- Given as an input a *non-empty* list, write a method that returns true if and only if the list is partitionable
- Test cases:
  - `isPartitionable([1, 1, 1, 2, 1])` --> true
  - `isPartitionable([2, 1, 1, 2, 1])` --> false

## Exercise 2.2 Partitionable

---

- Skeleton code:

```
/**  
 * Decide whether a list is partitionable.  
 * For example, isPartitionable([1, 1, 1, 2, 1]) -> true,  
 * and isPartitionable([2, 1, 1, 2, 1]) -> false.  
 * @param list is a non-empty list of integers.  
 * @return true iff list is partitionable.  
 */  
public static boolean isPartitionable(List<Integer> list) {  
  
    return true;  
}
```

## Exercise 2.3 Same First Letter

---

- Given a list of non-empty strings, write a method that returns a Map<String, String> with a key for every different first letter seen, with the value of all the strings starting with that letter appended with a comma (,) together in the order they appear in the list
- Test case 1:  
`sameFirstLetter(["alice", "bob", "apple", "banana"])`  
→ `{"a": "alice,apple", "b": "bob,banana"}`
- Test case 2:  
`sameFirstLetter(["after", "all", "this", "time", "always"])`  
→ `{"a": "after,all,always", "t": "this,time"}`

## Exercise 2.3 Same First Letter

---

- Skeleton code:

```
/*
 * Create a map with first letter as key and words with that same
 * first letter separated by comma.
 * For example, numWords(["alice", "bob", "apple", "banana"]) →
 * {"a": "alice,apple", "b": "bob,banana"}.
 * @param list is a list of strings.
 * The strings are non-empty.
 * @return a map with first letter and comma-separated-words pair.
 */
public static Map<String, String> sameFirstLetter(List<String> list) {

}
```

## Exercise 2.4 Match Swap

---

- We define that 2 strings **match** if they are non-empty and their *first letters* are the same
- Given a list of *non-empty* strings, return that list modified as follows: if a string matches an earlier string in the list, swap those 2 strings in the list. After a position in the list has been swapped, it no longer matches anything.
- Hint: Using a map, this can be solved by making just one pass over the list.
- Test case 1:  
`matchSwap(["apple", "avocado"]) → ["avocado", "apple"]`
- Test case 2:  
`matchSwap(["ab", "ac", "ad", "ae", "af"]) → ["ac", "ab", "ae", "ad", "af"]`
- Test case 3:  
`matchSwap(["ap", "bp", "cp", "aq", "cq", "bq"])  
→ ["aq", "bq", "cq", "ap", "cp", "bp"]`

## Exercise 2.4 Match Swap

---

- Skeleton code:

```
/*
 * Modify a list of strings such that two strings with same
 * first letter are swapped.
 * For example, matchSwap(["ap", "bp", "cp", "aq", "cq", "bq"]) →
 * ["aq", "bq", "cq", "ap", "cp", "bp"].
 * @param list is a list of strings.
 * The strings are non-empty.
 * @return the modified list.
 */
public static List<String> matchSwap(List<String> list) {

}
```

# Thank you for your attention !

---

- In this lab, you have learned:
  - Creating good test cases
    - Including corner cases
  - Test-driven Programming
  - Reviewing operations on Lists and Maps



# Advanced Object-Oriented Programming

CPT204 – Lab 3  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

# CPT204 Advanced Object-Oriented Programming

## Lab 3

# Recursion

# Welcome !

---

- Welcome to Lab 3 !
  - We are going continue practising with testing while reviewing recursion and recursion + helper method
- You will find in this lab
  1. Lab Exercise 3.1, 3.2 and their hints
  2. Exercise 3.1 - 3.5
- Download **lab3** zip file from Learning Mall
- Don't forget to import the **lab3** files and the library into an IntelliJ project
  - Read **lab1** again for reference

## Lab Exercise 3.1 CheckSubstring

---

- Given an input string, a ***non-empty*** substring `subs`, and a non-negative number ***n***, compute *recursively* and return **true** if and only if at least ***n*** copies of `subs` occur in the string, possibly with ***overlapping***.

Do **not** use loops (do not write **for** or **while** anywhere in your code).

Do **not** use any regular expressions and methods such as `matches`, `split`, `replaceAll`.

- Test case 1:

`checkSubstring("abcxxxxabc", "abc", 2) → true`

- Test case 2:

`checkSubstring("abcxxxxabc", "abc", 3) → false`

- Test case 3:

`checkSubstring("ababaxxxaba", "aba", 3) → true`

# Lab Exercise 3.1 CheckSubstring

---

- Skeleton code:

```
/**  
 * Check if at least n copies of a substring occur in input string.  
 * They may overlap.  
 * For example, checkSubstring("ababaxxxaba", "aba", 3) → true  
 * @param input is the input string.  
 * @param subs is the non-empty substring.  
 * @param n is non-negative number of occurrences.  
 * @return true iff at least n copies of subs occur in input.  
 */  
public static boolean checkSubstring(String input, String subs, int n) {  
    // base case  
  
    // recursive step  
}
```

# Continue with Test-Driven Programming

We use the same approach as last week, write the test code first

- Open CheckSubstringTest.java, create and add more test cases, for example:

```
@Test
public void testEmptyInput() {
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "", subs: "a", n: 0));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "", subs: "a", n: 1));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "", subs: "a", n: 2));
}
@Test
public void testOneCharInput() {
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "a", subs: "a", n: 0));
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "a", subs: "a", n: 1));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "a", subs: "a", n: 2));
}
@Test
public void testTotalOverlap() {
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "a", n: 5));
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "a", n: 6));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "a", n: 7));
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "aa", n: 4));
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "aa", n: 5));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "aa", n: 6));
}
@Test
public void testNoSubs() {
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "fexfe", subs: "ef", n: 0));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "fexfe", subs: "ef", n: 1));
}
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 3.1 CheckSubstring Hints

---

- The idea is to check whether subs is the prefix of input string, recursively call the method always on input with one less letter; and on one less n if subs is the prefix
- Base case, therefore, is when length of input is less than subs  
Depending on n, if n is positive, which means not enough subs found, return false; otherwise, n zero or negative, found enough or even more, return true
- Recursive step, two cases:
  - subs is a prefix, return recursive call on one less n,
  - subs is not prefix, return recursive call on original n,  
both calling on the input string shifted by 1, since we count overlaps

## Lab Exercise 3.2 EqualSum

---

- Given a list of integers, you want to know whether it is possible to divide the integers into two sets, so that the sums of the two sets are the same. Every integer **must be in** one set or the other.  
Write a *recursive* helper method that takes ***any number of arguments you like***, and make the initial call to your recursive helper method from equalSum().  
Do **not** use any loops or regular expressions.
- Test case 1:  
`equalSum([2, 3, 5]) → true`
- Test case 2:  
`equalSum([2, 2, 5]) → false`

## Lab Exercise 3.2 EqualSum

---

- Given a list of integers, you want to know whether it is possible to divide the integers into two sets, so that the sums of the two sets are the same.  
Every integer **must be in** one set or the other.  
Write a *recursive* helper method that takes **any number of arguments you like**, and make the initial call to your recursive helper method from equalSum().  
**Do not** use any loops or regular expressions.

- Test case 1:

equalSum([2, 3, 5]) → true

since {2, 3} and {5}  
have an equal sum

- Test case 2:

equalSum([2, 2, 5]) → false

the sets cannot be {2} and {2}  
5 must be in a set

## Lab Exercise 3.2 EqualSum

---

- Skeleton code:

```
/*
 * Decide if it is possible to divide the integers in a list into two sets,
 * so that the sums of the two sets are the same.
 * Every integer must be in one set or the other.
 * For example, equalSum([2, 3, 5]) → true.
 * @param list is a list of integers.
 * @return true iff there are two sets having the same sum.
 */
public static boolean equalSum(List<Integer> list) {

    // call your recursive helper method

}
```

# Continue with Test-Driven Programming

---

We use the same approach as last week, write the test code first

- Open EqualSumTest.java, create and add more test cases, for example:

```
@Test
public void testZeroSum() {
    List<Integer> list = Arrays.asList(5, -5);
    assertEquals( expected: true, EqualSum.equalSum(list));
}

@Test
public void testEmptyList() {
    List<Integer> list = Arrays.asList();
    assertEquals( expected: true, EqualSum.equalSum(list));
}

@Test
public void testSingletonList() {
    List<Integer> list = Arrays.asList(2);
    assertEquals( expected: false, EqualSum.equalSum(list));
}
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 3.2 EqualSum Hints

---

- One solution is to use a recursive helper method with 4 parameters: we pass the list, the usual starting index, and two sums: sum1 and sum2.
  - The original method just call it with initial start and sums equal zero
- Since we're going to increment start, the base case is when start exceeds the valid index, in which case we return true iff the two sums are equal.
- Two cases for the recursive step:
  - we add element at index start to the first sum,
  - or the second sum,  
and just pass the other sum unmodified.

So we do two recursive calls to list starting start+1, and return true iff at least one of those two cases are true.

# Week 3 Online Programming Exercises

---

- Start with creating a good set of test cases first !
  - Include the corner cases, such as empty list or empty string ; and corner cases for the elements of the list ; as well as a test case for every possible subdomain
- Use IntelliJ for testing
- Do **not** write for or while anywhere in your code

## Exercise 3.1 CountBabaMama

---

- Given a string, write a method that returns the number of occurrences of substrings "baba" or "mama" in the input string **recursively**.  
They **may** overlap.  
Do **not** use any loops within your code.  
Do **not** use any regular expressions and methods such as matches, split, replaceAll.
- Test case 1:  
`countBabaMama("aba babaa amama ma") → 2`
- Test case 2:  
`countBabaMama("bababamamama") → 4`  
(2 babas that overlap, plus 2 mamas that overlap)

## Exercise 3.1 CountBabaMama

---

- Skeleton code:

```
/**  
 * Count the number of occurrences of substrings "baba" or "mama"  
 * in the input string recursively. They may overlap.  
 * For example, countBabaMama("aba babaa amama ma") → 2,  
 * and countBabaMama("bababamamama") → 4.  
 * @param input is the input string.  
 * @return the number of occurrences.  
 */  
public static int countBabaMama(String input) {  
  
}
```

## Exercise 3.2 DelDuplicate

---

- Given an input string, delete *recursively* all the ***duplicate adjacent characters*** and return a string where all those adjacent characters that are the same have been reduced to just a ***single*** character.

Do **not** use any loops.

Do **not** use any regular expressions and methods such as matches, split, replaceAll.

