

Documenting, Using, and Testing Utility Classes

Readings: Chapter 2 of the Course Notes

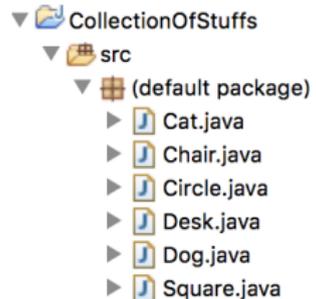


EECS2030: Advanced
Object Oriented Programming
Fall 2017

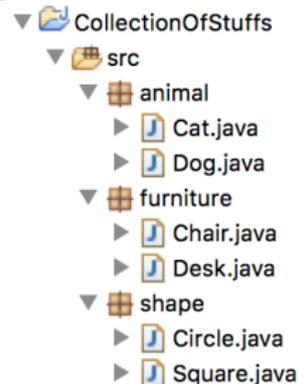
CHEN-WEI WANG

Structure of Project: Packages and Classes

A Java *project* may store a list of Java *classes*.



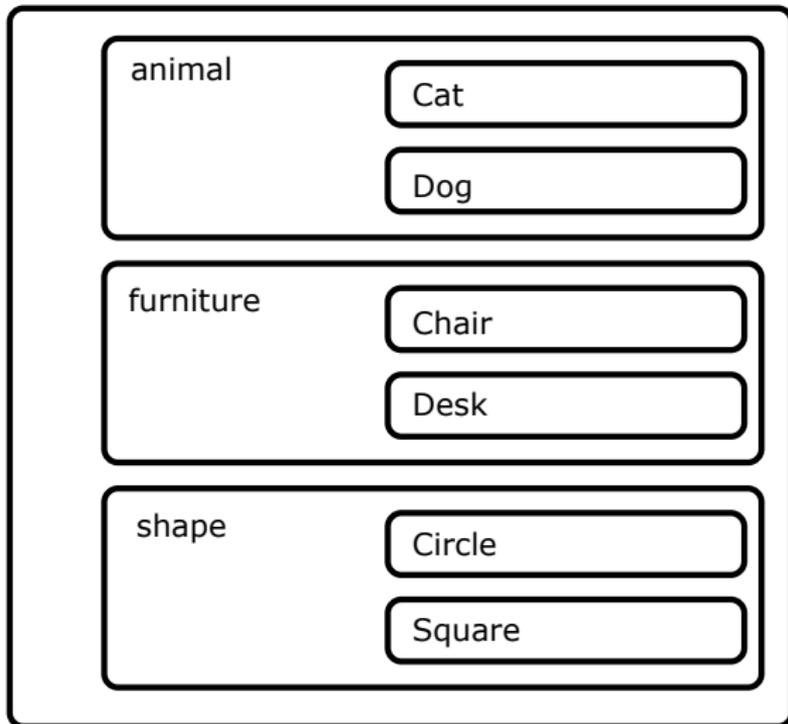
You may group each list of related classes into a *package*.



To see project structure in Eclipse: Package Explorer **view**.

Visibility: Project, Packages, Classes

CollectionOfStuffs

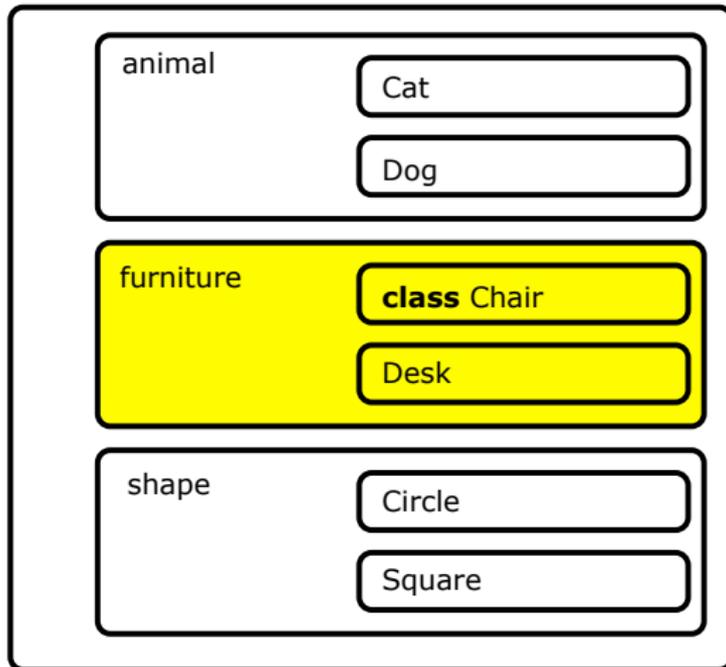


Visibility of Classes

- Only one modifier for declaring visibility of classes: *public*.
- Use of *private* is forbidden for declaring a class.
e.g., *private class* Chair is **not** allowed!!
- **Visibility** of a class may be declared using a modifier, indicating that it is accessible:
 1. Across classes within its resident package [no modifier]
e.g., Declare **class** Chair { ... }
 2. Across packages [*public*]
e.g., Declare *public class* Chair { ... }
- Consider class Chair in: Resident package furniture;
Resident project CollectionOfStuffs.

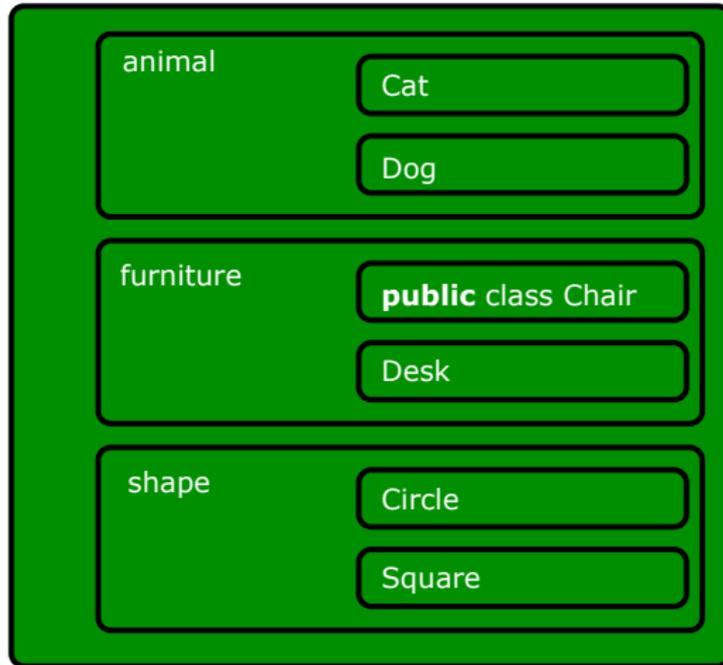
Visibility of Classes: Across All Classes Within the Resident Package (no modifier)

CollectionOfStuffs



Visibility of Classes: Across All Classes Within the Resident Package (no modifier)

CollectionOfStuffs

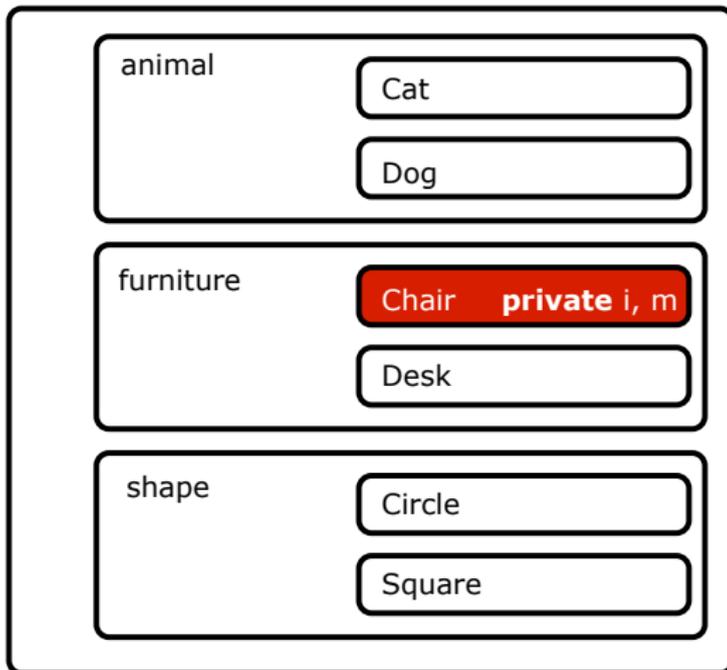


Visibility of Attributes/Methods: Using Modifiers to Define Scopes

- Two modifiers for declaring visibility of attributes/methods:
public and *private*
- **Visibility** of an attribute or a method may be declared using a modifier, indicating that it is accessible:
 1. Within its resident class (*most* restrictive) [*private*]
 - e.g., Declare attribute *private* static int i;
 - e.g., Declare method *private* static void m() {};
 2. Across classes within its resident package [no modifier]
 - e.g., Declare attribute static int i;
 - e.g., Declare method static void m() {};
 3. Across packages (*least* restrictive) [*public*]
 - e.g., Declare attribute *public* static int i;
 - e.g., Declare method *public* static void m() {};
- Consider *i* and *m* in: Resident class `Chair`; Resident package `furniture`; Resident project `CollectionOfStuffs`.

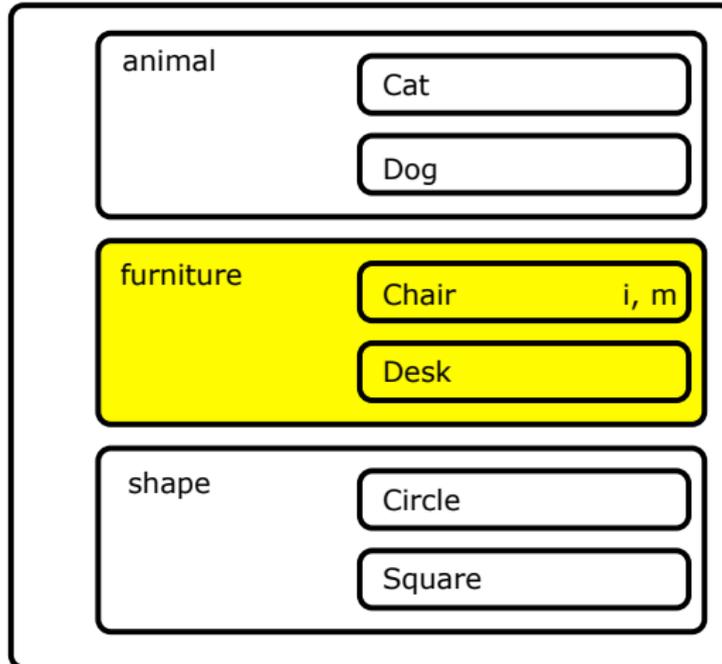
Visibility of Attr./Meth.: Across All Methods Within the Resident Class (*private*)

CollectionOfStuffs



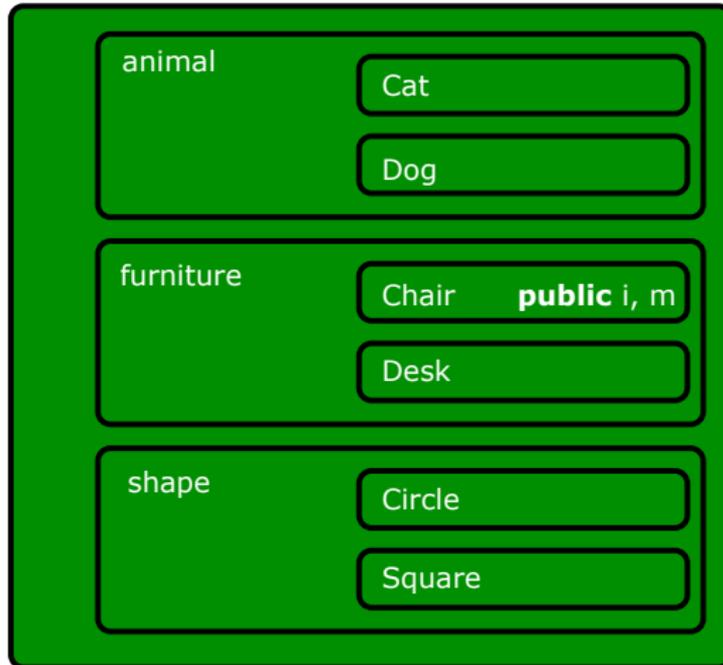
Visibility of Attr./Meth.: Across All Classes Within the Resident Package (no modifier)

CollectionOfStuffs



Visibility of Attr./Meth.: Across All Packages Within the Resident Project (`public`)

CollectionOfStuffs



Structure of Utility Classes

- **Utility classes** are a special kind of classes, where:
 - All **attributes** (i.e., stored data) are declared as *static*.
 - All **methods** (i.e., stored operations) are declared as *static*.
- For now, understand all these *static* attributes and methods collectively make their resident utility class a **single** (i.e., one that cannot be duplicated) machine, upon which you may:
 - Access the value of a data item. [attribute]
 - Compute and return a value. [accessor]
 - Computer and change the data (without returning). [mutator]
- We will later discuss non-static attributes and methods.

To see class structure in Eclipse: `Outline view`.

Structure of Utility Classes: Example (1.1)

```
1 public class CircleUtilities {
2     private static final int RADIUS_TO_DIAMETER = 2;
3     static int radius = 10;
4     public static final int PI = 3;
5     static int getDiameter() {
6         int diameter = radius * RADIUS_TO_DIAMETER;
7         return diameter;
8     }
9     static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10    static void setRadius(int newRadius) { radius = newRadius; }
11    public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12    public static int getCircumference1() { return getDiameter() * PI; }
13    private static int getCircumference2() { return getCircumference(radius); }
14 }
```

Three independent groups of modifiers in the above utility class:

1. **Access**: *private* (L2, L13), *public* (L4, L11, L12), and no access modifier (L3, L5, L9, L10).
2. **Uniqueness**: *static* (all attributes and methods) and non-static (not in a utility class)
3. **Assignable**: *final* (L2, L4) means it is a constant value and can never be assigned, and non-final attributes are variables.