- Test case 1:

delDuplicate("aaabbc") → "abc"

- Test case 2:

delDuplicate("aaaaa") → "a"

## Exercise 3.2 DelDuplicate

---

- Skeleton code:

```
/*
 * Remove adjacent duplicate characters in a string.
 * For example, delDuplicate("aaabbc") → "abc".
 * @param input is the input string.
 * @return the resulting string.
 */
public static String delDuplicate(String input) {

}
```

## Exercise 3.3 ExtractVowel

---

- Given an input string, ***complete a helper method*** that extracts the vowels and returns the string of the vowels in the input string ***recursively***.  
All strings are **lowercase**.  
Do **not** use any loops or regular expressions.
- Test case 1:  
`extractVowel("i love you 3000") → "ioeou"`
- Test case 2:  
`extractVowel("aidueo") → "aidueo"`

## Exercise 3.3 ExtractVowel Public and isVowel Methods

---

- Assume you are given two methods, one that calls a recursive helper method, and one that gives you true iff the input char is a vowel, that you can use.

```
/*
 * Extract the vowels from the input string.
 * For example, extractVowel("i love you 3000") → "ioeou".
 * @param str is the input string.
 * @return the vowels of the input string.
 */
public static String extractVowel(String str) {
    return extractVowelHelper(str, 0, "");
}

private static boolean isVowel(char c) {
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}
```

## Exercise 3.3 ExtractVowel Private Recursive Helper Method

---

- Your task is to complete the following recursive helper method, where
  - str is the original string
  - start is the starting index to look for the vowels
  - vowels store the previously found vowels

```
private static String extractVowelHelper(String str, int start, String vowels) {  
    // base case  
  
    // recursive step  
  
}
```

## Exercise 3.4 SkipSum

---

- Given a list of integers, we want to know whether it is possible to choose a subset of some of the integers, such that the integers in the subset adds up to the given sum ***recursively***.

We also want that if an integer is chosen to be in the sum, the integer **next to it** in the list must be skipped and **not** chosen to be in the sum.

Do **not** use any loops or regular expressions.

- Test cases:

- skipSum([2, 5, 10, 6], 12) → true (since we can choose {2, 10})
- skipSum([2, 5, 10, 6], 7) → false (since we cannot choose {2, 5})
- skipSum([2, 5, 10, 6], 16) → false (since we cannot choose {10, 6})

## Exercise 3.4 SkipSum

---

- Given code:

```
/**  
 * Decide whether there is a subset in the input list  
 * that adds up to the target sum, where adjacent integers  
 * in the list must not be both in the subset.  
 * For example, skipSum([2, 5, 10], 12) → true,  
 * and skipSum([2, 5, 10], 7) → false.  
 * @param list is the input list.  
 * @param sum is the target sum.  
 * @return true iff there is a subset of non-adjacent integers  
 * in the list that adds to sum.  
 */  
public static boolean skipSum(List<Integer> list, int sum) {  
  
    // call your recursive helper method  
    return skipSumHelper(list, 0, sum);  
}
```

## Exercise 3.4 SkipSum Helper Method

---

- Skeleton code:

```
private static boolean skipSumHelper(List<Integer> list, int start,  
int sum) {  
  
    // base case  
  
    // recursive call  
  
}
```

## Exercise 3.5 OddAndTen

---

- Given a list of integers, you want to know whether it is possible to divide the integers into two sets, so that the sum of one set is *odd*, and the sum of the other set is a *multiple of 10*.

Every integer **must be in** one set or the other.

You *can* write a *recursive helper method* that takes **any number of arguments** and then call it inside the method, but you *cannot* use any loops.

- Test cases:
  - `oddAndTen([5, 5, 3])` → true (the sets are {3} and {5, 5})
  - `oddAndTen([5, 5, 4])` → false (since we cannot choose odd and mult of 10)
  - `oddAndTen([5, 5, 4, 1])` → true (the sets are {5} and {5, 4, 1})

## Exercise 3.5 OddAndTen

---

- Skeleton code 1:

```
/*
 * Decide if it is possible to divide the integers in a list into two sets,
 * so that the sum of one set is odd, and the sum of the other set is a mult of 10.
 * Every integer must be in one set or the other.
 * For example, oddAndTen([5, 5, 3]) → true,
 * and oddAndTen([5, 5, 4]) → false.
 * @param list is a list of integers.
 * @return true iff there is one odd partition and the other multiple of 10.
 */
public static boolean oddAndTen(List<Integer> list) {

    // call your recursive helper method

}
```

## Exercise 3.5 OddAndTen Helper Method

---

- Skeleton code 2:

```
private static boolean oddAndTenHelper(...) { // add any parameters  
    // base case  
  
    // recursive call  
  
}
```

# Thank you for your attention !

---

- In this lab, you have learned:
  - Creating good test cases
    - Divide into subdomains, one test case for each subdomain
    - Include the boundaries
  - Solving problems using Recursion + Helper Method



# Advanced Object-Oriented Programming

CPT204 – Lab 4  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming Lab 4

**Testing 4, Linked List 1**

# Welcome !

---

- Welcome to Lab 4 !
  - We are going to add methods to our **MyList** implementation iteratively and recursively, either mutate the object or not
- You will find in this lab
  1. Lab Exercise 4.1 - 4.4, and their hints
  2. Exercise 4.1 - 4.4
- Download **lab4** zip files from Learning Mall
- Don't forget to import the **lab4** files and the library into an IntelliJ project
  - Read **lab1** again for reference

## Lab Exercise 4.1 MyList Iterative Square Mutate

---

- Complete the method `void iterSquareMutList(MyList list)` iteratively.  
The method ***modifies/mutates*** list so that all of its elements are squared.  
**Use loops.**
- Test case:  
`list = [1, 2, 3]`  
`MyList.iterSquareMutList(list);`  
`list → [1, 4, 9]`

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 4.1 MyList Iterative Square Mutate Hints

---

- Loop using while as long as list is not null
  - square the value
  - move list to list.next

## Lab Exercise 4.2 MyList Recursive Square Mutate

---

- Complete the method `void recSquareMutList(MyList list)` recursively.  
The method ***modifies/mutates*** list so that all of its elements are squared.  
**Do not** use loops.
- Test case:  
`list = [1, 2, 3]`  
`MyList.recSquareMutList(list);`  
`list → [1, 4, 9]`

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 4.2 MyList Recursive Square Mutate Hints

---

- Base case
  - when list is null, do nothing
  - when list.next is null, square the value
- Recursive step
  - square the value
  - call the method on list.next

## Lab Exercise 4.3 MyList Iterative Square Immutate

---

- Complete the method `MyList iterSquareList(MyList list)` iteratively.  
The method ***does not mutate*** list, but create a new MyList object with all of its elements squared.

**Use loops.**

- Test case 1:

```
list1 = [1, 2, 3]
```

```
MyList list2 = MyList.iterSquareList(list1);
```

```
list1 → [1, 2, 3]
```

```
list2 → [1, 4, 9]
```

the input MyList object  
is unchanged

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 4.3 MyList Iterative Square Immutate Hints

---

- if input list is null, return null
- create a new list result using new and MyList constructor, with squared list's value
- create a pointer ptr pointing to result
- move list to its next
- while list is not null
  - create a new MyList object with squared list value, and store the reference/address in ptr.next
  - move ptr and list to their next
- return result

## Lab Exercise 4.4 MyList Recursive Square Immutate

---

- Complete the method `MyList recSquareList(MyList list)` recursively.  
The method ***does not mutate*** list, but create a new `MyList` object with all of its elements squared.  
**Do not** use loops.
- Test case 1:

```
list1 = [1, 2, 3]
```

```
MyList list2 = myList.recSquareList(list1);
```

```
list1 → [1, 2, 3]
```

```
list2 → [1, 4, 9]
```

the input `MyList` object  
is unchanged

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 4.4 MyList Recursive Square Immutate Hints

---

- Base case
  - when list is null, no value to square so return null
- Recursive step
  - return a new MyList object,  
with square of input list's value as value  
while call the method recursively on list.next and store its result as  
the object's next

# Week 4 Online Programming Assignments

---

- Start with creating a good set of test cases first !
  - Include the corner/boundary cases,  
such as **empty** `MyList`, by: `MyList empty = MyList.ofEntries();`  
if the input is **empty** (which is **null**), then the output is also **null**
- Use IntelliJ to write and test your code, use the ***Java Visualizer!***

## Exercise 4.1 MyList Iterative Catenate Mutate

---

- Complete the method `MyList iterCatMutList(MyList listA, MyList listB)` iteratively, to return a list consisting of all elements of listA, followed by all elements of listB.

The method ***modifies/mutates*** listA so that it is concatenated with listB, if listA is not empty/null.

**Use** loops.

- Test case 1:

`list1 = [1, 2, 3], list2 = [4, 5, 6]`

`list = myList.iterCatMutList(list1, list2);`

`list → [1, 2, 3, 4, 5, 6]`

`list1 → [1, 2, 3, 4, 5, 6]`

list1 is changed

## Exercise 4.1 MyList Iterative Catenate Mutate

---

- Test case 2:

list1 = null, list2 = null

list = myList.iterCatMutList(list1, list2);

list → null

list1 → null

- Test case 3:

list1 = [5], list2 = null

list = myList.iterCatMutList(list1, list2);

list → [5]

list1 → [5]

- Test case 4:

list1 = null, list2 = [5]

list = myList.iterCatMutList(list1, list2);

list → [5]

list1 → null

## Exercise 4.2 MyList Recursive Catenate Mutate

---

- Complete the method `MyList recCatMutList(MyList listA, MyList listB)` recursively, to return a list consisting of all elements of listA, followed by all elements of listB.

The method ***modifies/mutates*** listA so that it is concatenated with listB, if listA is not empty/null.

Do **not** use loops.

- Test case 1:

`list1 = [1, 2, 3], list2 = [4, 5, 6]`

`list = myList.recCatMutList(list1, list2);`

`list → [1, 2, 3, 4, 5, 6]`

`list1 → [1, 2, 3, 4, 5, 6]`

list1 is changed

## Exercise 4.2 MyList Recursive Catenate Mutate

---

- Test case 2:

list1 = null, list2 = null

list = myList.recCatMutList(list1, list2);

list → null

list1 → null

- Test case 3:

list1 = [5], list2 = null

list = myList.recCatMutList(list1, list2);

list → [5]

list1 → [5]

- Test case 4:

list1 = null, list2 = [5]

list = myList.recCatMutList(list1, list2);

list → [5]

list1 → null

## Exercise 4.3 MyList Iterative Catenate Immutate

---

- Complete the method `MyList iterCatList(MyList listA, MyList listB)` iteratively, to return a list consisting of all elements of listA, followed by all elements of listB.

The method ***does not mutate*** listA.

**Use** loops.

- Test case 1:

`list1 = [1, 2, 3], list2 = [4, 5, 6]`

`list = MyList.iterCatList(list1, list2);`

`list → [1, 2, 3, 4, 5, 6]`

`list1 → [1, 2, 3]`

list1 is unchanged

## Exercise 4.3 MyList Iterative Catenate Immutate

---

- Test case 2:

list1 = null, list2 = null

list = myList.iterCatList(list1, list2);

list → null

list1 → null

- Test case 3:

list1 = [5], list2 = null

list = myList.iterCatList(list1, list2);

list → [5]

list1 → [5]

- Test case 4:

list1 = null, list2 = [5]

list = myList.iterCatList(list1, list2);

list → [5]

list1 → null

## Exercise 4.4 MyList Recursive Catenate Immutate

---

- Complete the method `MyList recCatList(MyList listA, MyList listB)` recursively, to return a list consisting of all elements of listA, followed by all elements of listB.

The method ***does not mutate*** listA.

Do **not** use loops.

- Test case 1:

`list1 = [1, 2, 3], list2 = [4, 5, 6]`

`list = myList.recCatList(list1, list2);`

`list → [1, 2, 3, 4, 5, 6]`

`list1 → [1, 2, 3]`

list1 is unchanged

## Exercise 4.4 MyList Recursive Catenate Immutate

---

- Test case 2:

list1 = null, list2 = null

list = myList.recCatList(list1, list2);

list → null

list1 → null

- Test case 3:

list1 = [5], list2 = null

list = myList.recCatList(list1, list2);

list → [5]

list1 → [5]

- Test case 4:

list1 = null, list2 = [5]

list = myList.recCatList(list1, list2);

list → [5]

list1 → null

# Thank you for your attention !

---

- In this lab, you have learned:
  - To create linked list methods *iteratively* and *recursively* that *mutate* or *do not mutate* the input list



# Advanced Object-Oriented Programming

CPT204 – Lab 5  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming Lab 5

**Linked List 2, Deque 1**

# Welcome !

---

- Welcome to Lab 5 !
  - We are going to improve the SLList in Lecture 5 into LLDeque
- You will find in this lab
  1. Lab Exercise 5.1 - 5.4, and their hints
  2. Exercise 5.1 - 5.4
- Download **lab5** zip files from ICE
- Don't forget to import the **lab5** files and the library into an IntelliJ project
  - Read **lab1** again for reference

# Deque

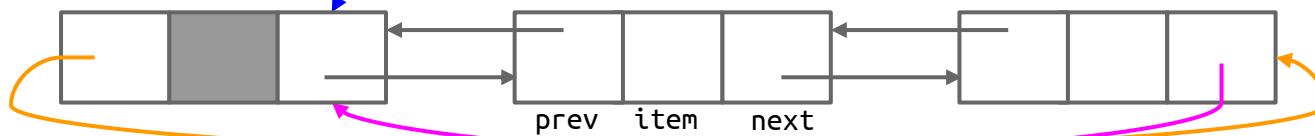
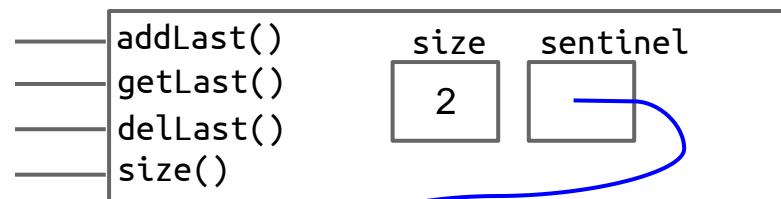
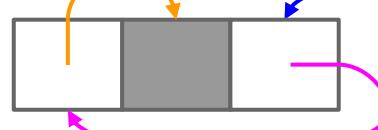
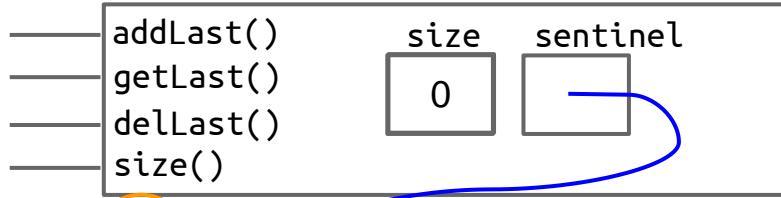
---



- You must have heard about **queue**, a data structure where you can add a new item at the front, and delete an item at the back, so that the first item in is the first item out (FIFO).
- In this lab, we are going to create a more general data structure called **deque**, an acronym of a double-ended queue.
- In a deque, you can add and delete item at both ends (either its front or its back).
  - In this lab, you will complete a number of methods including adding and deleting.
- In addition, you must use **generic types**, so that the deque can be instantiated to store *any type* of objects.

# Lab 5 : Deque using Linked List

- We will implement a deque using Linked List, in particular, **Circular Linked List**



```
private class Node {
```

```
    Node prev;
```

```
    T item;
```

```
    Node next;
```

```
    Node(Node p, T i, Node n) {
```

```
        prev = p;
```

```
        item = i;
```

```
        next = n;
```

```
}
```

```
private Node sentinel;
```

node with an item  
and two pointers

it has **one sentinel**  
that's both the front  
and the back

## Lab Exercise 5.1 - 5.4

---

- Lab Exercise 5.1 LLDeque EMPTY CONSTRUCTOR
  - Lab Exercise 5.2 LLDeque ADD TO FRONT
  - Lab Exercise 5.3 LLDeque PRINT ITEMS
  - Lab Exercise 5.4 LLDeque ITERATIVE GET ITEM
- 
- Hint: Draw and design your code in paper, test using JUnit, and debug using Java Visualizer

# Test Case for Lab Exercise 5.1 - 5.4

---

- Test case 1:

```
LLDeque<String> deque = new LLDeque<>();  
deque.isEmpty();           →      true  
deque.size();             →      0  
deque.addFirst("b");  
deque.addFirst("a");  
deque.iterGet(0);          →      "a"  
deque.iterGet(1);          →      "b"  
deque.printDeque();        →      "a b↔"  
deque.delFirst();  
deque.iterGet(0);          →      "b"
```

## Lab Exercise 5.1 LLDeque EMPTY CONSTRUCTOR

---

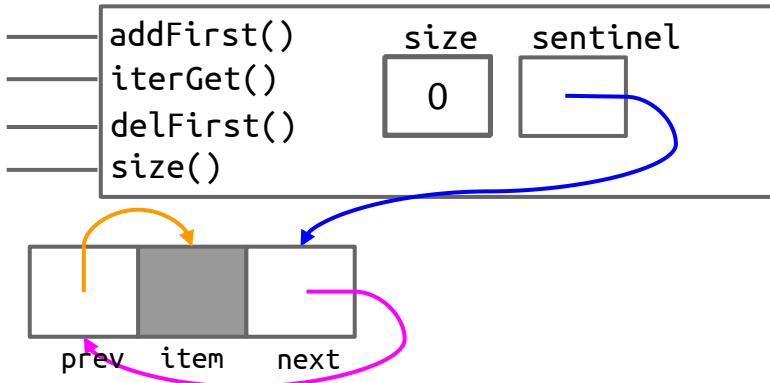
- Complete the empty deque constructor `public LLDeque()`.
- It creates an empty deque.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

# Lab Exercise 5.1 LLDeque EMPTY CONSTRUCTOR Hints

- An empty deque is just a single circular sentinel node, so let us code to create that!



- Create a new node, all null parameters, and set the sentinel to point to it
- Set both its `prev` and `next` to point to itself
- Set `size` to 0

## Lab Exercise 5.2 LLDeque ADD TO FRONT

---

- Complete the method `void addFirst(T item)`.
- It adds an item of type T to the front of the deque.
- It must **not** use any loops or recursion.
- Each operation must take **constant time**, that is,  
it does not depend on the deque's size.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 5.2 LLDeque ADD TO FRONT Hints

---

- We need to create a new node and place it immediately next to the sentinel
- Create a new node with the input item, with its prev points to the sentinel, and with its next point to the old node after the sentinel
- Set the prev of the old node after the sentinel to point to the new node
- Set the next of sentinel to point to the new node
- Increment the size

## Lab Exercise 5.3 LLDeque PRINT ITEMS

---

- Complete the method `void printDeque()`.
- It prints the items in the deque from first to last, separated by a space, ended with a new line.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 5.3 LLDeque PRINT ITEMS Hints

---

- We need to go through every item and print it
- The item starts at next of sentinel, so set a pointer p to it
- While p is not back to sentinel
  - Print item inside node pointed by p using print (not println)
  - Add a spacebar
  - Move p to point to the next node
- Add new line with println

## Lab Exercise 5.4 LLDeque ITERATIVE GET ITEM

---

- Complete the method `T iterGet(int index)` iteratively.
- It returns the item at the given index, where index 0 is the front.  
If no such item exists, it returns null.
- It must **use** loops, and **not** recursion.
- It must **not** mutate the deque.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 5.4 LLDeque ITERATIVE GET ITEM Hints

---

- If the deque is empty, or if index is invalid (negative, greater or equal size) then return null
- Create a node pointer p that starts from next of sentinel
- Use for/while to move the pointer to the index-th node
- Return the item inside node pointed by p

## Exercise 5.1 - 5.4

---

- Exercise 5.1 LLDeque ADD TO BACK
  - Exercise 5.2 LLDeque DELETE FRONT
  - Exercise 5.3 LLDeque DELETE BACK
  - Exercise 5.4 LLDeque RECURSIVE GET ITEM
- 
- Hint: Draw and design your code in paper, test using JUnit, and debug using Java Visualizer

## Test Case for Exercise 5.1 - 5.4

---

- Test case 1:

```
LLDeque<String> deque = new LLDeque<>();  
deque.addLast("a");  
deque.addLast("b");  
deque.addLast("c");  
deque.recGet(2);          →      "c"  
deque.delFirst();         →      "a"  
deque.delLast();          →      "c"  
deque.recGet(0);          →      "b"
```

## Exercise 5.1 LLDeque ADD TO BACK

---

- Complete the method `void addLast(T item)`.
- It adds an item of type T to the back of the deque.
- It must **not** use any loops or recursion.
- Each operation must take **constant time**, that is,  
it does not depend on the deque's size.

## Exercise 5.2 LLDeque DELETE FRONT

---

- Complete the method `T delFirst()`.
- It deletes and returns the item at the front of the deque.  
If no such item exists, returns null.
- It must **not** use any loops or recursion.
- Each operation must take **constant time**, that is,  
it does not depend on the deque's size.

## Exercise 5.3 LLDeque DELETE BACK

---

- Complete the method `T delLast()`.
- It deletes and returns the item at the back of the deque.  
If no such item exists, returns null.
- It must **not** use any loops or recursion.
- Each operation must take **constant time**, that is,  
it does not depend on the deque's size.

## Exercise 5.4 LLDeque RECURSIVE GET ITEM

---

- Complete the method `T recGet(int index)` recursively.
- It returns the item at the given index, where index 0 is the front.  
If no such item exists, it returns null.
- It must **not** use loops.
- It must **not** mutate the deque.

# Thank you for your attention !

---

- In this lab, you have learned:
  - To create a data structure called deque using circular linked list, complete with its methods that is either run in *constant-time, iteratively or recursively*



# Advanced Object-Oriented Programming

CPT204 – Lab 6  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming Lab 6

Exception, Linked List 3,  
Deque 2

# Welcome !

---

- Welcome to Lab 6 !
  - We are going to practice exception and equip LLDeque in Lab 5 with a Copy Constructor and methods that throw and catch an exception
- You will find in this lab
  1. Lab Exercise 6.1, 6.2 and 6.3, and their hints
  2. Exercise 6.1, 6.2 and 6.3
- Download **lab6** zip files from Learning Mall
- Don't forget to import the **lab6** files and the library into an IntelliJ project
  - Read **lab1** again for reference

## Lab Exercise 6.1 Vehicle CONSTRUCTOR

---

- Complete the constructor `public Vehicle(String type, int numWheels)`.
- It initializes the instance variables `type` and `numWheels`.
- Additionally, it must be illegal to construct a Vehicle of type "truck" with less than 4 wheels, or a Vehicle of type "motorcycle" with any number of wheels except 2.
  - Set the constructor so that an Illegal Argument Exception is thrown if the illegal arguments are detected.

## Lab Exercise 6.2 Vehicle TEST CONSTRUCTOR

---

- Complete the method `String testConstructor(String type, int numWheels)`.
- It tests the Vehicle constructor using the exception handling mechanism to determine whether the constructor completed normally, or an illegal argument exception thrown by the constructor in Lab Exercise 6.1 occurred.
- It returns "Vehicle constructed" when the constructor successfully creates a Vehicle object,  
and returns "Illegal number of wheels" when that exception happens.
- You ***must use try-catch*** in your code.