Structure of Utility Classes: Example (1.2)

```
1 public class CircleUtilities {
2     private static final int RADIUS_TO_DIAMETER = 2;
3     static int radius = 10;
4     public static final int PI = 3;
5     static int getDiameter() {
6         int diameter = radius * RADIUS_TO_DIAMETER;
7         return diameter;
8     }
9     static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10    static void setRadius(int newRadius) { radius = newRadius; }
11    public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12    public static int getCircumference1() { return getDiameter() * PI; }
13    private static int getCircumference2() { return getCircumference(radius); }
14 }
```

Each utility class contains a list of attributes and methods:

- L2 – L4:** Three attributes `RADIUS_TO_DIAMETER`, `radius`, `PI`
 - Each of these attributes has an initial value (2, 10, and 3).
 - Only the value of `radius` (non-final) may be changed.
- L5 – L13:** Six methods:
 - 1 Mutator** (with the return type `void`): `setRadius(int newRadius)`
 - 5 Accessors** (with an explicit return statement):
e.g., `getDiameter()`, `getCircumference(int radius)`

Structure of Utility Classes: Example (1.3)

```
1 public class CircleUtilities {
2     private static final int RADIUS_TO_DIAMETER = 2;
3     static int radius = 10;
4     public static final int PI = 3;
5     static int getDiameter() {
6         int diameter = radius * RADIUS_TO_DIAMETER;
7         return diameter;
8     }
9     static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10    static void setRadius(int newRadius) { radius = newRadius; }
11    public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12    public static int getCircumference1() { return getDiameter() * PI; }
13    private static int getCircumference2() { return getCircumference(radius); }
14 }
```

Each method has a (possibly empty) list of **parameters** (i.e., inputs) and their types:

- e.g., `getDiameter` (**L5**) has no parameters (i.e., it takes no inputs for its computation)
- e.g., `setRadius` (**L10**) has one parameter (i.e., `newRadius` of type `int`)

We talk about **parameters** in the context of method declarations.

Structure of Utility Classes: Example (1.4)

```
1 public class CircleUtilities {
2     private static final int RADIUS_TO_DIAMETER = 2;
3     static int radius = 10;
4     public static final int PI = 3;
5     static int getDiameter() {
6         int diameter = radius * RADIUS_TO_DIAMETER;
7         return diameter;
8     }
9     static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10    static void setRadius(int newRadius) { radius = newRadius; }
11    public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12    public static int getCircumference1() { return getDiameter() * PI; }
13    private static int getCircumference2() { return getCircumference(radius); }
14 }
```

When the name of a method parameter clashes with the name of an attribute (**L9**):

- Any mention about that name (e.g., `radius`) refers to the parameter, not the attribute anymore.
- To refer to the attribute, write: `Utilities.radius`
- If you know what you're doing, that's fine; otherwise, use a different name (e.g., **L10**) to avoid unintended errors.

Structure of Utility Classes: Example (1.5)

```
1 public class CircleUtilities {
2     private static final int RADIUS_TO_DIAMETER = 2;
3     static int radius = 10;
4     public static final int PI = 3;
5     static int getDiameter() {
6         int diameter = radius * RADIUS_TO_DIAMETER;
7         return diameter;
8     }
9     static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10    static void setRadius(int newRadius) { radius = newRadius; }
11    public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12    public static int getCircumference1() { return getDiameter() * PI; }
13    private static int getCircumference2() { return getCircumference(radius); }
14 }
```

The body (i.e., what's written between { and }) of a method (accessor or mutator) may:

1. Declare local variables (e.g., **L6**) to store intermediate computation results.
The scope of these local variables is only within that method.
2. Perform assignments to change values of either local variables (**L6**) or attributes (**L10**).

Structure of Utility Classes: Example (1.6)

```
1 public class CircleUtilities {
2     private static final int RADIUS_TO_DIAMETER = 2;
3     static int radius = 10;
4     public static final int PI = 3;
5     static int getDiameter() {
6         int diameter = radius * RADIUS_TO_DIAMETER;
7         return diameter;
8     }
9     static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10    static void setRadius(int newRadius) { radius = newRadius; }
11    public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12    public static int getCircumference1() { return getDiameter() * PI; }
13    private static int getCircumference2() { return getCircumference(radius); }
14 }
```

A method body may **call** another method (i.e., **reuse** code):

3. Call a utility accessor and use (e.g., store, print, return) its return value: **L11** and **L13**.

- **L11:** Since we are in the same class, we do not need to write `CircleUtilities.getDiameter(radius)`
- **L11:** `getDiameter(radius)` passes method *parameter* `radius` as an *argument* value to method `getDiameter(...)`
- **L11:** It is equivalent to write (without reusing any code):
`return radius * RADIUS_TO_DIAMETER * PI`

Structure of Utility Classes: Exercise

```
1 public class CircleUtilities {
2     private static final int RADIUS_TO_DIAMETER = 2;
3     static int radius = 10;
4     public static final int PI = 3;
5     static int getDiameter() {
6         int diameter = radius * RADIUS_TO_DIAMETER;
7         return diameter;
8     }
9     static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10    static void setRadius(int newRadius) { radius = newRadius; }
11    public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12    public static int getCircumference1() { return getDiameter() * PI; }
13    private static int getCircumference2() { return getCircumference(radius); }
14 }
```

Is the body of method `getCircumference1` equivalent to the body of method `getCircumference2`? Why or why not?

Structure of Utility Classes: Example (1.7)

```
1 public class CircleUtilities {
2     private static final int RADIUS_TO_DIAMETER = 2;
3     static int radius = 10;
4     public static final int PI = 3;
5     static int getDiameter() {
6         int diameter = radius * RADIUS_TO_DIAMETER;
7         return diameter;
8     }
9     static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10    static void setRadius(int newRadius) { radius = newRadius; }
11    public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12    public static int getCircumference1() { return getDiameter() * PI; }
13    private static int getCircumference2() { return getCircumference(radius); }
14 }
```

A method body may **call** another method (i.e., **reuse** code):

4. Call a utility mutator to change some data.

We will see an example about this later.

Visualizing a Utility Class

All *static* attributes and methods collectively make their resident utility class a **single** (i.e., one that cannot be duplicated) machine, which contains:

- Current values of attributes
- Definitions of methods (i.e., how computation is to be executed)

CircleUtilities	
RADIUS_TO_DIAMETER	2
radius	10
PI	3
getDiameter()	<code>int diameter = radius * RADIUS_TO_DIAMETER; return diameter;</code>
setRadius(int newRadius)	<code>radius = newRadius;</code>
getCircumference(int radius)	<code>return getDiameter(radius) * PI;</code>
getCircumference1()	<code>return getDiameter() * PI;</code>
getCircumference2()	<code>return getCircumference(radius);</code>

Using a Utility Class (1)

- We can either access a static attribute or call a static method in a utility class using its name.
- e.g., the method call `CircleUtilities.setRadius(40)` passes the value 40 as *argument*, which is used to instantiate every occurrence of the method *parameter* `newRadius` in method `setRadius` by 40.

```
void setRadius(int newRadius 40) {  
    radius = newRadius 40;  
}
```

- Consequently, the effect of this method call is to change the current value of `CircleUtilities.radius` to 40.

Entry Point of Execution: the “main” Method



The *main* method is treated by Java as the **starting point** of executing your program.

```
public class CircleUtilitiesApplication {  
    public static void main(String[] args) {  
        /* Your programming solution is defined here. */  
    }  
}
```

The execution starts with the first line in the *main* method, proceed line by line, from top to bottom, until there are no more lines to execute, then it **terminates**.

Using a Utility Class (2.1)

```
1 public class CircleUtilitesApplication {
2     public static void main(String[] args) {
3         System.out.println("Initial radius of CU: " + CircleUtilities.radius);
4         int d1 = CircleUtilities.getDiameter();
5         System.out.println("d1 is: " + d1);
6         System.out.println("c1 is: " + CircleUtilities.getCircumference1());
7         System.out.println("=====");
8         System.out.println("d2 is: " + CircleUtilities.getDiameter(20));
9         System.out.println("c2 is: " + CircleUtilities.getCircumference(20));
10        System.out.println("=====");
11        System.out.println("Change the radius of CU to 30...");
12        CircleUtilities.setRadius(30);
13        System.out.println("=====");
14        d1 = CircleUtilities.getDiameter();
15        System.out.println("d1 is: " + d1);
16        System.out.println("c1 is: " + CircleUtilities.getCircumference1());
17        System.out.println("=====");
18        System.out.println("d2 is: " + CircleUtilities.getDiameter(20));
19        System.out.println("c2 is: " + CircleUtilities.getCircumference(20));
20    }
21 }
```

Executing it, what will be output to the console?

Using a Utility Class (2.2)

```
Initial radius of CU: 10
d1 is: 20
c1 is: 60
=====
d2 is: 40
c2 is: 120
=====
Change the radius of CU to 30...
=====
d1 is: 60
c1 is: 180
=====
d2 is: 40
c2 is: 120
```

Using a Utility Class: Client vs. Supplier (1)

- A **supplier** implements/provides a service (e.g., microwave).
- A **client** uses a service provided by some supplier.
 - The client must follow certain instructions to obtain the service (e.g., supplier **assumes** that client powers on, closes door, and heats something that is not explosive).
 - If instructions are followed, the client would **expect** that the service does what is required (e.g., a lunch box is heated).
 - The client does not care how the supplier implements it.
- What then are the *benefits* and *obligations* of the two parties?

	<i>benefits</i>	<i>obligations</i>
CLIENT	obtain a service	follow instructions
SUPPLIER	give instructions	provide a service

- There is a **contract** between two parties, violated if:
 - The instructions are not followed. [Client's fault]
 - Instructions followed, but service not satisfactory. [Supplier's fault]

Using a Utility Class: Client vs. Supplier (2)

```
class CUtil {
    static int PI = 3;
    static int getArea(int r) {
        /* Assume: r positive */
        return r * r * 3;
    }
}
```

```
1 class CUtilApp {
2     public static void main(...) {
3         int radius = ???;
4         println(CUtil.getArea(radius));
5     }
```

- Method call CircleUtilities.getArea(radius), inside class CircleUtilitiesApp, suggests a **client-supplier relation**.
 - **Client**: resident class of the static method call [CUtilApp]
 - **Supplier**: context class of the static method [CUtil]
- What if the value of `???` at **L3** of CUtilApp is -10?

300

- What's wrong with this?
 - Client CUtil mistakenly gives illegal circle with radius -10.
 - Supplier CUtil should have reported a **contract violation**!

Using a Utility Class: Client vs. Supplier (3)

- **Method Precondition**: supplier's assumed circumstances, under which the client can expect a satisfactory service.
 - Precondition of `int divide(int x, int y)?` [`y != 0`]
 - Precondition of `int getArea(int r)?` [`r > 0`]
- When **supplier** is requested to provide service with **preconditions** **not** satisfied, **contract is violated** by client.
- **Precondition Violations** \approx `IllegalArgumentException`.
Use `if-elseif` statements to determine if a violation occurs.

```
class CUtil {
    static int PI = 3;
    static int getArea(int r) throws IllegalArgumentException {
        if(r < 0) {
            throw new IllegalArgumentException("Circle radius " + r + "is not positive.");
        }
        else {
            return r * r * PI;
        }
    }
}
```

Documenting Your Class using Javadoc (1)

There are three types of comments in Java:

- `//` [line comment]
- `/* */` [block comment]
 - These two types of comments are only for you as a **supplier** to document interworking of your code.
 - They are hidden from **clients** of your software.
- `/** */` [block documentation]
 - This type of comments is for **clients** to learn about how to use of your software.

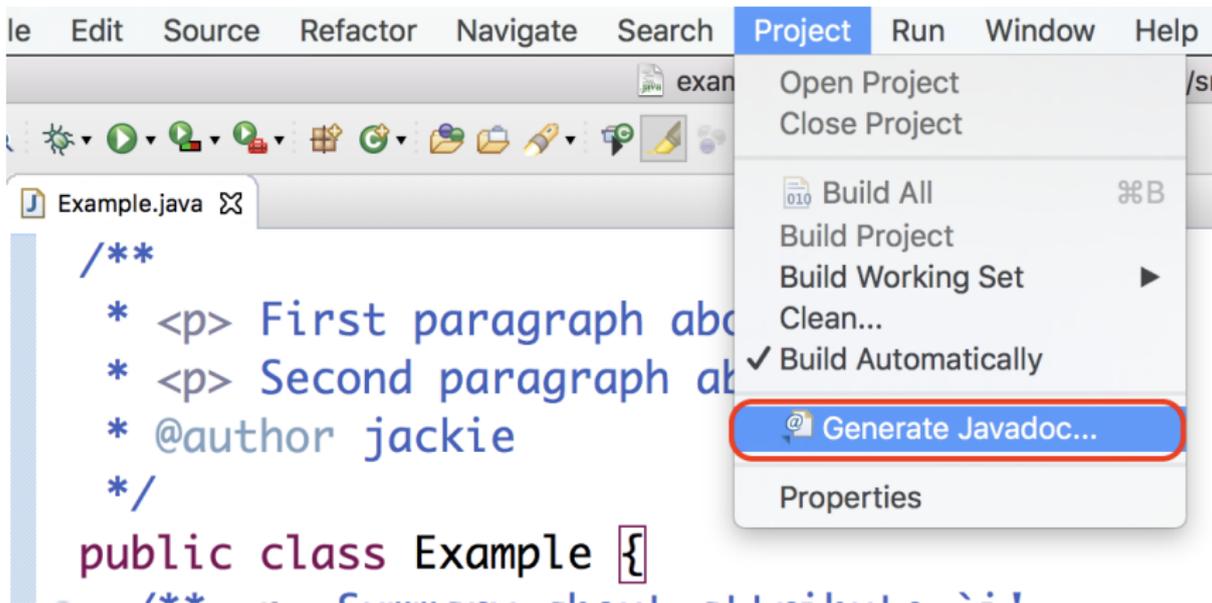
Documenting Classes using Javadoc (2.1)

```
/**
 * <p> First paragraph about this class.
 * <p> Second paragraph about this class.
 * @author jackie
 */
public class Example {
    /** <p> Summary about attribute 'i'
     * <p> More details about 'i'
     */
    public static int i;
    /**
     * <p> Summary about accesor method 'am' with two parameters.
     * <p> More details about 'am'.
     * @return Always false for some reason.
     * @param s Documentation about the first parameter
     * @param d Documentation about the second parameter
     */
    public static boolean am (String s, double d) { return false; }
    /**
     * <p> Summary about mutator method 'mm' with no parameters.
     * <p> More details about 'mm'.
     */
    public static void mm () { /* code omitted */ }
}
```

- o Use **@return** only for mutator methods (i.e., returning non-void).
- o Use **@param** for each input parameter.

Documenting Classes using Javadoc (2.2)

Generate an HTML documentation using the Javadoc tool supported by Eclipse:



The screenshot shows the Eclipse IDE interface. The 'Project' menu is open, and the 'Generate Javadoc...' option is highlighted with a red rectangle. The background shows a Java file named 'Example.java' with the following Javadoc comments:

```
/**  
 * <p> First paragraph about  
 * <p> Second paragraph about  
 * @author jackie  
 */  
public class Example {  
    /**  
     * Comment about the class  
     */  
}
```

Exercises

- Implement a utility class named `Counter`, where
 - There is a static integer counter `i` whose initial value is 5.
 - There is a static constant maximum `MAX` of value 10 for counter `i`.
 - There is a static constant minimum `MIN` of value 10 for counter `i`.
 - Your implementation should be such that the counter value can never fall out of the range `[5, 10]`.
 - There is a mutator method `incrementBy` which takes an integer input parameter `j`, and increments the counter `i` value by `j` if possible (i.e., it would not go above `MAX`).
 - There is a mutator method `decrementBy` which takes an integer input parameter `j`, and decrements the counter `i` value by `j` if possible (i.e., it would not go below `MIN`).
 - There is an accessor method `isPositive` which takes an integer input parameter `j`, and returns `true` if `j` is positive, or returns `false` if otherwise.
- Properly document your `Counter` class using Javadoc and generate the HTML documentation using Eclipse.