# Test Case for Lab Exercise 6.1 and Lab Exercise 6.2

---

- Test case 1:

Vehicle.testConstructor("car", 4); → "Vehicle constructed"

Vehicle.testConstructor("truck", 3); → "Illegal number of wheels"

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

# Lab Exercise 6.1, 6.2 Vehicle CONSTRUCTOR and TEST Hints

---

- In constructor:
  - check if the type is a truck and the #wheels is less than four or the type is a motorcycle and the #wheels is not two, then throw an `IllegalArgumentException` object
  - initialize the two instance variables
- In test of the constructor:
  - in try:
    - pass the parameter and create a `Vehicle` object
    - return the first string
  - in catch of the `IllegalArgumentException` object ex:
    - return the second string

## Lab Exercise 6.3 Vehicle COPY CONSTRUCTOR

---

- Complete the copy constructor `public Vehicle(Vehicle other)`.
- It creates a deep copy of other.
- Test case:

```
Vehicle v1 = new Vehicle("Type A", 2);
Vehicle v2 = new Vehicle(v1);
System.out.println(v1.getType());           →      "Type A"
System.out.println(v2.getType());           →      "Type A"

v2.setType("Type B");
System.out.println(v1.getType());           →      "Type A"
System.out.println(v2.getType());           →      "Type B"
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 6.3 Vehicle COPY CONSTRUCTOR Hints

---

- In the copy constructor:
  - set both instance variables of the object to the respective instance variables of other
  - access the instance variables of other with dot operator

## Exercise 6.1 LLDeque COPY CONSTRUCTOR

---

- Complete the copy constructor `public LLDeque(LLDeque<T> other)`.
- It creates a deep copy of other.
- Test case 1:

```
LLDeque<String> deque = new LLDeque<>();  
deque.addFirst("a");  
LLDeque<String> copyDeque = new LLDeque<>(deque);  
deque.addFirst("x");  
copyDeque.addFirst("y");  
deque.iterGet(0);           →      "x"  
deque.iterGet(1);           →      "a"  
copyDeque.iterGet(0);       →      "y"  
copyDeque.iterGet(1);       →      "a"
```

## Test Case for Exercise 6.2 and Exercise 6.3

---

- Test case 1:

```
LLDeque<String> deque = new LLDeque<>();  
deque.addFirst("c");  
deque.addLegalFirst("b", "x");  
deque.addLegalFirst(null, "a");  
deque.iterGet(0); → "a"  
deque.iterGet(1); → "b"  
deque.iterGet(2); → "c"
```

## Exercise 6.2 LLDeque ADD NOT NULL TO FRONT

---

- Complete the method `void addFirst(T item)`.
- It adds an item of type T to the front of the deque.
- It must **not** use any loops or recursion.
- Each operation must take **constant time**, that is,  
it does not depend on the deque's size.
- Additionally, if the item is null, instead of adding it into the deque,  
reject and throw an illegal argument exception.

## Exercise 6.3 LLDeque ADD LEGAL ITEM TO FRONT

---

- Complete the method `void addLegalFirst(T item1, T item2)`.
- It adds the first item of type `T` to the front of the deque,  
but if `item1` is an illegal item, it adds the second item instead.
- An illegal item is detected by handling an illegal argument exception,  
thrown by the method `addFirst` in Exercise 6.2.
- It must **not** use any loops or recursion.
- Each operation must take **constant time**, that is,  
it does not depend on the deque's size.
- You **must not use null** in your code.

# Thank you for your attention !

---

- In this lab, you have learned:
  - To create a method that throws an exception,  
especially an unchecked exception called `IllegalArgumentException`
  - To create a method that handles that exception
  - To create a copy constructor doing a deep copy



# Advanced Object-Oriented Programming

CPT204 – Lab 8  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming Lab 8

**Linked List 4, Deque 3,  
Exception 2**

# Welcome !

---

- Welcome to Lab 8 !
  - We are going to implement deque using array: the ARDeque
- You will find in this lab
  1. Lab Exercise 8.1 - 8.4, and their hints
  2. Exercise 8.1 - 8.4
- Download **lab8** zip files from Learning Mall
- Don't forget to import the **lab8** files and the library into an IntelliJ project
  - Read **lab1** again for reference

# ARDeque

---



- In this lab, we are going to implement **deque** using *an array*
  - previously in Lab 5, 6, we implemented deque using linked-list
- We will also use **generic types**, so that the deque can store *any type* of objects
- Here are the additional specifications:
  - The starting size/length of your array must be 4
  - Use the resizing : array doubling and array halving discussed in the lecture
    - before adding, double the size if it's full
    - after deleting, halve the size if it's less than or equal to a quarter full
  - Use **circular array** which is defined in the next slides, followed by examples

# ARDeque

---

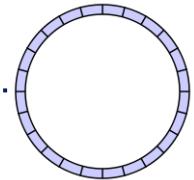


- In this lab, we are going to implement **deque** using *an array*
  - previously in Lab 5, 6, we implemented deque using linked-list
- We will also use **generic types**, so that the deque can store *any type* of objects
- Here are the additional specifications:
  - The starting size/length of your array must be 4
  - Use the resizing : array doubling and array halving discussed in the lecture
    - before adding, double the size if it's full
    - after deleting, halve the size if it's less than or equal to a quarter full
  - Use **circular array** which is defined in the next slides, followed by examples
  - You may define and submit your own private helper method(s)
    - for example, private void **resize** (int capacity)

include in your  
LMO submission

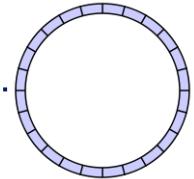
# Circular Array

---



- As the name suggests, think of the array a circular object
  - we keep two indices, **nextFirst** and **nextLast**
  - in the beginning, **nextFirst + 1** is **nextLast**
  - when we **addLast**, we put the new item in index **nextLast**, then shift it to right circularly
  - when we **addFirst**, we put the new item in index **nextFirst**, then shift it to left circularly
  - **delFirst** and **delLast** is also set accordingly

# Circular Array

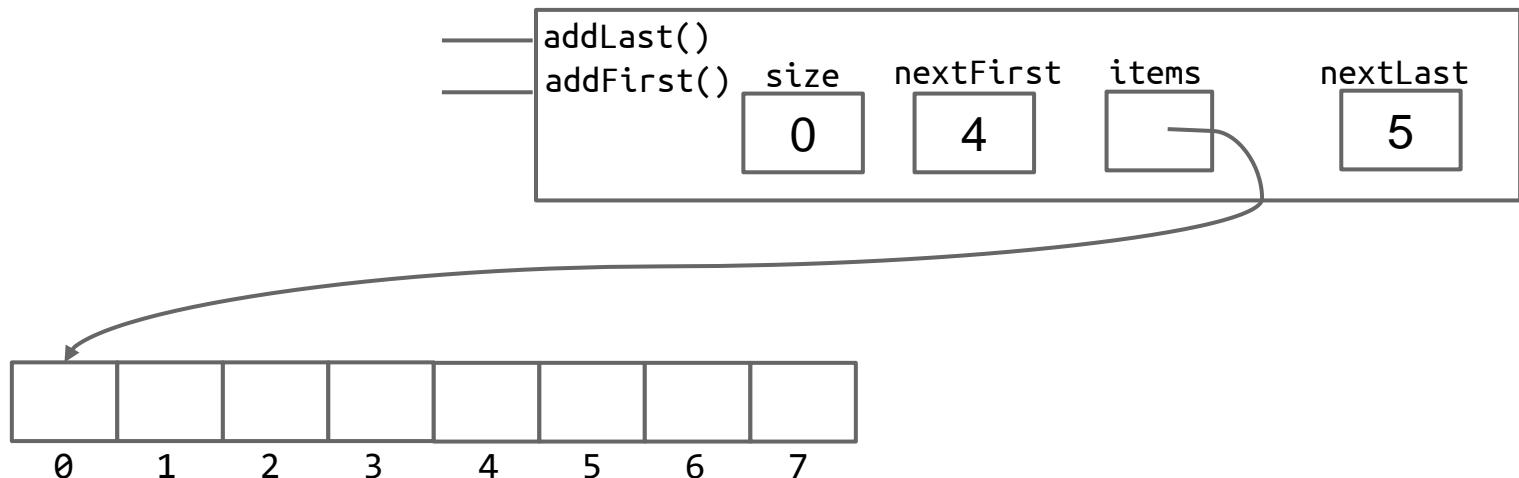


- As the name suggests, think of the array a circular object
    - we keep two indices, **nextFirst** and **nextLast**
    - in the beginning, **nextFirst + 1 is nextLast**
    - when we **addLast**, we put the new item in index **nextLast**, then shift it to right circularly
    - when we **addFirst**, we put the new item in index **nextFirst**, then shift it to left circularly
    - **delFirst** and **delLast** is also set accordingly
  - In the example on the next slides,
    - we start with an empty array of length 8
    - **nextFirst** is 4
    - **nextLast** is 5
- picked arbitrarily as long as  
following the rules above

# Circular Array Example

---

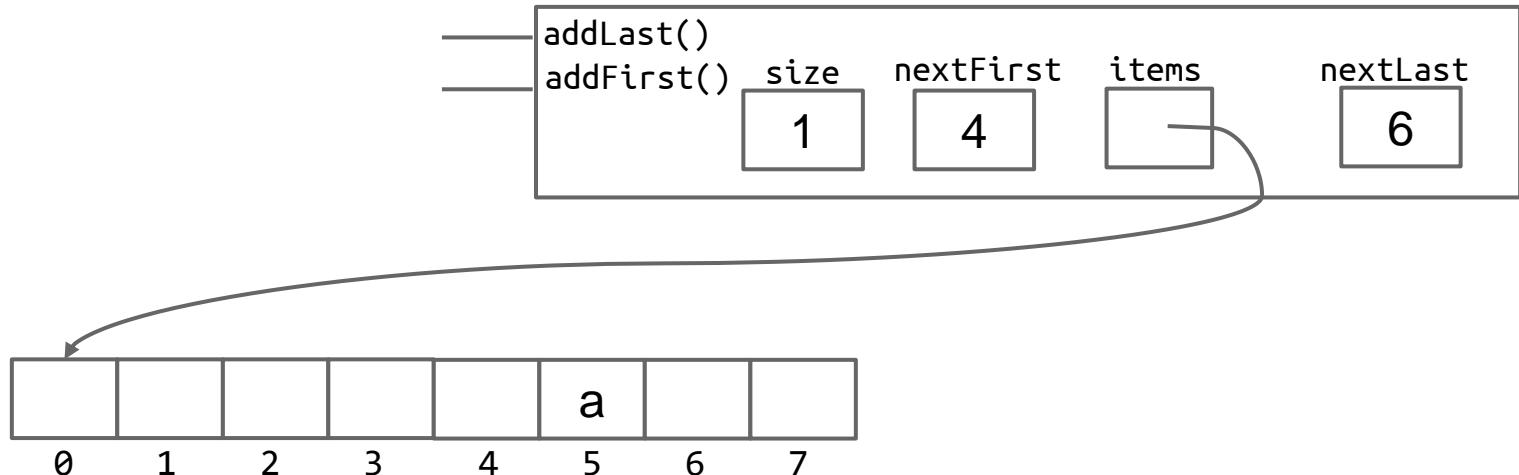
- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")



# Circular Array Example

---

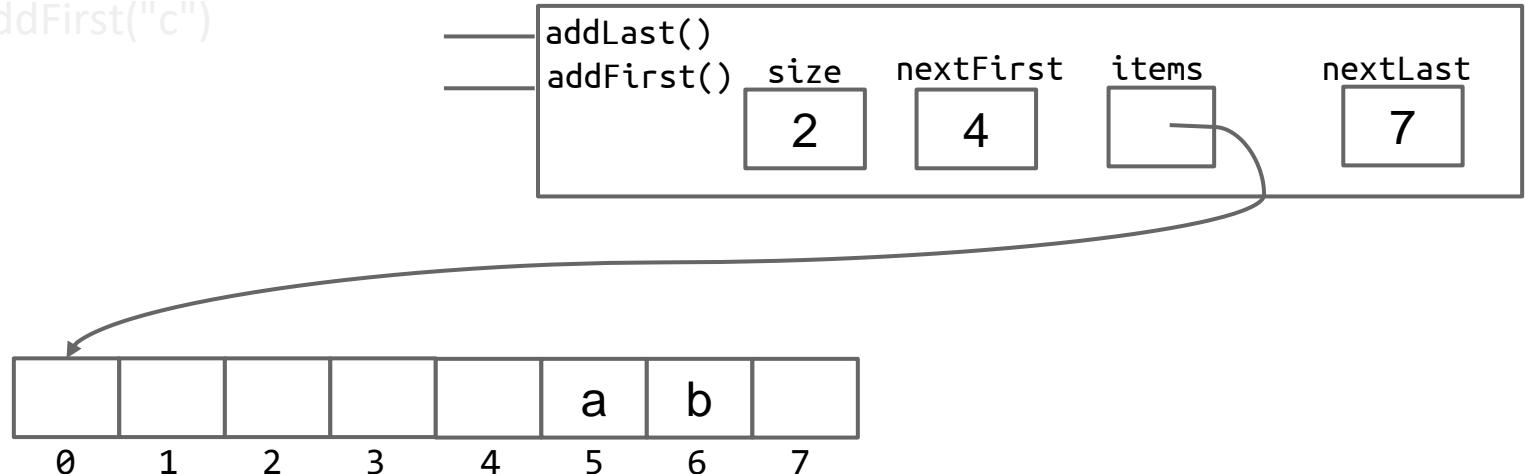
- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")



# Circular Array Example

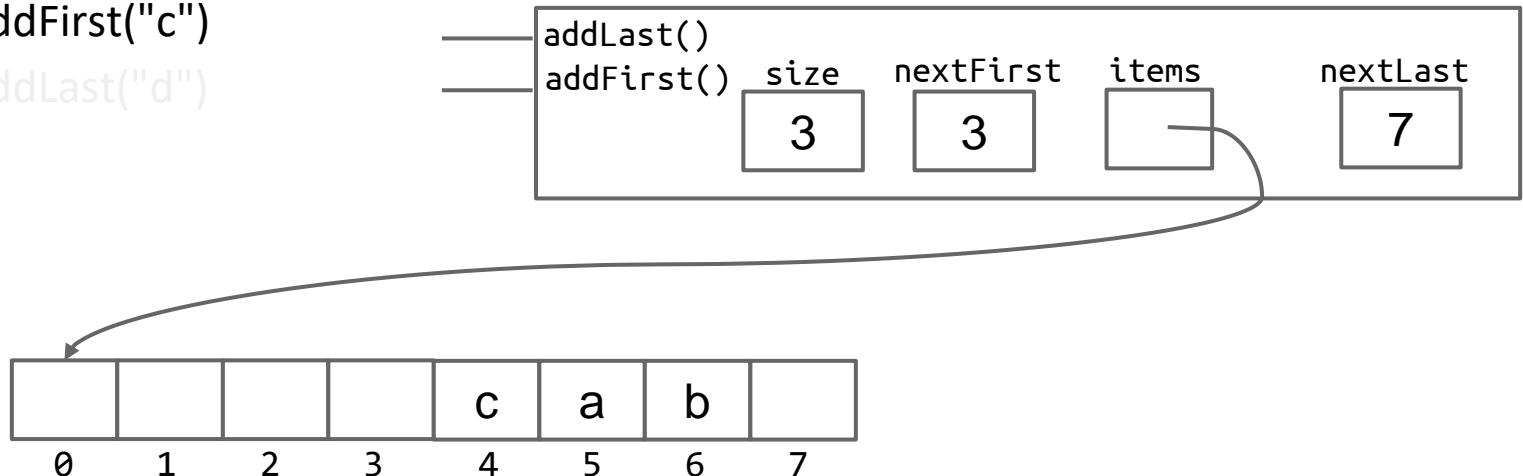
---

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")
  - addFirst("c")



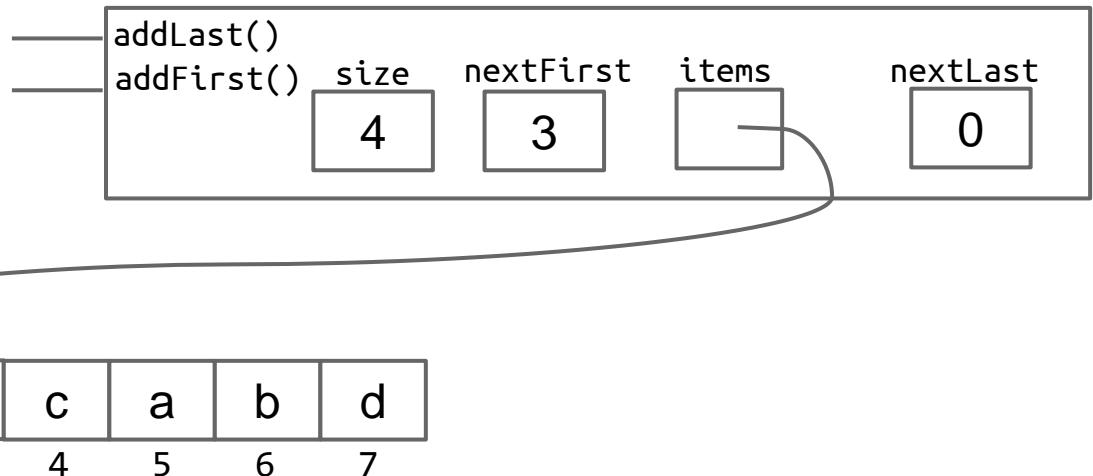
# Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")
  - addFirst("c")
  - addLast("d")



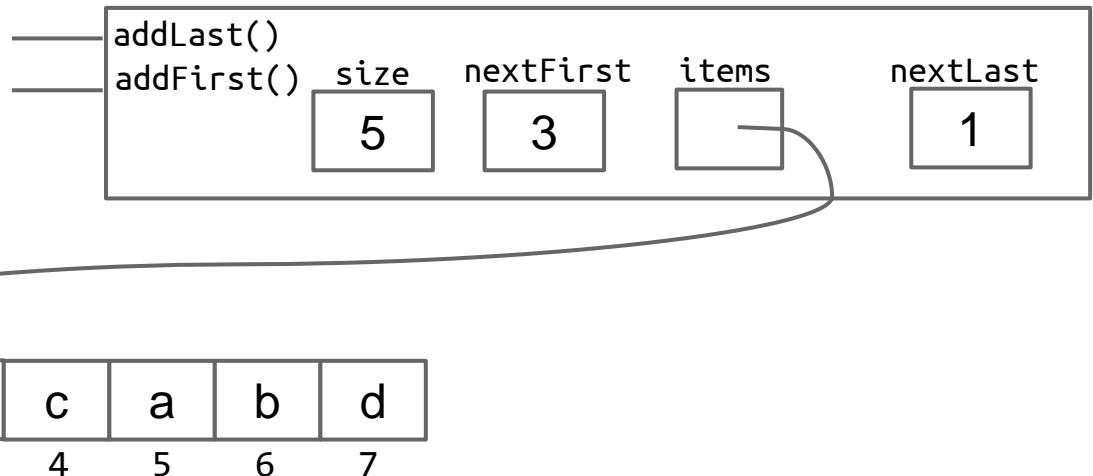
# Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")
  - addFirst("c")
  - addLast("d")
  - addLast("e")



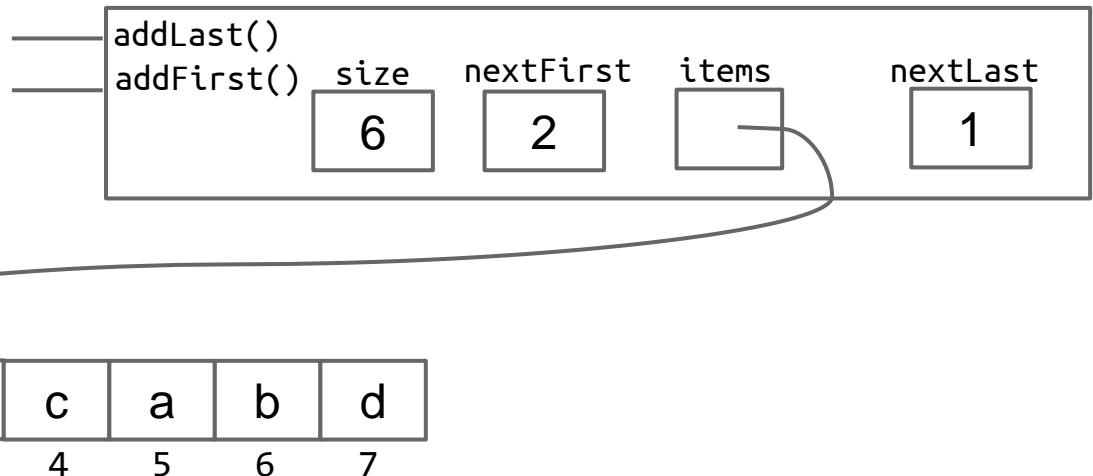
# Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")
  - addFirst("c")
  - addLast("d")
  - addLast("e")
  - addFirst("f")



# Circular Array Example

- In this example, the ARDeque<String> starts with an empty array items of length 8
  - addLast("a")
  - addLast("b")
  - addFirst("c")
  - addLast("d")
  - addLast("e")
  - addFirst("f")



## Lab Exercise 8.1 - 8.4

---

- Lab Exercise 8.1 ARDeque EMPTY CONSTRUCTOR
  - Lab Exercise 8.2 ARDeque ADD TO BACK
  - Lab Exercise 8.3 ARDeque PRINT ITEMS
  - Lab Exercise 8.4 ARDeque GET ITEM
- 
- Hint: Start without resizing/generic first, draw and design your code in paper, unit-test each method *separately* using JUnit, and debug using Java Visualizer

# Test Case for Lab Exercise 8.1 - 8.4

---

- Test case 1:

```
ARDeque<String> deque = new ARDeque<>();  
deque.isEmpty();           →      true  
deque.size();             →      0  
deque.itemsLength();      →      4  
deque.addLast("a");  deque.addLast("b");  deque.addLast("c");  deque.addLast("d");  
deque.size();             →      4  
deque.itemsLength();      →      4  
deque.get(0);              →      "a"  
deque.get(1);              →      "b"  
deque.get(2);              →      "c"  
deque.get(3);              →      "d"  
deque.printDeque();        →      "a b c d↵"  
deque.addLast("e");  
deque.size();             →      5  
deque.itemsLength();      →      8  
deque.get(0);              →      "a"  
deque.get(3);              →      "d"  
deque.get(4);              →      "e"  
deque.printDeque();        →      "a b c d e↵"
```

## Lab Exercise 8.1 ARDeque EMPTY CONSTRUCTOR

---

- Complete the empty deque constructor `public ARDeque()`.
- It creates an empty deque.
- You have to start with an array of length 4.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 8.1 ARDeque EMPTY CONSTRUCTOR Hints

---

- An empty deque is just an array of length 4, so let us code to create that!
- Initialize items with a new Object array of length 4, that is cast into array of T
- Set nextFirst and nextLast to valid indices following the setting of a circular array
  - Read page 6 of this lab notes
- Set size to 0

## Lab Exercise 8.2 ARDeque ADD TO BACK

---

- Complete the method `void addLast(T item)`.
- It adds an item of type T to the back of the deque.
- It must **not** use any loops or recursion, and each operation must take **constant time**, that is, it does not depend on the deque's size, *except* when resizing.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 8.2 ARDeque ADD TO BACK Hints

---

- We need to place the item in the correct index in the array, but we may need to do resizing beforehand!
- If the array is full, we have to resize first, and do array doubling
  - it is better to define private helper method resize with input parameter the new capacity
  - it can be reused by other methods in the exercises/assignments
- Set the item to array items index nextLast
- Increment nextLast circularly
  - it can also be done using a private helper method, useful for others
- Increment the size

## Lab Exercise 8.3 ARDeque PRINT ITEMS

---

- Complete the method `void printDeque()`.
- It prints the items in the deque from first to last, separated by a space, ended with a new line.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 8.3 ARDeque PRINT ITEMS Hints

---

- We need to go through every item in array items and print it
  - thus, we need to compute the real indices first
- The item starts after nextFirst
- The item ends before nextLast and there are size items
- Use while/for and print to display the items separated by a space
- Add a new line with println

## Lab Exercise 8.4 ARDeque GET ITEM

---

- Complete the method `T get(int index)`.
- It returns the item at the given index, where index 0 is the front.
- If no such item exists, throw an **IndexOutOfBoundsException** with message as in the test case 2 below.
- It must **not** use any loops or recursion, and it must **not** mutate the deque.  
Each operation must take **constant time**, that is, it does not depend on the deque's size.
- Test case 2:

```
ARDeque<String> deque = new ARDeque<>();  
deque.addFirst("a");  
try {  
    deque.get(1);  
} catch (IndexOutOfBoundsException e) {  
    System.out.println(e.getMessage());    → "Index 1 is not valid"  
}
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 8.4 ARDeque GET ITEM Hints

---

- If the deque is empty, or if index is invalid (negative, greater or equal size), then throws an object of IndexOutOfBoundsException
  - pass the message into the constructor, including the invalid index
- Compute the real index in the array items
  - relative to the nextFirst, nextLast, length of items
- Return the item at the real index

## Exercise 8.1 - 8.4

---

- Exercise 8.1 ARDeque ADD TO FRONT
  - Exercise 8.2 ARDeque DELETE FRONT
  - Exercise 8.3 ARDeque DELETE BACK
  - Exercise 8.4 ARDeque COPY CONSTRUCTOR
- 
- Hint: Start without resizing/generic first, draw and design your code in paper, unit-test each method *separately* using JUnit, and debug using Java Visualizer

# Test Case for Exercise 8.1 - 8.4

- Test case 1:

```
ARDeque<String> deque = new ARDeque<>();
for (int i=0; i<8; i++) { deque.addFirst("test"); }
deque.size();                                →      8
deque.itemsLength();                         →      8
deque.addLast("test");
deque.size();                                →      9
deque.itemsLength();                         →     16
deque.addFirst("test");
deque.size();                                →     10
deque.itemsLength();                         →     16
for (int i=0; i<5; i++) { deque.delFirst(); }
deque.size();                                →      5
deque.itemsLength();                         →     16
deque.delLast();
deque.size();                                →      4
deque.itemsLength();                         →      8
deque.delFirst();
deque.size();                                →      3
deque.itemsLength();                         →      8
deque.delLast();
deque.size();                                →      2
deque.itemsLength();                         →      4
```

## Exercise 8.1 ARDeque ADD TO FRONT

---

- Complete the method `void addFirst(T item)`.
- It adds an item of type T to the front of the deque.
- It must **not** use any loops or recursion, and each operation must take **constant time**, that is, it does not depend on the deque's size, *except* when resizing.

## Exercise 8.2 ARDeque DELETE FRONT

---

- Complete the method `T delFirst()`.
- It deletes and returns the item at the front of the deque.  
If no such item exists, returns null.
- It must **not** use any loops or recursion, and  
each operation must take **constant time**, that is,  
it does not depend on the deque's size, *except* when resizing.

## Exercise 8.3 ARDeque DELETE BACK

---

- Complete the method `T delLast()`.
- It deletes and returns the item at the back of the deque.  
If no such item exists, returns null.
- It must **not** use any loops or recursion, and  
each operation must take **constant time**, that is,  
it does not depend on the deque's size, *except* when resizing.

## Exercise 8.4 ARDeque COPY CONSTRUCTOR

---

- Complete the copy constructor `public ARDeque(ARDeque<T> other)`.
- It creates a deep copy of other.
- Test case 1:

```
ARDeque<String> deque = new ARDeque<>();  
deque.addFirst("a");  
ARDeque<String> copyDeque = new ARDeque<>(deque);  
deque.addFirst("x");  
copyDeque.addFirst("y");  
deque.get(0);           →      "x"  
deque.get(1);           →      "a"  
copyDeque.get(0);       →      "y"  
copyDeque.get(1);       →      "a"
```

# Thank you for your attention !

---

- In this lab, you have learned:
  - To create a data structure called deque using a circular array, complete with
    - resizing array technique: dynamically expanding and shrinking
    - fast constant-(amortized)-time methods
    - deep copy, and
    - unchecked exception



# Advanced Object-Oriented Programming

CPT204 – Lab 9  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

# CPT204 Advanced Object-Oriented Programming

## Lab 9

### Iterator, Disjoint Sets

# Welcome !

---

- Welcome to Lab 9 !
  - We are going to create an **ARDeque Iterator** for ARDeque,
  - and we are going to implement **Weighted Quick Union** Disjoint Sets
  - You will find the information about them, needed to complete this lab,  
in *the lecture notes*
- You will find in this lab
  1. Lab Exercise 9.1 - 9.4, and their hints
  2. Exercise 9.1 - 9.3
- Download **lab9** zip files from Learning Mall
- Don't forget to import the **lab9** files and the library into an IntelliJ project
  - Read **lab1** again for reference

## Lab Exercise 9.1 ARDequelterator CONSTRUCTOR and HASNEXT

---

- Complete the constructor of **ARDequelterator**, and the method **hasNext**.
- The constructor makes an iterator for ARDeque objects.
- The method tests whether the iterator has more items to return.

## Lab Exercise 9.2 ARDequilterator NEXT

---

- Complete the method `next`.
- It returns the next item, and then advances to item after that in the deque.

# Test Case for Lab Exercise 9.1 and Lab Exercise 9.2

---

- Test case 1:

```
ARDeque<String> deque = new ARDeque<>();
deque.addLast("a");
deque.addLast("b");
deque.addLast("c");

ARDequeIterator<String> iter = new ARDequeIterator<>(deque);
while (iter.hasNext()) {
    String str = iter.next();
    System.out.print(str + " ");
}
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

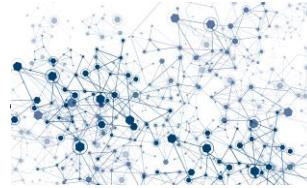
## Lab Exercise 9.1, 9.2 ARDequilterator Hints

---

- Follow the MyIterator for ArrayList<String> implementation in the lecture notes
  - it will be very similar, but using type parameter

# Weighted Quick Union Disjoint Sets

---



- The next exercise and assignments are about the Weighted Quick Union Disjoint Sets data structure
  - we use the parent array and the weight strategy to connect the root of smaller size tree to root of larger size tree
  - if the sizes of the trees are equal in `connect(p, q)`, break the tie by connecting p's root to q's root
- We will implement it using Idea 4.2 described in Lecture Notes page 150
  - that is, store the **negative size** of the trees in the root

# Test Case for Lab Exercise 9.3, Exercise 9.1 - 9.3

---

- Test case 1:

```
WeightedQuickUnionDS ds = new WeightedQuickUnionDS(4);
ds.connect(1, 0);
ds.isConnected(1, 0);           →      true
ds.parent(1);                  →      0
ds.parent(0);                  →      -2
ds.connect(3, 2);
ds.isConnected(2, 1);          →      false
ds.connect(3, 1);
ds.isConnected(2, 1);          →      true
ds.parent(2);                  →      0
ds.sizeOf(1);                  →      4
ds.printParent();              →      -4 0 0 2
```

## Lab Exercise 9.3 WeightedQuickUnionDS CONSTRUCTOR

---

- Complete the constructor `public WeightedQuickUnionDS(int N)`.
- It creates a Disjoint Sets data structure with  $N$  elements, 0 through  $N-1$ .
- Initially, each element is in its own set.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 9.3 WeightedQuickUnionDS CONSTRUCTOR Hints

---

- Each element initially is its own tree, so let us code to create that!
- Initialize the parent array to hold N integers
- Set each integer to represent the negative size of each tree, where each tree initially consists of just one element

## Lab Exercise 9.4 WeightedQuickUnionDS VALIDATE

---

- Complete the method `void validate(int p)`.
- It validates that `p` is a valid element/index.
- If `p` is not a valid index, throw an `IllegalArgumentException` as in the test case 2 below.
- Test case 2:

```
WeightedQuickUnionDS ds = new WeightedQuickUnionDS(5);  
ds.validate(10);    →    IllegalArgumentException
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 9.4 WeightedQuickUnionDS VALIDATE Hints

---

- The valid indices (not the values) are simply the ones set in the constructor, if that is not the case, then throws an object of IllegalArgumentException

## Exercise 9.1 WeightedQuickUnionDS SIZE OF

---

- Complete the method `int sizeOf(int p)`.
- It returns the size of the set element `p` belongs to.

## Exercise 9.2 WeightedQuickUnionDS IS CONNECTED

---

- Complete the method boolean **isConnected(int p, int q)**.
- It return *true* if p and q are connected / in the same set,
  - and *false* otherwise.
- It throws **IllegalArgumentException** if p or q is not a valid index.

## Exercise 9.3 WeightedQuickUnionDS CONNECT

---

- Complete the method `void connect(int p, int q)`.
- It connects two elements `p` and `q` together, by combining the sets containing them, connecting the root of smaller size tree to root of larger size tree.
  - If the sizes of the trees are *equal*, *break the tie* by connecting `p`'s root to `q`'s root.
- It throws `IllegalArgumentException` if `p` or `q` is not a valid index.