Index (1)

Structure of Project: Packages and Classes

Visibility: Project, Packages, Classes

Visibility of Classes

Visibility of Classes: Across All Classes

Within the Resident Package (no modifier)

Visibility of Classes: Across All Classes

Within the Resident Package (no modifier)

Visibility of Attributes/Methods:

Using Modifiers to Define Scopes

Visibility of Attr./Meth.: Across All Methods

Within the Resident Class (`private`)

Visibility of Attr./Meth.: Across All Classes

Within the Resident Package (no modifier)

Visibility of Attr./Meth.: Across All Packages

Within the Resident Project (`public`)

Index (2)

Structure of Utility Classes

Structure of Utility Classes: Example (1.1)

Structure of Utility Classes: Example (1.2)

Structure of Utility Classes: Example (1.3)

Structure of Utility Classes: Example (1.4)

Structure of Utility Classes: Example (1.5)

Structure of Utility Classes: Example (1.6)

Structure of Utility Classes: Exercise

Structure of Utility Classes: Example (1.7)

Visualizing a Utility Class

Using a Utility Class (1)

Entry Point of Execution: the “main” Method

Using a Utility Class (2.1)

Using a Utility Class (2.2)

Index (3)

Using a Utility Class: Client vs. Supplier (1)

Using a Utility Class: Client vs. Supplier (2)

Using a Utility Class: Client vs. Supplier (3)

Documenting Your Class using Javadoc (1)

Documenting Classes using Javadoc (2.1)

Documenting Classes using Javadoc (2.2)

Exercises

Unit and Regression Testing using JUnit



EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

A Simple Counter (1)

Consider a **utility class** (where attributes and methods are **static**) for keeping track of an integer counter value:

```
public class Counter {  
    public final static int MAX_COUNTER_VALUE = 3;  
    public final static int MIN_COUNTER_VALUE = 0;  
    public static int value = MIN_COUNTER_VALUE;  
    ... /* more code later! */  
}
```

- When attempting to access the **static** attribute `value` **outside** the `Counter` class, write `Counter.value`.
- Two constants (i.e., `final`) for lower and upper bounds of the counter value.
- Initialize the counter value to its lower bound.
- **Requirement** :
The counter value must be between its lower and upper bounds.

Encode Precondition Violation as `IllegalArgumentException`

Consider two possible scenarios of *Precondition Violations* (i.e., scenarios of throwing `IllegalArgumentException`):

- When the counter value is attempted (but not yet) to be updated above its upper bound.
- When the counter value is attempted (but not yet) to be updated below its upper bound.

A Simple Counter (2)

```
public static void increment() {
    if( value == Counter.MAX_COUNTER_VALUE ) {
        /* Precondition Violation */
        throw new IllegalArgumentException("Too large to increment");
    }
    else { value ++; }
}

public static void decrement() {
    if( value == Counter.MIN_COUNTER_VALUE ) {
        /* Precondition Violation */
        throw new IllegalArgumentException("Too small to decrement");
    }
    else { value --; }
}
```

- Change the counter value via two mutator methods.
- Changes on the counter value may **violate a precondition**:
 - Attempt to **increment** when counter value reaches its **maximum**.
 - Attempt to **decrement** when counter value reaches its **minimum**.

Testing the Counter Class from Console: Test Case 1

Consider a class for testing the Counter class:

```
public class CounterTester1 {  
    public static void main(String[] args) {  
        System.out.println("Init val: " + Counter.value);  
        System.out.println("Attempt to decrement:");  
        /* Right before calling the decrement mutator,  
         * Counter.value is 0 and too small to be decremented.  
         */  
        Counter.decrement();  
    }  
}
```

Executing it as Java Application gives this Console Output:

```
Init val: 0  
Attempt to decrement:  
Exception in thread "main"  
    java.lang.IllegalArgumentException: Too small to decrement
```

Testing the Counter Class from Console: Test Case 2

Consider **another** class for testing the Counter class:

```
public class CounterTester2 {  
    public static void main(String[] args) {  
        Counter.increment(); Counter.increment(); Counter.increment();  
        System.out.println("Current val: " + Counter.value);  
        System.out.println("Attempt to increment:");  
        /* Right before calling the increment mutator,  
         * Counter.value is 3 and too large to be incremented.  
         */  
        Counter.increment();  
    }  
}
```

Executing it as Java Application gives this Console Output:

```
Current val: 3  
Attempt to increment:  
Exception in thread "main"  
    java.lang.IllegalArgumentException: Too large to increment
```

Limitations of Testing from the Console

- Do **Test Cases 1 & 2** suffice to test `Counter`'s *correctness*?
 - Is it plausible to claim that the implementation of `Counter` is *correct* because it passes the two test cases?
- What other test cases can you think of?

<code>Counter.value</code>	<code>Counter.increment ()</code>	<code>Counter.decrement ()</code>
0	1	<code>ValueTooSmall</code>
1	2	0
2	3	1
3	<code>ValueTooBig</code>	2

- So in total we need 8 test cases.
 - ⇒ 6 more separate `CounterTester` classes to create!
- Problems? It is inconvenient to:
 - Run each TC by executing `main` of a `CounterTester` and comparing console outputs *with your eyes*.
 - Re-run *manually* all TCs whenever `Counter` is changed.
 - Principle:** Any **change** introduced to your software *must not compromise* its established **correctness**.

Why JUnit?

- **Automate** the *testing of correctness* of your Java classes.
- Once you derive the list of tests, translate it into a JUnit test case, which is just a Java class that you can execute upon.
- JUnit tests are *helpful clients* of your classes, where each test may:
 - Either attempt to use a method in a *legal* way (i.e., *satisfying* its precondition), and report:
 - **Success** if the result is as expected
 - **Failure** if the result is *not* as expected
 - Or attempt to use a method in an *illegal* way (i.e., *not satisfying* its precondition), and report:
 - **Success** if precondition violation (i.e., `IllegalArgumentException`) occurs.
 - **Failure** if precondition violation (i.e., `IllegalArgumentException`) does *not* occur.

How to Use JUnit: Packages

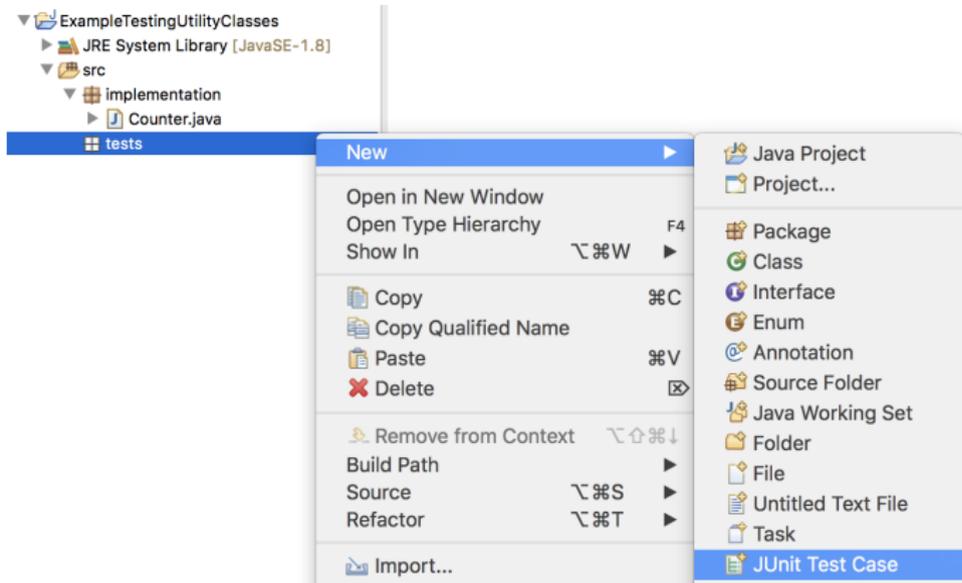
Step 1:

- In Eclipse, create a Java project
ExampleTestingUtilityClasses
- **Separation of concerns** :
 - Group classes for **implementation** (i.e., Counter) into package `implementation`.
 - Group classes classes for **testing** (to be created) into package `tests`.



How to Use JUnit: New JUnit Test Case (1)

Step 2: Create a new *JUnit Test Case* in tests package.

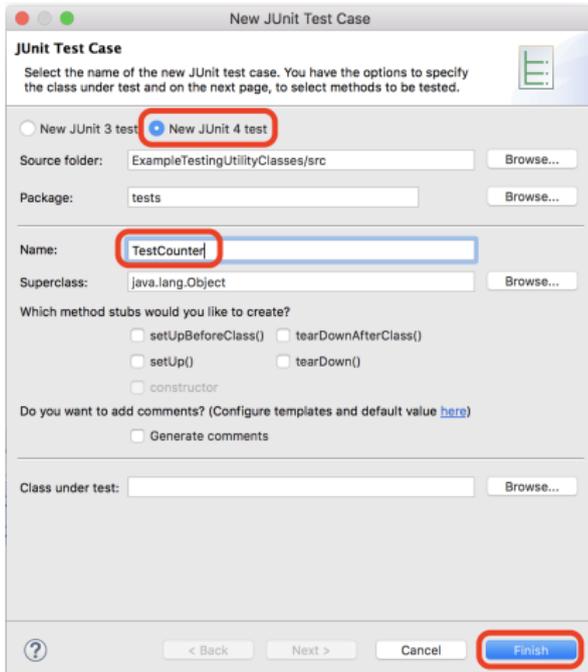


Create one JUnit Test Case to test one Java class only.

⇒ If you have *n Java classes to test*, create *n JUnit test cases*.

How to Use JUnit: New JUnit Test Case (2)

Step 3: Select the version of JUnit (JUnit 4); Enter the name of test case (`TestCounter`); Finish creating the new test case.



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test **New JUnit 4 test**

Source folder: ExampleTestingUtilityClasses/src

Package: tests

Name: **TestCounter**

Superclass: java.lang.Object

Which method stubs would you like to create?

setUpBeforeClass() tearDownAfterClass()
 setUp() tearDown()
 constructor

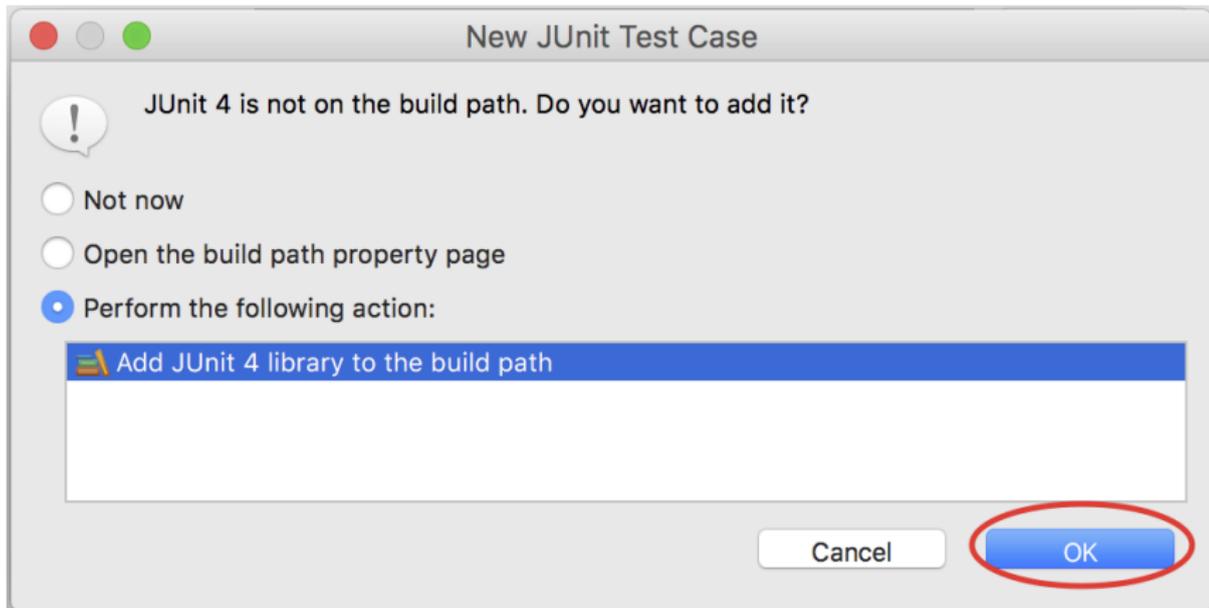
Do you want to add comments? (Configure templates and default value [here](#))

Generate comments

Class under test:

How to Use JUnit: Adding JUnit Library

Upon creating the very first test case, you will be prompted to add the JUnit library to your project's build path.



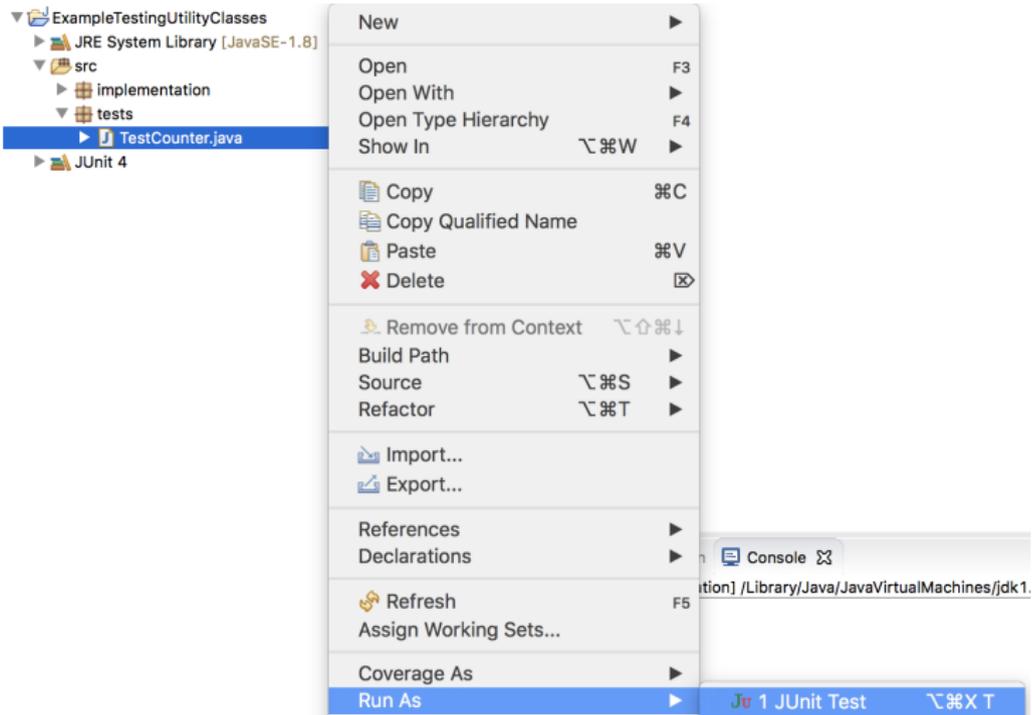
How to Use JUnit: Generated Test Case

```
TestCounter.java ⌵
1 package tests;
2 import static org.junit.Assert.*;
3 import org.junit.Test;
4 public class TestCounter {
5     @Test
6     public void test() {
7         fail("Not yet implemented");
8     }
9 }
```

- **Lines 6 – 8:** `test` is just an *ordinary mutator method* that has a one-line implementation body.
- **Line 5** is critical: Prepend the tag `@Test` verbatim, requiring that *the method is to be treated as a JUnit test*.
⇒ When `TestCounter` is run as a JUnit Test Case, only *those methods prepended by the @Test tags* will be run and reported.
- **Line 7:** By default, we deliberately fail the test with a message “Not yet implemented”.