# Thank you for your attention !

---

- In this lab, you have learned:
  - To create an iterator class of a data structure
    - hasNext() and next()
  - To create a data structure called Disjoint Sets / Union Find using a provable efficient weighted quick union technique



# Advanced Object-Oriented Programming

CPT204 – Lab 10  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming Lab 10

**Interface, Inheritance,  
Iterable, Set**

# Welcome !

---

- Welcome to Lab 10 !
  - We are going to create an interface for ARDeque and make it iterable,
  - and implement an array-based set ARSet
  - We will also create a new data structure by extending another data structure, and equip it with a new method
- You will find in this lab
  1. Lab Exercise 10.1 - 10.3, and their hints
  2. Exercise 10.1 - 10.3
- Download **lab10** zip files from Learning Mall
- Don't forget to import the **lab10** files and the library into an IntelliJ project
  - Read **lab1** again for reference

## Lab Exercise 10.1 ARDeque DEQUE INTERFACE

---

- Complete the interface of ARDeque called **Deque**.

- It will be implemented by ARDeque as follows:

```
public class ARDeque<T> implements Deque<T> { ... }
```

- In ARDeque: size, methods in Lab Exercise 8.2 - 8.4 and Exercise 8.1 - 8.3 will be annotated with **@Override**

# Lab Exercise 10.1 ARDeque DEQUE INTERFACE Test Case

---

- Test case 1:

```
Deque<String> deque = new ARDeque<>();  
deque.size();                                →      0  
deque.addFirst("a");  
deque.addLast("b");  
deque.addLast("c");  
deque.get(0);                                →      "a"  
deque.get(2);                                →      "c"  
deque.printDeque();                          →      "a b c"  
deque.delFirst();                            →      "a"  
deque.delLast();                            →      "c"  
deque.get(0);                                →      "b"  
deque.size();                                →      1
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 10.1 ARDeque DEQUE INTERFACE Hints

---

- Use keyword interface and type parameter
- There will be seven method signatures
  - end with semicolon

## Lab Exercise 10.2 RSLList ROTATE RIGHT

---

- Complete the class public **RSLList** and method void **rotateRight()**.
- RSLList is a subclass of SLList.
- It implements a new method called `rotateRight` that moves the back item to the front.
  - for example, suppose we have `["a", "b", "c"]`, after `rotateRight`, we will have `["c", "a", "b"]`.
  - do nothing if the list is empty.
- The class will be instantiated and the method will be called as in the test case on the next page.

## Lab Exercise 10.2 RSLList ROTATE RIGHT Test Case

---

- Test case 1:

```
RSLList<String> rlist = new RSLList<>();  
rlist.addLast("a");  
rlist.addLast("b");  
rlist.addLast("c");  
rlist.rotateRight();  
rlist.get(0);           →      "c"  
rlist.get(1);           →      "a"  
rlist.get(2);           →      "b"
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

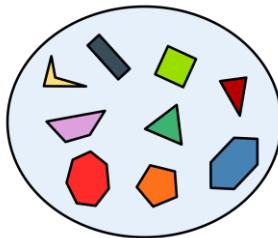
## Lab Exercise 10.2 RSLList ROTATE RIGHT Hints

---

- Set RSLList to be a subclass of SLList
  - using the keyword extend, followed by SLList with the same type parameter
- Write the definition of rotateRight
  - you can simply use methods inherited from SLList to implement this method
  - you need to take special care if the list is empty

# Sets

---



- The next exercise and two assignments are about Sets
  - Set stores a group of values with **no** duplicates
  - Set has **no** sense of order: either an item is in the set, or not
- We will implement it using an array and generics
  - The array has size 100
  - We assume that the items added will **not** exceed 100

## Lab Exercise 10.3 ARSet ITERATOR

---

- Complete the method **iterator**, and the private class **ARSetIterator**.
- The class ARSet implements the Iterable interface.
  - The `java.util.Iterator` is imported.
- The iterator method and the private class will be used in for-each / enhanced for loop as in the case next page.
  - Although the order of items in a set actually does not matter, iterate in the order of the items *added*.

## Lab Exercise 10.3 ARSet ITERATOR Test Case

---

- Test case 1:

```
ARSet<String> set = new ARSet<>();  
set.add("a");  
set.add("b");  
set.add("c");  
  
for (String item : set) {  
    System.out.print(item + " ");  
}  
→ "a b c "
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 10.3 ARSet ITERATOR Hints

---

- Follow the implementation to enable the for-each/enhanced for loop in the lecture notes
  - it will be very similar

## Exercise 10.1 ARSet CONTAINS

---

- Complete the method `boolean contains(T item)`.
- It checks whether an item is inside the set.
  - the method returns *true* iff the set contains the item.

## Exercise 10.2 ARSet ADD

---

- Complete the method `void add(T item)`.
- It adds an item into the set if it is **not** already inside.
- It throws `IllegalArgumentException` if item is *null*.

## Test Case for Exercise 10.1, 10.2

---

- Test case 1:

```
ARSet<String> set = new ARSet<>();
set.add("a");
set.contains("a");                                →      true
set.size();                                       →      1
for (String item : set) {
    System.out.println(item);
}
```

→ "a"

## Exercise 10.3 ARDeque ITERATOR

---

- Complete the method **iterator**, and the private class **ARDequelterator**.
- The class ARDeque implements the Iterable interface.
  - The `java.util.Iterator` is imported.
- The iterator method and the private class will be used in for-each / enhanced for loop as in the case next page.
  - Iterate from the *first* to the *last* item in the deque.

## Exercise 10.3 ARDeque ITERATOR Test Case

---

- Test case 1:

```
ARDeque<String> deque = new ARDeque<>();  
deque.addLast("a");  
deque.addLast("b");  
deque.addLast("c");  
  
for (String item : deque) {  
    System.out.print(item + " ");  
}  
→ "a b c "
```

# Thank you for your attention !

---

- In this lab, you have learned:
  - To create an interface
  - To create a subclass and to use the superclass's methods to create a new method
  - To equip a data structure with an iterator to enable enhanced for loop
  - To create a data structure called Set

|          |  |
|----------|--|
| The Best |  |
| Amazing  |  |
| Great    |  |
| Fine     |  |
| Bad      |  |
| Terrible |  |

# Advanced Object-Oriented Programming

CPT204 – Lab 11  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming Lab 11

**Comparable, Equals,  
Set 2**

# Welcome !

---

- Welcome to Lab 11 !
  - We are going to make our classes and data structures support equality, and their objects can be compared to each other!
- You will find in this lab
  1. Lab Exercise 11.1, and 11.2, and their hints
  2. Exercise 11.1, and 11.2
- Download **lab11** zip files from Learning Mall
- Don't forget to import the **lab11** files and the library into an IntelliJ project
  - Read **lab1** again for reference

## Lab Exercise 11.1 Duration COMPARETO

---

- Complete the method `int compareTo(Duration other)` of `Duration` which implements Comparable.
- It returns a positive, zero, negative number if its duration is larger than, equal to, smaller than other's duration, respectively.
- Test case 1:

```
Duration d1 = new Duration(2, 5);
```

```
Duration d2 = new Duration(1, 2);
```

```
d1.compareTo(d2) > 0 → true
```

```
d2.compareTo(d1) < 0 → true
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 11.1 Duration COMPARETO Hints

---

- Use the trick in the lecture for a one-liner code

## Lab Exercise 11.2 Dog EQUALS

---

- Complete the method boolean **equals**(Object that) of Dog that overrides the one in Object.
- To be the same, they must share the same name and weight.
- The method must satisfy the Object Contract.
- Test case 1:

```
Dog d1 = new Dog("Baobei", 5);
```

```
Dog d2 = new Dog("Baobei", 5);
```

```
Dog d3 = new Dog("Jiaozi", 7);
```

```
d1.equals(d2); → true
```

```
d1.equals(d3); → false
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 11.2 Dog EQUALS Hints

---

- Similar to the equals implementation example in the lecture

# Triple

---



- We want to write a generic class `Triple<T, S, U>` that stores triples of objects, each of which can have an arbitrary type
- In addition, we also want to be able to compare two `Triple` objects
  - To compare two triples, compare the first elements of each; if they are the same, compare the second elements; and if they are the same, compare the third
- Of course, it is required that the first, second and third element in the triple are comparable
  - we declare it using extends on the generic types as follows:

```
public class Triple<T extends Comparable<T>, S extends Comparable<S>,
U extends Comparable<U>> implements Comparable<Triple<T, S, U>>
```

## Exercise 11.1 Triple COMPARETO

---

- Complete the method `int compareTo(Triple<T, S, U> other)` of a comparable Triple.
- It must compare the first element first, followed by the second and third as in the following test case:
- Test case 1:

```
Triple<Integer, String, Double> t1 = new Triple<>(1, "b", 2.0);
Triple<Integer, String, Double> t2 = new Triple<>(1, "a", 5.0);
Triple<Integer, String, Double> t3 = new Triple<>(1, "b", 2.0001);
t1.compareTo(t2) > 0                      →      true
t1.compareTo(t3) < 0                      →      true
```

## Exercise 11.2 ARSet EQUALS

---

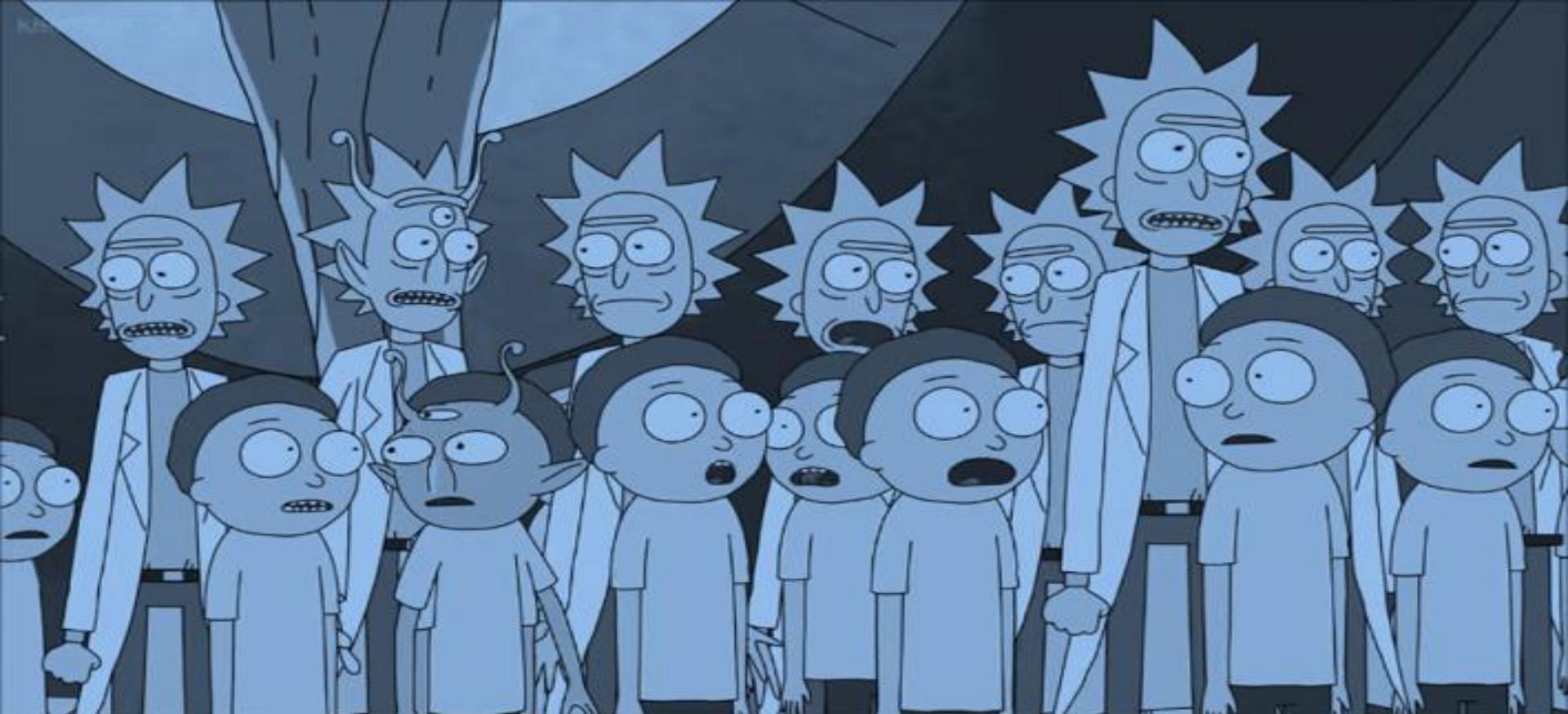
- Complete the method boolean **equals(Object that)** of ARSet that overrides the one in Object.
- To be the same, they must share the same items.
- The method must satisfy the Object Contract.
- Test case 1:

```
ARSet<String> set1 = new ARSet<>();  
set1.add("a");  set1.add("b");  set1.add("c");  
ARSet<String> set2 = new ARSet<>();  
set2.add("b");  set2.add("c");  set2.add("a");  
set1.equals(set2);          →      true  
set2.add("d");  
set1.equals(set2);          →      false
```

# Thank you for your attention !

---

- In this lab, you have learned:
  - To make your data structure comparable
  - To equip your data structure with an equality method that adheres to the Object Contract



# Advanced Object-Oriented Programming

CPT204 – Lab 12  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming Lab 12

**Comparator, Hash Table,  
Map**

# Welcome !

---

- Welcome to Lab 12 !
  - We are going to create a comparator for our hash-based set (hash table) based on the size,
  - and to create a hash-based map with average constant-time operations!
- You will find in this lab
  1. Lab Exercise 12.1 - 12.4, and their hints
  2. Exercise 12.1 - 12.3
- Download **lab12** zip files from Learning Mall
- Import the **lab12** files and the library to an IntelliJ project
  - Read **lab1** again for reference

# Lab Exercise 12.1 HASet SIZE COMPARATOR

---

- Complete the inner non-static class SizeComparator and its getter method getSizeComparator() to create a comparator of two sets of HASet based on their size.
- Test case 1:

```
Comparator<HASet<String>> sizeComp = new HASet<String>().getSizeComparator();
HASet<String> set1 = new HASet<>();
set1.add("a");
set1.add("b");
HASet<String> set2 = new HASet<>();
set2.add("c");
sizeComp.compare(set1, set2) > 0           →      true
set2.add("d");
sizeComp.compare(set1, set2)                 →      0
```

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 12.1 HASet SIZE COMPARATOR Hints

---

- Follow the design pattern discussed in the lecture, except that it is non-static.

# Map

---



- **Map**, also known as Dictionary, is a data structure that stores key and value pairs (see Week 3)
  - you can add/put and remove pairs of key and value (mappings)
  - but most importantly, it supports **fast** searching of value based on its key
- We are going to implement Hash-based Map using
  - ArrayList to store and also to implement the buckets
  - HashSet (see next slide) to store all your keys
  - Separate Chaining and Resizing Techniques (see lecture slides) to achieve average constant-time operations
- Implement resizing such that the capacity is doubled when the load factor  $N/M$  exceeds the given loadFactor
  - there will be default values given in the skeleton code

# HashSet

---



- Hash tables are the most popular implementation for sets
- In Java, they are implemented as `java.util.HashSet`
  - it can store any objects
  - you can store new items by `add`, and check membership by `contains`
  - it is Iterable
- You are using it to store keys in your HAMap
  - and for implementing many methods in the exercises and assignments
- Thus, in this lab, we are using `ArrayList`, `HashSet` and iterator libraries
  - you are not allowed to import any other libraries

# Test Case for Lab Exercise 12.2 - 12.4, Exercise 12.1 - 12.3

---

- Test case 1:

```
HAMap<String, Integer> map = new HAMap<>();  
  
map.containsKey("a");           → false  
map.put("a", 1);  
map.containsKey("a");           → true  
map.get("a");                 → 1  
map.size();                   → 1  
map.put("b", 2);  
map.put("c", 3);  
map.remove("a", 1);           → 1  
for (String key : map) {  
    System.out.println("(" + key + ", " + map.get(key) + ")");    → (b, 2)  
}  
   (c, 3)  
map.clear();  
map.size();           → 0  
map.containsKey("b");           → false  
map.containsKey("c");           → false
```

## Lab Exercise 12.2 HAMap CONSTRUCTOR

---

- Complete **three constructors** of HAMap that take zero, one, or two arguments.
- Use the given default values for the absent arguments.
  - DEFAULT\_CAPACITY = 16 and DEFAULT\_LOAD\_FACTOR = 1.5.
- Initialize all five member variables,
  - with the initialCapacity being the starting numBuckets.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 12.2 HAMap CONSTRUCTOR Hints

---

- you can implement the constructor taking two arguments first,
  - and then calling it for the other two constructors passing the default value(s).
- initialize the buckets list with initialCapacity empty ArrayLists.
- initialize the set of key with an empty HashSet.

## Lab Exercise 12.3 HAMap CLEAR

---

- Complete the method `void clear()` of HAMap that removes all the entries in the map.
- Keep the current number of buckets the same.
  - We do not implement halving/resizing down this time.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 12.3 HAMap CLEAR Hints

---

- reset the buckets and set of keys to the empty buckets and an empty set.
- reset the number of entries.

## Lab Exercise 12.4 HAMap CONTAINSKEY and ITERATOR

---

- Complete the method `boolean containsKey(K key)` and method `Iterator<K> iterator()` of HAMap.
- *containsKey* returns true if the entry with the specified key exists in the map.
  - returns false otherwise.
- *iterator* returns an Iterator that iterates over the stored keys.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 12.4 HAMap CONTAINSKEY and ITERATOR Hints

---

- use the contains and iterator methods of the HashSet.

## Exercise 12.1 HAMap GET

---

- Complete the method `V get(K key)` of HAMap.
- It returns the value to which the specified key is mapped,
  - and return `null` if this map contains *no* entries of the key.

## Exercise 12.2 HAMap PUT

---

- Complete the method `void put(K key, V value)` of HAMap that adds the key, value entry into the map.
- If the same key is added more than once, then the value must be *replaced* each time.
- Assume that *null* keys will *never* be added.
- Before adding the entry, check if the ratio of the number of entries and the number of buckets exceeds the load factor.
  - if so, resize and double the number of buckets, even if *no* new entry is added after.

## Exercise 12.3 HAMap REMOVE

---

- Complete the method `V remove(K key, V value)` of HAMap.
- It removes the entry for the specified key only if it is currently mapped to the specified value.
  - in that case, return the value.
- If the key, value entry does **not** exist in the map, return *null*.
- We do not implement halving/resizing down this time.

# Thank you for your attention !

---

- In this lab, you have learned:
  - To make a comparator comparing your data structure according to your preference
  - To create a map/dictionary using hash table techniques, with ArrayList and HashSet as the underlying data structures



# Advanced Object-Oriented Programming

CPT204 – Lab 13  
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大学

## CPT204 Advanced Object-Oriented Programming Lab 13

# Priority Queue

# Welcome !

---

- Welcome to Lab 13 !
- We are going to create a priority queue with **an resizing array based binary heap** that support logarithmic-time operations
  - double the size of the array if it is full
- You will find in this lab
  1. Lab Exercise 13.1 - 13.4, and their hints
  2. *No Exercises* this week, you can use your time to complete Lab 14 Part A, B : Explicit MinPQ and its application
- Download **lab13** zip files from Learning Mall
- Import the **lab13** files and the library to an IntelliJ project
  - Read **lab1** again for reference

# Test Case for Lab Exercise 13.1 - 13.4

---

- Test case 1:

```
ARBinHeap<Integer> pq = new ARBinHeap<>();  
pq.isEmpty();                                → true  
pq.size();                                    → 0  
  
pq.add(6);  pq.add(3);  pq.add(9);  
pq.getMin();                                  → 3  
  
pq.add(7);  pq.add(5);  pq.add(8);  pq.add(2);  
  
Object[] arr = pq.toArray();                  → [ null 2 5 3 7 6 9 8 ]  
  
pq.delMin();                                 → 2  
  
arr = pq.toArray();                          → [ null 3 5 8 7 6 9 ]
```

## Lab Exercise 13.1 ARBinHeap CONSTRUCTORS

---

- Complete **two constructors** of ARBinHeap that take zero or one argument.
- It initializes an empty binary heap with the given initial capacity,
  - or initial capacity 1 for the one with no arguments.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 13.1 ARBinHeap CONSTRUCTORS Hints

---

- you can implement the constructor taking one arguments first,
  - and then calling it for the empty constructor passing the default value of 1
- initialize the array heap with array of Comparable
  - need to add 1 to the initial capacity because we don't use the first array element
  - and then cast to an array of type parameter T
- initialize size

## Lab Exercise 13.2 ARBinHeap GETMIN

---

- Complete the method `T getMin()` of ARBinHeap.
- It returns a smallest item on this binary heap,
  - and this binary heap must not be empty.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 13.2 ARBinHeap GETMIN Hints

---

- simply returns the root of the binary heap

## Lab Exercise 13.3 ARBinHeap ADD

---

- Complete the method `void add(T item)` of ARBinHeap.
- It adds a new item to this binary heap.
- Double the size of the array if the array is full.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 13.3 ARBinHeap ADD Hints

---

- double the size of the array if it is full
  - use a helper method
- increment size
- place the new item at the next element in the array
  - to keep the tree complete
- swim that new item
  - to maintain the heap property

## Lab Exercise 13.4 ARBinHeap DELMIN

---

- Complete the method `T delMin()` of ARBinHeap.
- It removes and returns a smallest item on this binary heap,
  - and this binary heap must not be empty.
- We do not implement halving/resizing down this time.

**WARNING:** Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

## Lab Exercise 13.4 ARBinHeap DELMIN Hints

---

- store the root value
  - to be returned at the end of the method
- swap the root with the last item
  - to keep the tree complete
- decrement size
- sink that new root
  - to maintain the heap property
- nullify the reference to the deleted item to avoid loitering

# Test Case for Lab Exercise 13.1 - 13.4

---

- Test case 1:

```
ARBinHeap<Integer> pq = new ARBinHeap<>();  
pq.isEmpty();                                → true  
pq.size();                                    → 0  
  
pq.add(6);  pq.add(3);  pq.add(9);  
pq.getMin();                                  → 3  
  
pq.add(7);  pq.add(5);  pq.add(8);  pq.add(2);  
  
Object[] arr = pq.toArray();                  → [ null 2 5 3 7 6 9 8 ]  
  
pq.delMin();                                 → 2  
  
arr = pq.toArray();                          → [ null 3 5 8 7 6 9 ]
```

# Thank you for your attention !

---

- In this lab, you have learned:
  - To create a priority queue that supports logarithmic-time operations, with resizing array as the underlying data structure