How to Use JUnit: Running Test Case

Step 4: Run the `TestCounter` class as a JUnit Test.



The screenshot shows an IDE interface. On the left, a project tree is visible with the following structure:

- ExampleTestingUtilityClasses
 - JRE System Library [JavaSE-1.8]
 - src
 - implementation
 - tests
 - TestCounter.java**
 - JUnit 4

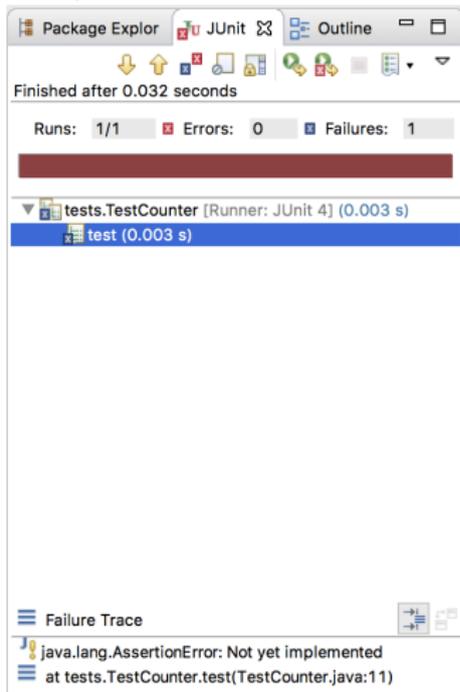
A context menu is open over `TestCounter.java`, listing the following actions:

- New
- Open (F3)
- Open With
- Open Type Hierarchy (F4)
- Show In (⌘ %W)
- Copy (⌘ C)
- Copy Qualified Name
- Paste (⌘ V)
- Delete (⌘ X)
- Remove from Context (⌘ ⬆ ⬇)
- Build Path
- Source (⌘ %S)
- Refactor (⌘ %T)
- Import...
- Export...
- References
- Declarations
- Refresh (F5)
- Assign Working Sets...
- Coverage As
- Run As (⌘ %X T)**

In the bottom right corner, a console window is partially visible, showing the text: `...tion] /Library/Java/JavaVirtualMachines/jdk1.`

How to Use JUnit: Generating Test Report

A **report** is generated after running all tests (i.e., methods prepended with **@Test**) in `TestCounter`.



The screenshot shows the JUnit test runner interface. At the top, it says "Finished after 0.032 seconds". Below that, the summary shows "Runs: 1/1", "Errors: 0", and "Failures: 1". A red progress bar indicates the failure. The test tree shows "tests.TestCounter [Runner: JUnit 4] (0.003 s)" expanded to show "test (0.003 s)". At the bottom, the "Failure Trace" section shows the following error:

```
java.lang.AssertionError: Not yet implemented
    at tests.TestCounter.test(TestCounter.java:11)
```

How to Use JUnit: Interpreting Test Report

- A **test** is a method prepended with the **@Test** tag.
- The result of running a test is considered:
 - **Failure** if either
 - an assertion failure (e.g., caused by `fail`, `assertTrue`, `assertEquals`) occurs; or
 - an **unexpected** exception (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`) is thrown.
 - **Success** if neither assertion failures nor **unexpected** exceptions occur.
- After running all tests:
 - A **green** bar means that **all** tests succeed.
⇒ Keep challenging yourself if **more tests** may be added.
 - A **red** bar means that **at least one** test fails.
⇒ Keep fixing the class under test and re-running all tests, until you receive a **green** bar.
- **Question:** What is the easiest way to making test a **success**?
Answer: Delete the call `fail("Not yet implemented")`.

How to Use JUnit: Revising Test Case

```
TestCounter.java ✕
1 package tests;
2 import static org.junit.Assert.*;
3 import org.junit.Test;
4 public class TestCounter {
5     @Test
6     public void test() {
7         // fail("Not yet implemented");
8     }
9 }
```

Now, the body of `test` simply does nothing.

⇒ Neither assertion failures nor exceptions will occur.

⇒ The execution of `test` will be considered as a **success**.

∴ There is currently only one test in `TestCounter`.

∴ We will receive a **green** bar!

Caution: `test` which passes at the moment is **not useful** at all!

How to Use JUnit: Re-Running Test Case

A new report is generated after re-running all tests (i.e., methods prepended with `@Test`) in `TestCounter`.



How to Use JUnit: Adding More Tests (1)

- Recall the complete list of cases for testing Counter:

<code>c.getValue()</code>	<code>c.increment()</code>	<code>c.decrement()</code>
0	1	ValueTooSmall
1	2	0
2	3	1
3	ValueTooBig	2

- Let's turn the two cases in the 1st row into two JUnit tests:
 - Test for left cell *succeeds* if:
 - No failures and exceptions occur; and
 - The new counter value is 1.
 - Test for right cell *succeeds* if the *expected precondition violation* occurs (`IllegalArgumentException` is thrown).
- Common JUnit assertion methods (complete list in next slide):
 - `void assertNull(Object o)`
 - `void assertEquals(expected, actual)`
 - `void assertTrue(boolean condition)`
 - `void fail(String message)`

How to Use JUnit: Assertion Methods

method name / parameters	description
<code>assertTrue(test)</code> <code>assertTrue("message", test)</code>	Causes this test method to fail if the given boolean test is not <code>true</code> .
<code>assertFalse(test)</code> <code>assertFalse("message", test)</code>	Causes this test method to fail if the given boolean test is not <code>false</code> .
<code>assertEquals(expectedValue, value)</code> <code>assertEquals("message", expectedValue, value)</code>	Causes this test method to fail if the given two values are not equal to each other. (For objects, it uses the <code>equals</code> method to compare them.) The first of the two values is considered to be the result that you expect; the second is the actual result produced by the class under test.
<code>assertNotEquals(value1, value2)</code> <code>assertNotEquals("message", value1, value2)</code>	Causes this test method to fail if the given two values <i>are</i> equal to each other. (For objects, it uses the <code>equals</code> method to compare them.)
<code>assertNull(value)</code> <code>assertNull("message", value)</code>	Causes this test method to fail if the given value is not <code>null</code> .
<code>assertNotNull(value)</code> <code>assertNotNull("message", value)</code>	Causes this test method to fail if the given value <i>is</i> <code>null</code> .
<code>assertSame(expectedValue, value)</code> <code>assertSame("message", expectedValue, value)</code> <code>assertNotSame(value1, value2)</code> <code>assertNotSame("message", value1, value2)</code>	Identical to <code>assertEquals</code> and <code>assertNotEquals</code> respectively, except that for objects, it uses the <code>==</code> operator rather than the <code>equals</code> method to compare them. (The difference is that two objects that have the same state might be <code>equals</code> to each other, but not <code>==</code> to each other. An object is only <code>==</code> to itself.)
<code>fail()</code> <code>fail("message")</code>	Causes this test method to fail.

How to Use JUnit: Adding More Tests (2.1)

```
1 @Test  
2 public void testIncAfterCreation() {  
3     /* Assert that initial value of counter is correct. */  
4     assertEquals(Counter.MIN_COUNTER_VALUE, Counter.value);  
5     /* Attempt to increment the counter value,  
6      * which is expected to succeed.  
7     */  
8     Counter.increment();  
9     /* Assert that the updated counter value is correct. */  
10    assertEquals(1, Counter.value);  
11 }
```

- **L4:** Alternatively, you can write:

```
assertTrue(Counter.MIN_COUNTER_VALUE == Counter.value);
```

- **L10:** Alternatively, you can write:

```
assertTrue(1 == Counter.value);
```

How to Use JUnit: Adding More Tests (2.2)

- Don't lose the big picture!
- The JUnit test in the previous slide automates the following console tester which requires interaction with the external user:

```
public class CounterTester1 {  
    public static void main(String[] args) {  
        System.out.println("Init val: " + Counter.value);  
        System.out.println("Attempt to decrement:");  
        /* Right before calling the decrement mutator,  
         * Counter.value is 0 and too small to be decremented.  
         */  
        Counter.decrement();  
    }  
}
```

- **Automation is exactly rationale behind using JUnit!**

How to Use JUnit: Adding More Tests (3.1)

```
1  @Test
2  public void testDecAfterCreation() {
3      assertTrue(Counter.MIN_COUNTER_VALUE == Counter.value);
4      try {
5          Counter.decrement();
6          /* Reaching this line means
7           * IllegalArgumentException not thrown! */
8          fail("Expected Precondition Violation Did Not Occur!");
9      }
10     catch(IllegalArgumentException e) {
11         /* Precondition Violated Occurred as Expected. */
12     } }
```

- **Lines 4 & 10:** We need a try-catch block because of **Line 5**.
 - Method decrement from class Counter is expected to throw the IllegalArgumentException because of a **precondition violation**.
- **Lines 3 & 8** are both assertions:
 - **Lines 3** asserts that Counter.value returns the expected value (Counter.MIN_COUNTER_VALUE).
 - **Line 8:** an assertion failure
 - ∴ expected IllegalArgumentException not thrown

How to Use JUnit: Adding More Tests (3.2)

- Again, don't lose the big picture!
- The JUnit test in the previous slide automates the following console tester which requires interaction with the external user:

```
public class CounterTester2 {  
    public static void main(String[] args) {  
        Counter.increment(); Counter.increment(); Counter.increment();  
        System.out.println("Current val: " + Counter.value);  
        System.out.println("Attempt to increment:");  
        /* Right before calling the increment mutator,  
         * Counter.value is 3 and too large to be incremented.  
         */  
        Counter.increment();  
    }  
}
```

- Again, **automation is exactly rationale behind using JUnit!**

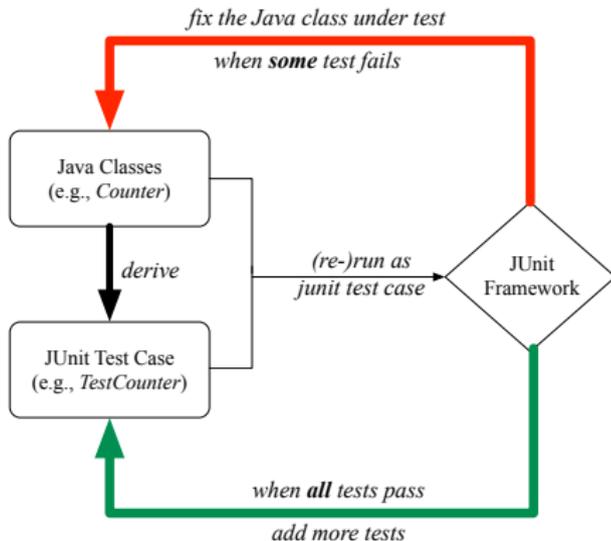
Exercises

1. Convert the rest of the cells into JUnit tests:

<code>c.getValue()</code>	<code>c.increment()</code>	<code>c.decrement()</code>
0	1	ValueTooSmall
1	2	0
2	3	1
3	ValueTooBig	2

2. Run all 8 tests and make sure you receive a *green* bar.
3. Now, introduction an error to the implementation: Change the line `value ++` in `Counter.increment` to `--`.
 - Re-run all 8 tests and you should receive a *red* bar. [Why?]
 - Undo the error injection, and re-run all 8 tests. [What happens?]

Regression Testing



Maintain a collection of tests which define the *correctness* of your Java class under development (CUD):

- Derive and run tests as soon as your CUD is **testable**.
i.e., A Java class is testable when defined with method signatures.
- **Red** bar reported: Fix the class under test (CUT) until **green** bar.
- **Green** bar reported: Add more tests and Fix CUT when necessary.

- Official Site of JUnit 4:

`http://junit.org/junit4/`

- API of JUnit assertions:

`http://junit.sourceforge.net/javadoc/org/junit/Assert.html`

- Another JUnit Tutorial example:

`https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml`

Index (1)

A Simple Counter (1)

Encode Precondition Violation

as `IllegalArgumentException`

A Simple Counter (2)

Testing the Counter Class from Console:

Test Case 1

Testing the Counter Class from Console:

Test Case 2

Limitations of Testing from the Console

Why JUnit?

How to Use JUnit: Packages

How to Use JUnit: New JUnit Test Case (1)

How to Use JUnit: New JUnit Test Case (2)

How to Use JUnit: Adding JUnit Library

How to Use JUnit: Generated Test Case

Index (2)

- How to Use JUnit: Running Test Case
- How to Use JUnit: Generating Test Report
- How to Use JUnit: Interpreting Test Report
- How to Use JUnit: Revising Test Case
- How to Use JUnit: Re-Running Test Case
- How to Use JUnit: Adding More Tests (1)
- How to Use JUnit: Assertion Methods
- How to Use JUnit: Adding More Tests (2.1)
- How to Use JUnit: Adding More Tests (2.2)
- How to Use JUnit: Adding More Tests (3.1)
- How to Use JUnit: Adding More Tests (3.2)
- Exercises
- Regression Testing
- Resources

Classes and Objects

Readings: Chapters 3 – 4 of the Course Notes



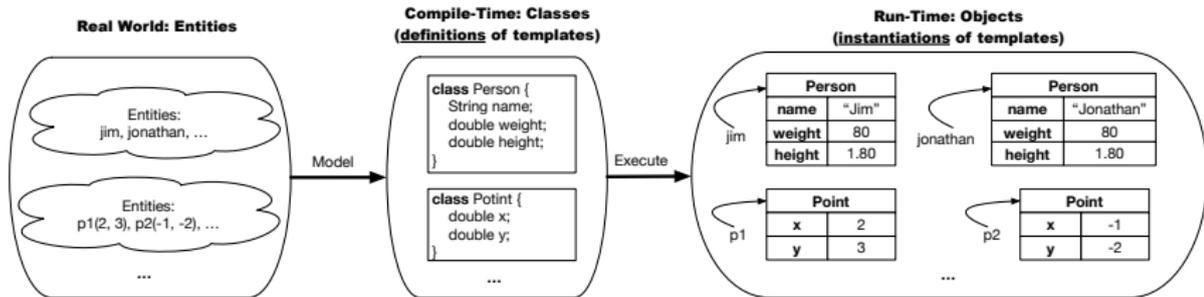
EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Separation of Concerns: App vs. Model

- So far we have developed:
 - Supplier:** A single utility class.
 - Client:** A class with its `main` method using the utility methods.
- In Java:
 - We may define more than one (non-utility) *classes*
 - Each class may contain more than one *methods*
- *object-oriented programming* in Java:
 - Use *classes* to define templates
 - Use *objects* to instantiate classes
 - At *runtime*, *create* objects and *call* methods on objects, to *simulate interactions* between real-life entities.

Object Orientation: Observe, Model, and Execute



- We **observe** how real-world *entities* behave.
- We **model** the common *attributes* and *behaviour* of a set of entities in a single *class*.
- We **execute** the program by creating *instances* of classes, which interact in a way analogous to that of real-world *entities*.

Object-Oriented Programming (OOP)

- In real life, lots of **entities** exist and interact with each other.
 - e.g., *People* gain/lose weight, marry/divorce, or get older.
 - e.g., *Cars* move from one point to another.
 - e.g., *Clients* initiate transactions with banks.
- Entities:
 - Possess *attributes*;
 - Exhibit *behaviour*; and
 - Interact with each other.
- Goals: Solve problems *programmatically* by
 - *Classifying* entities of interest
Entities in the same class share *common* attributes and behaviour.
 - *Manipulating* data that represent these entities
Each entity is represented by *specific* values.

OO Thinking: Templates vs. Instances (1.1)

A person is a being, such as a human, that has certain attributes and behaviour constituting personhood: a person ages and grows on their heights and weights.

- A template called `Person` defines the common
 - *attributes* (e.g., age, weight, height) [≈ nouns]
 - *behaviour* (e.g., get older, gain weight) [≈ verbs]

OO Thinking: Templates vs. Instances (1.2)

- Persons share these common *attributes* and *behaviour*.
 - Each person possesses an age, a weight, and a height.
 - Each person's age, weight, and height might be *distinct*
e.g., `jim` is 50-years old, 1.8-meters tall and 80-kg heavy
e.g., `jonathan` is 65-years old, 1.73-meters tall and 90-kg heavy
- Each person, depending on the *specific values* of their attributes, might exhibit *distinct* behaviour:
 - When `jim` gets older, he becomes 51
 - When `jonathan` gets older, he becomes 66.
 - `jim`'s BMI is based on his own height and weight
 - `jonathan`'s BMI is based on his own height and weight

$$\left[\frac{80}{1.8^2} \right]$$
$$\left[\frac{90}{1.73^2} \right]$$

OO Thinking: Templates vs. Instances (2.1)

Points on a two-dimensional plane are identified by their signed distances from the X- and Y-axes. A point may move arbitrarily towards any direction on the plane. Given two points, we are often interested in knowing the distance between them.

- A template called `Point` defines the common
 - *attributes* (e.g., `x`, `y`) [≈ nouns]
 - *behaviour* (e.g., `move up`, `get distance from`) [≈ verbs]

OO Thinking: Templates vs. Instances (2.2)

- Points share these common *attributes* and *behaviour*.
 - Each point possesses an x-coordinate and a y-coordinate.
 - Each point's location might be *distinct*
e.g., p1 is located at (3, 4)
e.g., p2 is located at (-4, -3)
- Each point, depending on the *specific values* of their attributes (i.e., locations), might exhibit *distinct* behaviour:
 - When p1 moves up for 1 unit, it will end up being at (3, 5)
 - When p2 moves up for 1 unit, it will end up being at (-4, -2)
 - Then, p1's distance from origin: $[\sqrt{3^2 + 5^2}]$
 - Then, p2's distance from origin: $[\sqrt{(-4)^2 + (-2)^2}]$

OO Thinking: Templates vs. Instances (3)

- A **template** defines what's **shared** by a set of related entities.
 - Common **attributes** (age in `Person`, `x` in `Point`)
 - Common **behaviour** (get older for `Person`, move up for `Point`)
- Each template may be **instantiated** into multiple instances.
 - `Person` instances: jim and jonathan
 - `Point` instances: p1 and p2
- Each **instance** may have **specific values** for the attributes.
 - Each `Person` instance has an age:
jim is 50-years old, jonathan is 65-years old
 - Each `Point` instance has a location:
p1 is at (3,4), p2 is at (-3,-4)
- Therefore, instances of the same template may exhibit **distinct behaviour**.
 - Each `Person` instance can get older: jim getting older from 50 to 51; jonathan getting older from 65 to 66.
 - Each `Point` instance can move up: p1 moving up from (3,3) results in (3,4); p2 moving up from (-3,-4) results in (-3,-3).

OOP: Classes \approx Templates

In Java, you use a **class** to define a *template* that enumerates *attributes* that are common to a set of *entities* of interest.

```
public class Person {  
    int age;  
    String nationality;  
    double weight;  
    double height;  
}
```

```
public class Point {  
    double x;  
    double y;  
}
```

OOP:

Define Constructors for Creating Objects (1.1)

- Within class `Point`, you define **constructors**, specifying how instances of the `Point` template may be created.

```
public class Point {  
    ... /* attributes: x, y */  
    Point(double newX, double newY) {  
        x = newX;  
        y = newY; } }  
}
```

- In the corresponding tester class, each **call** to the `Point` constructor creates an instance of the `Point` template.

```
public class PersonTester {  
    public static void main(String[] args) {  
        Point p1 = new Point(2, 4);  
        println(p1.x + " " + p1.y);  
        Point p2 = new Point(-4, -3);  
        println(p2.x + " " + p2.y); } }  
}
```

OOP:

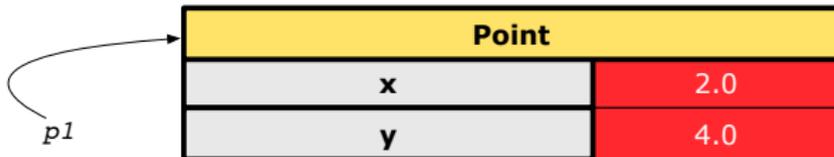
Define Constructors for Creating Objects (1.2)

```
Point p1 = new Point(2, 4);
```

- RHS (Source) of Assignment:** `new Point(2, 4)` creates a new *Point object* in memory.

Point	
x	2.0
y	4.0

- LHS (Target) of Assignment:** `Point p1` declares a *variable* that is meant to store the *address* of *some Point object*.
- Assignment:** Executing `=` stores new object's address in `p1`.



The `this` Reference (1)

- Each *class* may be instantiated to multiple *objects* at runtime.

```
class Point {  
    double x; double y;  
    void moveUp(double units) { y += units; }  
}
```

- Each time when we call a method of some class, using the dot notation, there is a specific *target/context* object.

```
1 Point p1 = new Point(2, 3);  
2 Point p2 = new Point(4, 6);  
3 p1.moveUp(3.5);  
4 p2.moveUp(4.7);
```

- `p1` and `p2` are called the *call targets* or *context objects*.
- **Lines 3 and 4** apply the same definition of the `moveUp` method.
- But how does Java distinguish the change to `p1.y` versus the change to `p2.y`?

The `this` Reference (2)

- In the *method* definition, each *attribute* has an *implicit* `this` which refers to the **context object** in a call to that method.

```
class Point {  
    double x;  
    double y;  
    Point(double newX, double newY) {  
        this.x = newX;  
        this.y = newY;  
    }  
    void moveUp(double units) {  
        this.y = this.y + units;  
    }  
}
```

- Each time when the *class* definition is used to create a new `Point` *object*, the `this` reference is substituted by the name of the new object.

The `this` Reference (3)

- After we create `p1` as an instance of `Point`

```
Point p1 = new Point(2, 3);
```

- When invoking `p1.moveUp(3.5)`, a version of `moveUp` that is specific to `p1` will be used:

```
class Point {  
    double x;  
    double y;  
    Point(double newX, double newY) {  
        p1.x = newX;  
        p1.y = newY;  
    }  
    void moveUp(double units) {  
        p1.y = p1.y + units;  
    }  
}
```

The this Reference (4)

- After we create `p2` as an instance of `Point`

```
Point p2 = new Point(4, 6);
```

- When invoking `p2.moveUp(4.7)`, a version of `moveUp` that is specific to `p2` will be used:

```
class Point {  
    double x;  
    double y;  
    Point(double newX, double newY) {  
        p2.x = newX;  
        p2.y = newY;  
    }  
    void moveUp(double units) {  
        p2.y = p2.y + units;  
    }  
}
```

The `this` Reference (5)

The `this` reference can be used to **disambiguate** when the names of *input parameters* clash with the names of *class attributes*.

```
class Point {  
    double x;  
    double y;  
    Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    void setX(double x) {  
        this.x = x;  
    }  
    void setY(double y) {  
        this.y = y;  
    }  
}
```

The `this` Reference (6.1): Common Error

The following code fragment compiles but is problematic:

```
class Person {  
    String name;  
    int age;  
    Person(String name, int age) {  
        name = name;  
        age = age;  
    }  
    void setAge(int age) {  
        age = age;  
    }  
}
```

Why? Fix?

The `this` Reference (6.2): Common Error

Always remember to use `this` when *input parameter* names clash with *class attribute* names.

```
class Person {  
    String name;  
    int age;  
    Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    void setAge(int age) {  
        this.age = age;  
    }  
}
```

OOP:

Define Constructors for Creating Objects (2.1)

- Within class `Person`, you define **constructors**, specifying how instances of the `Person` template may be created.

```
public class Person {  
    ... /* attributes: age, nationality, weight, height */  
    Person(int newAge, String newNationality) {  
        age = newAge;  
        nationality = newNationality; } }  
}
```

- In the corresponding tester class, each **call** to the `Person` constructor creates an instance of the `Person` template.

```
public class PersonTester {  
    public static void main(String[] args) {  
        Person jim = new Person(50, "British");  
        println(jim.nationlaity + " " + jim.age);  
        Person jonathan = new Person(60, "Canadian");  
        println(jonathan.nationlaity + " " + jonathan.age); } }  
}
```

OOP:

Define Constructors for Creating Objects (2.2)

```
Person jim = new Person(50, "British");
```

- RHS (Source) of Assignment:** `new Person(50, "British")` creates a new *Person object* in memory.

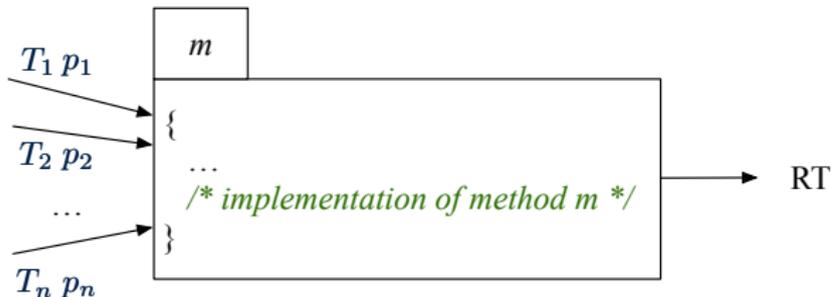
Person	
age	50
nationality	"British"
weight	0.0
height	0.0

- LHS (Target) of Assignment:** `Point jim` declares a *variable* that is meant to store the *address* of *some Person object*.
- Assignment:** Executing `=` stores new object's address in `jim`.



OOP: Methods (1.1)

- A **method** is a named block of code, *reusable* via its name.



- The **Signature** of a method consists of:
 - Return type [*RT* (which can be void)]
 - Name of method [*m*]
 - Zero or more *parameter names* [*p₁, p₂, ..., p_n*]
 - The corresponding *parameter types* [*T₁, T₂, ..., T_n*]
- A call to method *m* has the form: $m(a_1, a_2, \dots, a_n)$
 Types of **argument values** a_1, a_2, \dots, a_n must match the the corresponding parameter types T_1, T_2, \dots, T_n .

OOP: Methods (1.2)

- In the body of the method, you may
 - Declare and use new *local variables*
 - **Scope** of local variables is only within that method.
 - Use or change values of *attributes*.
 - Use values of *parameters*, if any.

```
class Person {  
    String nationality;  
    void changeNationality(String newNationality) {  
        nationality = newNationality; } }  
}
```

- **Call** a *method*, with a **context object**, by passing *arguments*.

```
class PersonTester {  
    public static void main(String[] args) {  
        Person jim = new Person(50, "British");  
        Person jonathan = new Person(60, "Canadian");  
        jim.changeNationality("Korean");  
        jonathan.changeNationality("Korean"); } }  
}
```

OOP: Methods (2)

- Each **class** C defines a list of methods.
 - A **method** m is a named block of code.
- We *reuse* the code of method m by calling it on an **object** obj of class C .
 - For each **method call** $obj.m(\dots)$:
 - obj is the *context object* of type C
 - m is a method defined in class C
 - We intend to apply the *code effect of method* m to object obj .
e.g., `jim.getOlder()` vs. `jonathan.getOlder()`
e.g., `p1.moveUp(3)` vs. `p2.moveUp(3)`
- All objects of class C share *the same definition* of method m .
- However:
 - ∴ Each object may have *distinct attribute values*.
 - ∴ Applying *the same definition* of method m has *distinct effects*.

OOP: Methods (3)

1. *Constructor*

- Same name as the class. No return type. *Initializes* attributes.
- Called with the **new** keyword.
- e.g., `Person jim = new Person(50, "British");`

2. *Mutator*

- *Changes* (re-assigns) attributes
- void return type
- Cannot be used when a value is expected
- e.g., `double h = jim.setHeight(78.5)` is illegal!

3. *Accessor*

- *Uses* attributes for computations (without changing their values)
- Any return type other than void
- An explicit *return statement* (typically at the end of the method) returns the computation result to where the method is being used.
e.g., `double bmi = jim.getBMI();`
e.g., `println(pl.getDistanceFromOrigin());`

OOP: The Dot Notation (1)

- A binary operator:
 - **LHS** an object
 - **RHS** an attribute or a method
- Given a *variable* of some *reference type* that is **not** null:
 - We use a dot to retrieve any of its **attributes**.
Analogous to 's in English
e.g., `jim.nationality` means jim's nationality
 - We use a dot to invoke any of its **mutator methods**, in order to *change* values of its attributes.
e.g., `jim.changeNationality("CAN")` changes the `nationality` attribute of `jim`
 - We use a dot to invoke any of its **accessor methods**, in order to *use* the result of some computation on its attribute values.
e.g., `jim.getBMI()` computes and returns the BMI calculated based on jim's weight and height
 - Return value of an *accessor method* must be stored in a variable.
e.g., `double jimBMI = jim.getBMI()`

OOP: The Dot Notation (2)

- LHS of dot *can be more complicated than a variable*:
 - It can be a *path* that brings you to an object

```
class Person {  
    String name;  
    Person spouse;  
}
```

- Say we have `Person jim = new Person("Jim Davies")`
- Inquire about jim's name? `[jim.name]`
- Inquire about jim's spouse's name? `[jim.spouse.name]`
- But what if jim is single (i.e., `jim.spouse == null`)?
Calling `jim.spouse.name` will trigger *NullPointerException*!!
- Assuming that:
 - jim is not single. `[jim.spouse != null]`
 - The marriage is mutual. `[jim.spouse.spouse != null]`
- What does `jim.spouse.spouse.name` mean? `[jim.name]`

OOP: Method Calls

```
1 Point p1 = new Point (3, 4);
2 Point p2 = new Point (-6, -8);
3 System.out.println(p1. getDistanceFromOrigin() );
4 System.out.println(p2. getDistanceFromOrigin() );
5 p1. moveUp(2) ;
6 p2. moveUp(2) ;
7 System.out.println(p1. getDistanceFromOrigin() );
8 System.out.println(p2. getDistanceFromOrigin() );
```

- **Lines 1 and 2** create two different instances of `Point`
- **Lines 3 and 4:** invoking the same accessor method on two different instances returns *distinct* values
- **Lines 5 and 6:** invoking the same mutator method on two different instances results in *independent* changes
- **Lines 3 and 7:** invoking the same accessor method on the same instance *may* return *distinct* values, why?

Line 5

OOP: Class Constructors (1)

- The purpose of defining a *class* is to be able to create *instances* out of it.
- To *instantiate* a class, we use one of its **constructors**.
- A constructor
 - declares input *parameters*
 - uses input parameters to *initialize* **some or all** of its *attributes*

OOP: Class Constructors (2)

```
public class Person {
    int age;
    String nationality;
    double weight;
    double height;
    Person(int initAge, String initNat) {
        age = initAge;
        nationality = initNat;
    }
    Person (double initW, double initH) {
        weight = initW;
        height = initH;
    }
    Person(int initAge, String initNat,
           double initW, double initH) {
        ... /* initialize all attributes using the parameters */
    }
}
```

OOP: Class Constructors (3)

```
public class Point {  
    double x;  
    double y;  
  
    Point(double initX, double initY) {  
        x = initX;  
        y = initY;  
    }  
  
    Point(char axis, double distance) {  
        if (axis == 'x') { x = distance; }  
        else if (axis == 'y') { y = distance; }  
        else { System.out.println("Error: invalid axis.") }  
    }  
}
```

OOP: Class Constructors (4)

- For each *class*, you may define *one or more* **constructors** :
 - *Names* of all constructors must match the class name.
 - *No return types* need to be specified for constructors.
 - Each constructor must have a *distinct* list of *input parameter types*.
 - Each *parameter* that is used to initialize an attribute must have a *matching type*.
 - The *body* of each constructor specifies how *some or all* *attributes* may be *initialized*.

OOP: Object Creation (1)

```
Point p1 = new Point(2, 4);  
System.out.println(p1);
```

```
Point@677327b6
```

By default, the address stored in `p1` gets printed.
Instead, print out attributes separately:

```
System.out.println("(" + p1.x + ", " + p1.y + ")");
```

```
(2.0, 4.0)
```

OOP: Object Creation (2)

A constructor may only *initialize* some attributes and leave others *uninitialized*.

```
public class PersonTester {
    public static void main(String[] args) {
        /* initialize age and nationality only */
        Person jim = new Person(50, "BRI");
        /* initialize age and nationality only */
        Person jonathan = new Person(65, "CAN");
        /* initialize weight and height only */
        Person alan = new Person(75, 1.80);
        /* initialize all attributes of a person */
        Person mark = new Person(40, "CAN", 69, 1.78);
    }
}
```

OOP: Object Creation (3)

`Person jim = new Person(50, "BRI")`



`Person jonathan = new Person(65, "CAN")`



`Person alan = new Person(75, 1.80)`



`Person mark = new Person(40, "CAN", 69, 1.78)`



OOP: Object Creation (4)

A constructor may only *initialize* some attributes and leave others *uninitialized*.

```
public class PointTester {  
    public static void main(String[] args) {  
        Point p1 = new Point(3, 4);  
        Point p2 = new Point(-3 -2);  
        Point p3 = new Point('x', 5);  
        Point p4 = new Point('y', -7);  
    }  
}
```

OOP: Object Creation (5)

Point p1 = new Point(3, 4)



Point p3 = new Point('x', 5)



Point p2 = new Point(-3, -2)



Point p4 = new Point('y', -7)



OOP: Object Creation (6)

- When using the constructor, pass **valid** *argument values*:
 - The type of each argument value must match the corresponding parameter type.
 - e.g., `Person(50, "BRI")` matches
`Person(int initAge, String initNationality)`
 - e.g., `Point(3, 4)` matches
`Point(double initX, double initY)`
- When creating an instance, *uninitialized* attributes implicitly get assigned the **default values**.
 - Set *uninitialized* attributes properly later using **mutator** methods

```
Person jim = new Person(50, "British");  
jim.setWeight(85);  
jim.setHeight(1.81);
```

OOP: Mutator Methods

- These methods *change* values of attributes.
- We call such methods **mutators** (with `void` return type).

```
public class Person {  
    ...  
    void gainWeight(double units) {  
        weight = weight + units;  
    }  
}
```

```
public class Point {  
    ...  
    void moveUp() {  
        y = y + 1;  
    }  
}
```

OOP: Accessor Methods

- These methods *return* the result of computation based on attribute values.
- We call such methods **accessors** (with non-void return type).

```
public class Person {  
    ...  
    double getBMI() {  
        double bmi = height / (weight * weight);  
        return bmi;  
    }  
}
```

```
public class Point {  
    ...  
    double getDistanceFromOrigin() {  
        double dist = Math.sqrt(x*x + y*y);  
        return dist;  
    }  
}
```

OOP: Use of Mutator vs. Accessor Methods

- Calls to **mutator methods** *cannot* be used as values.
 - e.g., `System.out.println(jim.setWeight(78.5));` ×
 - e.g., `double w = jim.setWeight(78.5);` ×
 - e.g., `jim.setWeight(78.5);` ✓
- Calls to **accessor methods** *should* be used as values.
 - e.g., `jim.getBMI();` ×
 - e.g., `System.out.println(jim.getBMI());` ✓
 - e.g., `double w = jim.getBMI();` ✓

OOP: Method Parameters

- **Principle 1:** A **constructor** needs an *input parameter* for every attribute that you wish to initialize.

e.g., `Person(double w, double h)` vs.
`Person(String fName, String lName)`

- **Principle 2:** A **mutator** method needs an *input parameter* for every attribute that you wish to modify.

e.g., `In Point, void moveToXAxis()` vs.
`void moveUpBy(double unit)`

- **Principle 3:** An **accessor method** needs *input parameters* if the attributes alone are not sufficient for the intended computation to complete.

e.g., `In Point, double getDistFromOrigin()` vs.
`double getDistFrom(Point other)`

The `this` Reference (7.1): Exercise

Consider the `Person` class

```
class Person {  
    String name;  
    Person spouse;  
    Person(String name) {  
        this.name = name;  
    }  
}
```

How do you implement a mutator method `marry` which marries the current `Person` object to an input `Person` object?

The `this` Reference (7.2): Exercise

```
void marry(Person other) {  
    if(this.spouse != null || other.spouse != null) {  
        System.out.println("Error: both must be single.");  
    }  
    else { this.spouse = other; other.spouse = this; }  
}
```

When we call `jim.marry(elsa)`: `this` is substituted by the call target `jim`, and `other` is substituted by the argument `elsa`.

```
void marry(Person other) {  
    ...  
    jim.spouse = elsa;  
    elsa.spouse = jim;  
}  
}
```

Java Data Types (1)

A (data) type denotes a set of related *runtime values*.

1. Primitive Types

- *Integer* Type
 - `int` [set of 32-bit integers]
 - `long` [set of 64-bit integers]
- *Floating-Point Number* Type
 - `double` [set of 64-bit FP numbers]
- *Character* Type
 - `char` [set of single characters]
- *Boolean* Type
 - `boolean` [set of `true` and `false`]

2. Reference Type : *Complex Type with Attributes and Methods*

- *String* [set of references to character sequences]
- *Person* [set of references to Person objects]
- *Point* [set of references to Point objects]
- *Scanner* [set of references to Scanner objects]

Java Data Types (2)

- A variable that is declared with a *type* but *uninitialized* is implicitly assigned with its **default value**.
 - **Primitive Type**
 - `int i;` [`0` is implicitly assigned to `i`]
 - `double d;` [`0.0` is implicitly assigned to `d`]
 - `boolean b;` [`false` is implicitly assigned to `b`]
 - **Reference Type**
 - `String s;` [`null` is implicitly assigned to `s`]
 - `Person jim;` [`null` is implicitly assigned to `jim`]
 - `Point p1;` [`null` is implicitly assigned to `p1`]
 - `Scanner input;` [`null` is implicitly assigned to `input`]
- You *can* use a primitive variable that is *uninitialized*.
Make sure the **default value** is what you want!
- Calling a method on a *uninitialized* reference variable crashes your program. [`NullPointerException`]
Always initialize reference variables!

Java Data Types (3.1)

- An attribute may be of type `ArrayList`, storing references to other objects.

```
class Person { Person spouse; }
```

- Methods may take as `parameters` references to other objects.

```
class Person {  
    void marry(Person other) { ... } }
```

- `Return values` from methods may be references to other objects.

```
class Point {  
    void moveUpBy(int i) { y = y + i; }  
    Point movedUpBy(int i) {  
        Point np = new Point(x, y);  
        np.moveUp(i);  
        return np;  
    }  
}
```

Java Data Types (3.2.1)

An attribute may be of type `ArrayList<Point>`, storing references to `Point` objects.

```
1 class PointCollector {
2     ArrayList<Point> points;
3     PointCollector() { points = new ArrayList<>(); }
4     void addPoint(Point p) {
5         points.add(p); }
6     void addPoint(double x, double y) {
7         points.add(new Point(x, y)); }
8     ArrayList<Point> getPointsInQuadrantI() {
9         ArrayList<Point> qlPoints = new ArrayList<>();
10        for(int i = 0; i < points.size(); i++) {
11            Point p = points.get(i);
12            if(p.x > 0 && p.y > 0) { qlPoints.add(p); } }
13        return qlPoints;
14    } }
```

L8 & L9 may be replaced by:

```
for(Point p : points) { qlPoints.add(p); }
```

Java Data Types (3.2.2)

```
1 class PointCollectorTester {
2     public static void main(String[] args) {
3         PointCollector pc = new PointCollector();
4         System.out.println(pc.points.size()); /* 0 */
5         pc.addPoint(3, 4);
6         System.out.println(pc.points.size()); /* 1 */
7         pc.addPoint(-3, 4);
8         System.out.println(pc.points.size()); /* 2 */
9         pc.addPoint(-3, -4);
10        System.out.println(pc.points.size()); /* 3 */
11        pc.addPoint(3, -4);
12        System.out.println(pc.points.size()); /* 4 */
13        ArrayList<Point> ps = pc.getPointsInQuadrantI();
14        System.out.println(ps.length); /* 1 */
15        System.out.println("(" + ps[0].x + ", " + ps[0].y + ")");
16        /* (3, 4) */
17    }
18 }
```

Java Data Types (3.3.1)

An attribute may be of type `Point[]`, storing references to `Point` objects.

```
1 class PointCollector {
2     Point[] points; int nop; /* number of points */
3     PointCollector() { points = new Point[100]; }
4     void addPoint(double x, double y) {
5         points[nop] = new Point(x, y); nop++; }
6     Point[] getPointsInQuadrantI() {
7         Point[] ps = new Point[nop];
8         int count = 0; /* number of points in Quadrant I */
9         for(int i = 0; i < nop; i ++ ) {
10            Point p = points[i];
11            if(p.x > 0 && p.y > 0) { ps[count] = p; count ++; } }
12        Point[] qlPoints = new Point[count];
13        /* ps contains null if count < nop */
14        for(int i = 0; i < count; i ++ ) { qlPoints[i] = ps[i] }
15        return qlPoints;
16    } }
```

Required Reading: Point and PointCollector

Java Data Types (3.3.2)

```
1 class PointCollectorTester {
2     public static void main(String[] args) {
3         PointCollector pc = new PointCollector();
4         System.out.println(pc.nop); /* 0 */
5         pc.addPoint(3, 4);
6         System.out.println(pc.nop); /* 1 */
7         pc.addPoint(-3, 4);
8         System.out.println(pc.nop); /* 2 */
9         pc.addPoint(-3, -4);
10        System.out.println(pc.nop); /* 3 */
11        pc.addPoint(3, -4);
12        System.out.println(pc.nop); /* 4 */
13        Point[] ps = pc.getPointsInQuadrantI();
14        System.out.println(ps.length); /* 1 */
15        System.out.println("(" + ps[0].x + ", " + ps[0].y + ")");
16        /* (3, 4) */
17    }
18 }
```

OOP: Object Alias (1)

```
1 int i = 3;
2 int j = i; System.out.println(i == j); /* true */
3 int k = 3; System.out.println(k == i && k == j); /* true */
```

- **Line 2** copies the number stored in `i` to `j`.
- After **Line 4**, `i`, `j`, `k` refer to three separate integer placeholder, which happen to store the same value 3.

```
1 Point p1 = new Point(2, 3);
2 Point p2 = p1; System.out.println(p1 == p2); /* true */
3 Point p3 = new Point(2, 3);
4 System.out.println(p3 == p1 || p3 == p2); /* false */
5 System.out.println(p3.x == p1.x && p3.y == p1.y); /* true */
6 System.out.println(p3.x == p2.x && p3.y == p2.y); /* true */
```

- **Line 2** copies the *address* stored in `p1` to `p2`.
- Both `p1` and `p2` refer to the same object in memory!
- `p3`, whose *contents* are same as `p1` and `p2`, refer to a different object in memory.

OO Program Programming: Object Alias (2.1)

Problem: Consider assignments to *primitive* variables:

```
1  int i1 = 1;
2  int i2 = 2;
3  int i3 = 3;
4  int[] numbers1 = {i1, i2, i3};
5  int[] numbers2 = new int[numbers1.length];
6  for(int i = 0; i < numbers1.length; i++) {
7      numbers2[i] = numbers1[i];
8  }
9  numbers1[0] = 4;
10 System.out.println(numbers1[0]);
11 System.out.println(numbers2[0]);
```

OO Program Programming: Object Alias (2.2)

Problem: Consider assignments to **reference** variables:

```
1 Person alan = new Person("Alan");
2 Person mark = new Person("Mark");
3 Person tom = new Person("Tom");
4 Person jim = new Person("Jim");
5 Person[] persons1 = {alan, mark, tom};
6 Person[] persons2 = new Person[persons1.length];
7 for(int i = 0; i < persons1.length; i ++) {
8     persons2[i] = persons1[i]; }
9 persons1[0].setAge(70);
10 System.out.println(jim.age);
11 System.out.println(alan.age);
12 System.out.println(persons2[0].age);
13 persons1[0] = jim;
14 persons1[0].setAge(75);
15 System.out.println(jim.age);
16 System.out.println(alan.age);
17 System.out.println(persons2[0].age);
```

Call by Value vs. Call by Reference (1)

- Consider the general form of a call to some *mutator method* m , with *context object* co and **argument value** arg :

```
co.m (arg)
```

- Argument variable arg is *not* passed directly for the method call.
- Instead, argument variable arg is passed *indirectly*: a **copy** of the value stored in arg is made and passed for the method call.
- What can be the type of variable arg ? [Primitive or Reference]
 - arg is primitive type (e.g., `int`, `char`, `boolean`, etc.):
Call by Value: Copy of arg 's **stored value** (e.g., `2`, `'j'`, `true`) is made and passed.
 - arg is reference type (e.g., `String`, `Point`, `Person`, etc.):
Call by Reference: Copy of arg 's **stored reference/address** (e.g., `Point@5cb0d902`) is made and passed.

Call by Value vs. Call by Reference (2.1)

For illustration, let's assume the following variant of the `Point` class:

```
class Point {
    int x;
    int y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    void moveVertically(int y) {
        this.y += y;
    }
    void moveHorizontally(int x) {
        this.x += x;
    }
}
```

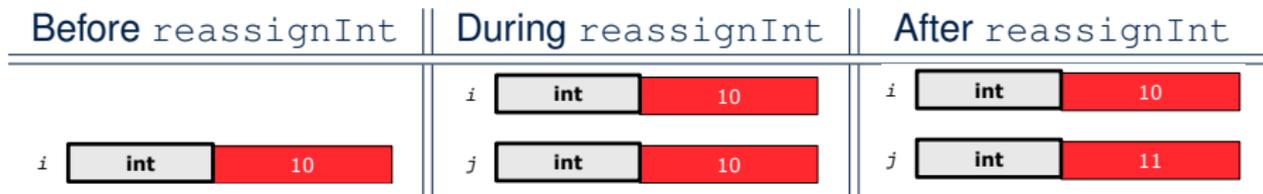
Call by Value vs. Call by Reference (2.2.1)

```
public class Util {  
    void reassignInt(int j) {  
        j = j + 1; }  
    void reassignRef(Point q) {  
        Point np = new Point(6, 8);  
        q = np; }  
    void changeViaRef(Point q) {  
        q.moveHorizontally(3);  
        q.moveVertically(4); } }  
}
```

```
1 @Test  
2 public void testCallByVal() {  
3     Util u = new Util();  
4     int i = 10;  
5     assertTrue(i == 10);  
6     u.reassignInt(i);  
7     assertTrue(i == 10);  
8 }
```

- **Before** the mutator call at **L6**, **primitive** variable `i` stores 10.
- **When** executing the mutator call at **L6**, due to **call by value**, a copy of variable `i` is made.
 - ⇒ The assignment `i = i + 1` is only effective on this copy, not the original variable `i` itself.
- ∴ **After** the mutator call at **L6**, variable `i` still stores 10.

Call by Value vs. Call by Reference (2.2.2)



Call by Value vs. Call by Reference (2.3.1)

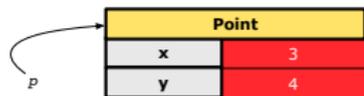
```
public class Util {  
    void reassignInt(int j) {  
        j = j + 1; }  
    void reassignRef(Point q) {  
        Point np = new Point(6, 8);  
        q = np; }  
    void changeViaRef(Point q) {  
        q.moveHorizontally(3);  
        q.moveVertically(4); } }  
}
```

```
1 @Test  
2 public void testCallByRef_1() {  
3     Util u = new Util();  
4     Point p = new Point(3, 4);  
5     Point refOfPBefore = p;  
6     u.reassignRef(p);  
7     assertTrue(p==refOfPBefore);  
8     assertTrue(p.x==3 && p.y==4);  
9 }
```

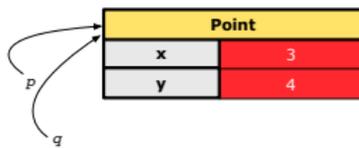
- **Before** the mutator call at **L6**, **reference** variable `p` stores the **address** of some `Point` object (whose `x` is 3 and `y` is 4).
- **When** executing the mutator call at **L6**, due to **call by reference**, a **copy of address** stored in `p` is made.
 - ⇒ The assignment `p = np` is only effective on this copy, not the original variable `p` itself.
- ∴ **After** the mutator call at **L6**, variable `p` still stores the original address (i.e., same as `refOfPBefore`).

Call by Value vs. Call by Reference (2.3.2)

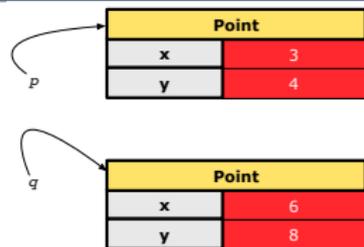
Before `reassignRef`



During `reassignRef`



After `reassignRef`



Call by Value vs. Call by Reference (2.4.1)

```
public class Util {  
    void reassignInt(int j) {  
        j = j + 1; }  
    void reassignRef(Point q) {  
        Point np = new Point(6, 8);  
        q = np; }  
    void changeViaRef(Point q) {  
        q.moveHorizontally(3);  
        q.moveVertically(4); } }  
}
```

```
1 @Test  
2 public void testCallByRef_2() {  
3     Util u = new Util();  
4     Point p = new Point(3, 4);  
5     Point refOfPBefore = p;  
6     u.changeViaRef(p);  
7     assertTrue(p==refOfPBefore);  
8     assertTrue(p.x==6 && p.y==8);  
9 }
```

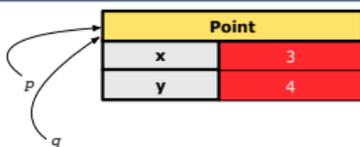
- **Before** the mutator call at **L6**, **reference** variable `p` stores the **address** of some `Point` object (whose `x` is 3 and `y` is 4).
- **When** executing the mutator call at **L6**, due to **call by reference**, a **copy of address** stored in `p` is made. [**Alias**: `p` and `q` store same address.]
⇒ Calls to `q.moveHorizontally` and `q.moveVertically` are effective on both `p` and `q`.
- ∴ **After** the mutator call at **L6**, variable `p` still stores the original address (i.e., same as `refOfPBefore`), but its `x` and `y` have been modified via `q`.

Call by Value vs. Call by Reference (2.4.2)

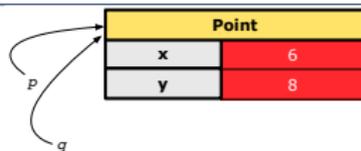
Before `changeViaRef`



During `changeViaRef`



After `changeViaRef`



Aggregation: Independent Containees Shared by Containers (1.1)



```
class Course {
    String title;
    Faculty prof;
    Course(String title) {
        this.title = title;
    }
    void setProf(Faculty prof) {
        this.prof = prof;
    }
    Faculty getProf() {
        return this.prof;
    }
}
```

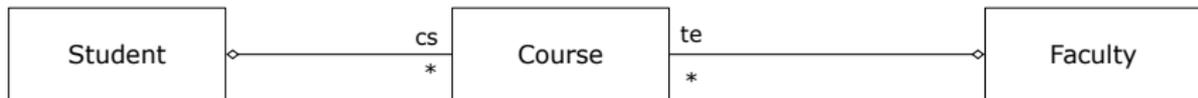
```
class Faculty {
    String name;
    Faculty(String name) {
        this.name = name;
    }
    void setName(String name) {
        this.name = name;
    }
    String getName() {
        return this.name;
    }
}
```

Aggregation: Independent Containees Shared by Containers (1.2)

```
@Test
public void testAggregation1() {
    Course eecs2030 = new Course("Advanced OOP");
    Course eecs3311 = new Course("Software Design");
    Faculty prof = new Faculty("Jackie");
    eecs2030.setProf(prof);
    eecs3311.setProf(prof);
    assertTrue(eecs2030.getProf() == eecs3311.getProf());
    /* aliasing */
    prof.setName("Jeff");
    assertTrue(eecs2030.getProf() == eecs3311.getProf());
    assertTrue(eecs2030.getProf().getName().equals("Jeff"));

    Faculty prof2 = new Faculty("Jonathan");
    eecs3311.setProf(prof2);
    assertTrue(eecs2030.getProf() != eecs3311.getProf());
    assertTrue(eecs2030.getProf().getName().equals("Jeff"));
    assertTrue(eecs3311.getProf().getName().equals("Jonathan"));
}
```

Aggregation: Independent Containees Shared by Containers (2.1)



```
class Student {
    String id; ArrayList<Course> cs; /* courses */
    Student(String id) { this.id = id; cs = new ArrayList<>(); }
    void addCourse(Course c) { cs.add(c); }
    ArrayList<Course> getCS() { return cs; }
}
```

```
class Course { String title; }
```

```
class Faculty {
    String name; ArrayList<Course> te; /* teaching */
    Faculty(String name) { this.name = name; te = new ArrayList<>(); }
    void addTeaching(Course c) { te.add(c); }
    ArrayList<Course> getTE() { return te; }
}
```

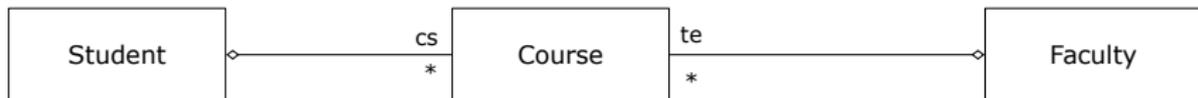
Aggregation: Independent Containees Shared by Containers (2.2)

```
@Test
public void testAggregation2() {
    Faculty p = new Faculty("Jackie");
    Student s = new Student("Jim");
    Course eeecs2030 = new Course("Advanced OOP");
    Course eeecs3311 = new Course("Software Design");
    eeecs2030.setProf(p);
    eeecs3311.setProf(p);
    p.addTeaching(eeecs2030);
    p.addTeaching(eeecs3311);
    s.addCourse(eeecs2030);
    s.addCourse(eeecs3311);

    assertTrue(eeecs2030.getProf() == s.getCS().get(0).getProf());
    assertTrue(s.getCS().get(0).getProf() == s.getCS().get(1).getProf());
    assertTrue(eeecs3311 == s.getCS().get(1));
    assertTrue(s.getCS().get(1) == p.getTE().get(1));
}
```

OOP: The Dot Notation (3.1)

In real life, the relationships among classes are sophisticated.



```

class Student {
    String id;
    ArrayList<Course> cs;
}
  
```

```

class Course {
    String title;
    Faculty prof;
}
  
```

```

class Faculty {
    String name;
    ArrayList<Course> te;
}
  
```

Aggregation links between classes constrain how you can **navigate** among these classes.

e.g., In the context of class `Student`:

- Writing `cs` denotes the list of registered courses.
- Writing `cs[i]` (where `i` is a valid index) navigates to the class `Course`, which changes the context to class `Course`.

OOP: The Dot Notation (3.2)

```
class Student {  
    String id;  
    ArrayList<Course> cs;  
}
```

```
class Course {  
    String title;  
    Faculty prof;  
}
```

```
class Faculty {  
    String name;  
    ArrayList<Course> te;  
}
```

```
class Student {  
    ... /* attributes */  
    /* Get the student's id */  
    String getID() { return this.id; }  
    /* Get the title of the ith course */  
    String getCourseTitle(int i) {  
        return this.cs.get(i).title;  
    }  
    /* Get the instructor's name of the ith course */  
    String getInstructorName(int i) {  
        return this.cs.get(i).prof.name;  
    }  
}
```

OOP: The Dot Notation (3.3)

```
class Student {  
    String id;  
    ArrayList<Course> cs;  
}
```

```
class Course {  
    String title;  
    Faculty prof;  
}
```

```
class Faculty {  
    String name;  
    ArrayList<Course> te;  
}
```

```
class Course {  
    ... /* attributes */  
    /* Get the course's title */  
    String getTitle() { return this.title; }  
    /* Get the instructor's name */  
    String getInstructorName() {  
        return this.prof.name;  
    }  
    /* Get title of ith teaching course of the instructor */  
    String getCourseTitleOfInstructor(int i) {  
        return this.prof.te.get(i).title;  
    }  
}
```

OOP: The Dot Notation (3.4)

```
class Student {  
    String id;  
    ArrayList<Course> cs;  
}
```

```
class Course {  
    String title;  
    Faculty prof;  
}
```

```
class Faculty {  
    String name;  
    ArrayList<Course> te;  
}
```

```
class Faculty {  
    ... /* attributes */  
    /* Get the instructor's name */  
    String getName() {  
        return this.name;  
    }  
    /* Get the title of ith teaching course */  
    String getCourseTitle(int i) {  
        return this.te.get(i).title;  
    }  
}
```

Composition: Dependent Containees Owned by Containers (1.1)



Assumption: Files are not shared among directories.

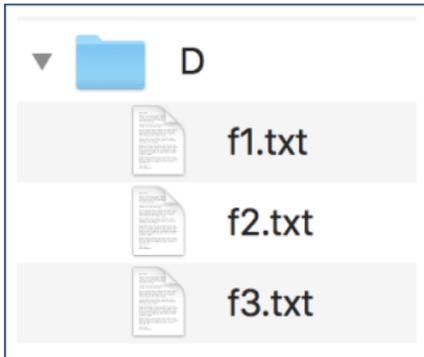
```

class File {
    String name;
    File(String name) {
        this.name = name;
    }
}
  
```

```

class Directory {
    String name;
    File[] files;
    int nof; /* num of files */
    Directory(String name) {
        this.name = name;
        files = new File[100];
    }
    void addFile(String fileName) {
        files[nof] = new File(fileName);
        nof ++;
    }
}
  
```

Composition: Dependent Containees Owned by Containers (1.2.1)



```

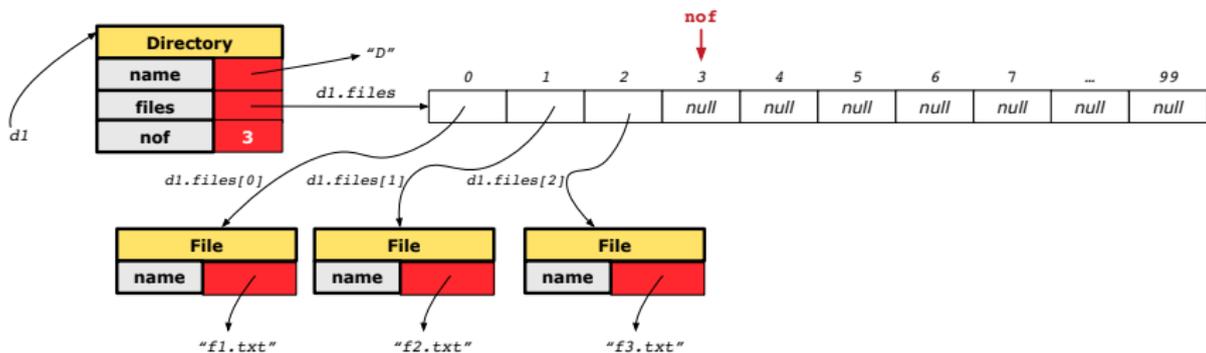
1  @Test
2  public void testComposition() {
3      Directory d1 = new Directory("D");
4      d1.addFile("f1.txt");
5      d1.addFile("f2.txt");
6      d1.addFile("f3.txt");
7      assertTrue(
8          d1.files[0].name.equals("f1.txt"));
9  }

```

- **L4:** a 1st `File` object is created and **owned exclusively** by `d1`. No other directories are sharing this `File` object with `d1`.
- **L5:** a 2nd `File` object is created and **owned exclusively** by `d1`. No other directories are sharing this `File` object with `d1`.
- **L6:** a 3rd `File` object is created and **owned exclusively** by

Composition: Dependent Containees Owned by Containers (1.2.2)

Right before test method `testComposition` terminates:



Composition: Dependent Containees Owned by Containers (1.3)

Problem: How do you implement a *copy instructor* for the `Directory` class?

```
class Directory {  
    Directory(Directory other) {  
        /* ?? */  
    }  
}
```

Hints:

- The implementation should be consistent with the effect of copying and pasting a directory.
- Separate copies of files are created.

Composition: Dependent Containees Owned by Containers (1.4.1)

Version 1: *Shallow Copy* by copying all attributes using =.

```
class Directory {  
    Directory(Directory other) {  
        /* value copying for primitive type */  
        nof = other.nof;  
        /* address copying for reference type */  
        name = other.name; files = other.files; } }  
}
```

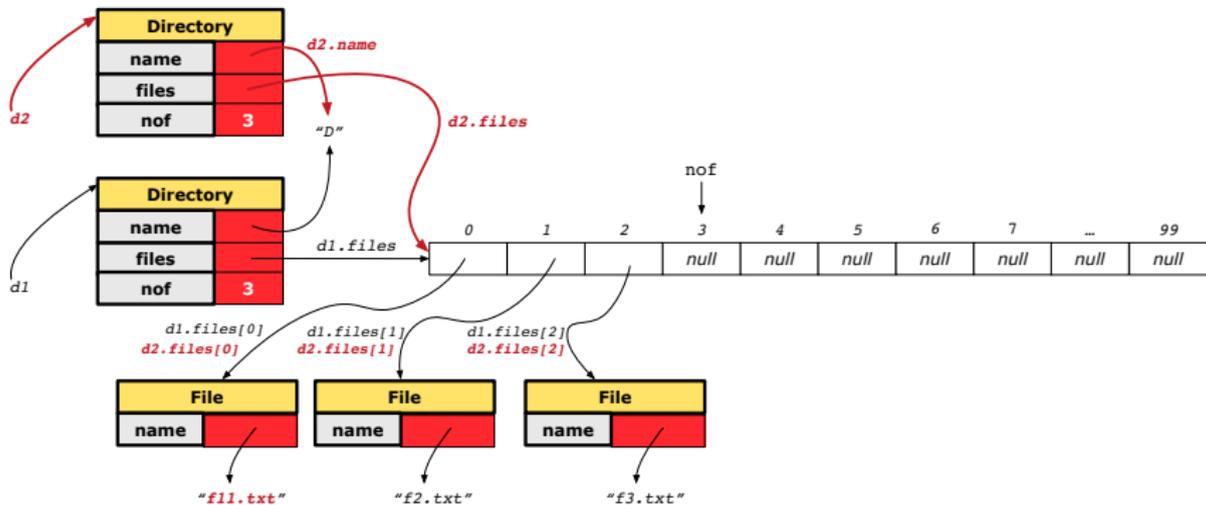
Is a shallow copy satisfactory to support composition?
i.e., Does it still forbid sharing to occur?

[**NO**]

```
@Test  
void testShallowCopyConstructor() {  
    Directory d1 = new Directory("D");  
    d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");  
    Directory d2 = new Directory(d1);  
    assertTrue(d1.files == d2.files); /* violation of composition */  
    d2.files[0].changeName("f11.txt");  
    assertFalse(d1.files[0].name.equals("f1.txt")); }  
}
```

Composition: Dependent Containees Owned by Containers (1.4.2)

Right before test method `testShallowCopyConstructor` terminates:



Composition: Dependent Containees Owned by Containers (1.5.1)

Version 2: a **Deep Copy**

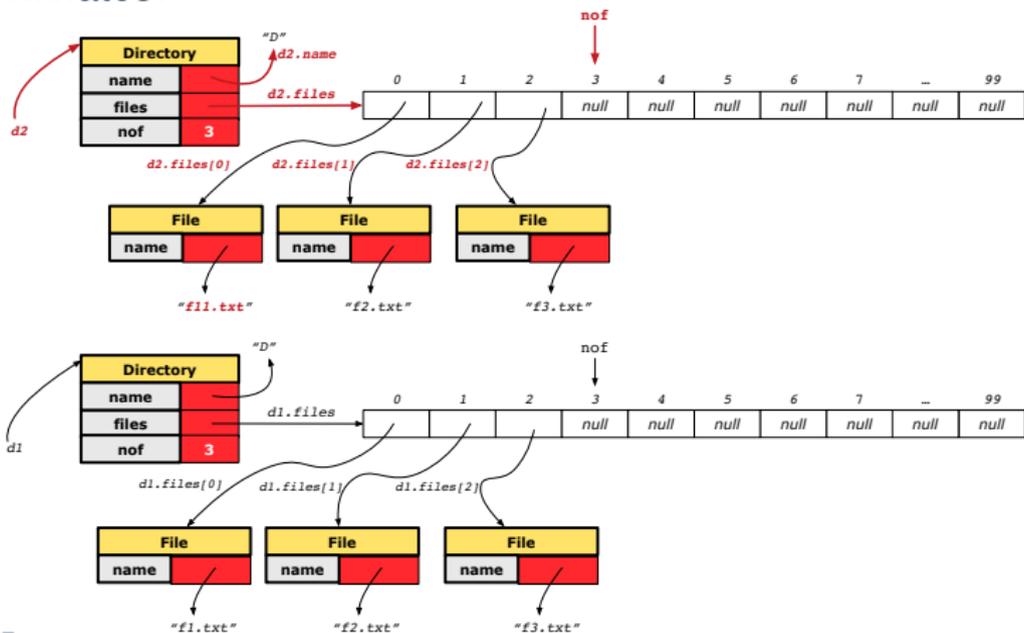
```
class File {  
    File(File other) {  
        this.name =  
            new String(other.name);  
    }  
}
```

```
class Directory {  
    Directory(String name) {  
        this.name = new String(name);  
        files = new File[100]; }  
    Directory(Directory other) {  
        this (other.name);  
        for(int i = 0; i < nof; i ++ ) {  
            File src = other.files[i];  
            File nf = new File(src);  
            this.addFile(nf); } } }
```

```
@Test  
void testDeepCopyConstructor() {  
    Directory d1 = new Directory("D");  
    d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");  
    Directory d2 = new Directory(d1);  
    assertTrue(d1.files != d2.files); /* composition preserved */  
    d2.files[0].changeName("f11.txt");  
    assertTrue(d1.files[0].name.equals("f1.txt")); } }
```

Composition: Dependent Containees Owned by Containers (1.5.2)

Right before test method `testDeepCopyConstructor` terminates:



Composition: Dependent Containees Owned by Containers (1.6)

Exercise: Implement the accessor in class `Directory`

```
class Directory {  
    File[] files;  
    int nof;  
    File[] getFiles() {  
        /* Your Task */  
    }  
}
```

so that it **preserves composition**, i.e., does not allow references of files to be shared.

Aggregation vs. Composition (1)

Terminology:

- **Container** object: an object that contains others.
- **Containee** object: an object that is contained within another.

Aggregation :

- Containees (e.g., Course) may be *shared* among containers (e.g., Student, Faculty).
- Containees *exist independently* without their containers.
- When a container is destroyed, its containees still exist.

Composition :

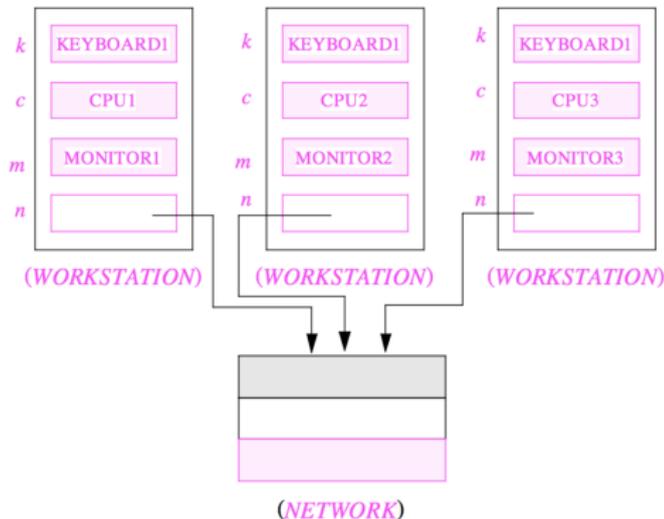
- Containers (e.g., Directory, Department) *own* exclusive access to their containees (e.g., File, Faculty).
- Containees cannot exist without their containers.
- Destroying a container destroys its containees *cascadingly*.

Aggregation vs. Composition (2)

Aggregations and *Compositions* may exist at the same time!

e.g., Consider a workstation:

- Each workstation owns CPU, monitor, keyboard. [*compositions*]
- All workstations share the same network. [*aggregations*]



OOP: Equality (1)

```
Point p1 = new Point(2, 3);  
Point p2 = new Point(2, 3);  
boolean sameLoc = ( p1 == p2 );  
System.out.println("p1 and p2 same location?" + sameLoc);
```

```
p1 and p2 same location? false
```

OOP: Equality (2)

- Recall that
 - A **primitive** variable stores a primitive *value*
e.g., `double d1 = 7.5; double d2 = 7.5;`
 - A **reference** variable stores the *address* to some object (rather than storing the object itself)
e.g., `Point p1 = new Point(2, 3)` assigns to `p1` the address of the new `Point` object
e.g., `Point p2 = new Point(2, 3)` assigns to `p2` the address of *another* new `Point` object
- The binary operator `==` may be applied to compare:
 - **Primitive** variables: their *contents* are compared
e.g., `d1 == d2` evaluates to *true*
 - **Reference** variables: the *addresses* they store are compared (rather than comparing contents of the objects they refer to)
e.g., `p1 == p2` evaluates to *false* because `p1` and `p2` are addresses of *different* objects, even if their contents are *identical*.

OOP: Equality (3)

- Implicitly:
 - Every class is a *child/sub* class of the `Object` class.
 - The `Object` class is the *parent/super* class of every class.
- There are two useful *accessor methods* that every class *inherits* from the `Object` class:
 - `boolean equals(Object other)`
Indicates whether some other object is “equal to” this one.
 - The default definition inherited from `Object`:

```
boolean equals(Object other) {  
    return (this == other); }
```
 - `String toString()`
Returns a string representation of the object.
- Very often when you define new classes, you want to *redefine* / *override* the inherited definitions of `equals` and `toString`.

OOP: Contract of equals

Given that reference variables `x`, `y`, `z` are not `null`:

- $\neg x.equals(null)$
- **Reflexive**:
 $x.equals(x)$
- **Symmetric**
 $x.equals(y) \iff y.equals(x)$
- **Transitive**
 $x.equals(y) \wedge y.equals(z) \Rightarrow x.equals(z)$

OOP: Equality (4.1)

- How do we compare *contents* rather than addresses?
- Define the **accessor method** `equals`, e.g.,

```
class Point {
    double x; double y;
    boolean equals(Object obj) {
        if(this == obj) { return true; }
        if(obj == null) { return false; }
        if(this.getClass() != obj.getClass()) { return false; }
        Point other = (Point) obj;
        return this.x == other.x && this.y == other.y; } }

```

```
class PointTester {
    String s = "(2, 3)";
    Point p1 = new Point(2, 3); Point p2 = new Point(2, 3);
    System.out.println(p1.equals(p1)); /* true */
    System.out.println(p1.equals(null)); /* false */
    System.out.println(p1.equals(s)); /* false */
    System.out.println(p1 == p2); /* false */
    System.out.println(p1.equals(p2)); /* true */ }

```

OOP: Equality (4.2)

- When making a method call `p.equals(o)`:
 - Variable `p` is of type `Point`
 - Variable `o` can be any type
- We define `p` and `o` as **equal** if:
 - Either `p` and `o` refer to the same object;
 - Or:
 - `o` is not null.
 - `p` and `o` are of the same type.
 - The `x` and `y` coordinates are the same.
- **Q:** In the `equals` method of `Point`, why is there no such a line:

```
class Point {  
    boolean equals(Object obj) {  
        if(this == null) { return false; }  
    }  
}
```

A: If `this` is `null`, a `NullPointerException` would have occurred and prevent the body of `equals` from being executed.

OOP: Equality (4.3)

```
1 class Point {
2     boolean equals (Object obj) {
3         ...
4         Point other = (Point) obj;
5         return this.x == other.x && this.y == other.y; } }
```

- `Object obj` at **L2** declares a parameter `obj` of type `Object`.

- `Point p` at **L4** declares a variable `p` of type `Point`.

We call such types declared at compile time as **static type**.

- The list of applicable methods that we may call on a variable depends on its **static type**.

e.g., We may only call the small list of methods defined in `Object` class on `obj`, which does not include `x` and `y` (specific to `Point`).

- If we are SURE that an object's "actual" type is different from its **static type**, then we can **cast** it.

e.g., Given that `this.getClass() == obj.getClass()`, we are sure that `obj` is also a `Point`, so we can cast it to `Point`.

- Such cast allows more attributes/methods to be called upon

89 of 147 `(Point) obj` at **L5**.

OOP: Equality (5.1)

Exercise: Persons are *equal* if names and measures are equal.

```
1  class Person {
2      String firstName; String lastName; double weight; double height;
3      boolean equals (Object obj) {
4          if(this == obj) { return true }
5          if(obj == null || this.getClass() != obj.getClass()) {
6              return false; }
7          Person other = (Person) obj;
8          return
9              this.weight == other.weight && this.height == other.height
10             && this.firstName.equals (other.firstName)
11             && this.lastName.equals (other.lastName) } }
```

Q: At L5, will we get NullPointerException if obj is Null?

A: **No** ∴ Short-Circuit Effect of ||

obj is null, then obj == null evaluates to **true**

⇒ no need to evaluate the RHS

The left operand obj == null acts as a **guard constraint** for the right operand this.getClass() != obj.getClass().

OOP: Equality (5.2)

Exercise: Persons are *equal* if names and measures are equal.

```
1 class Person {
2   String firstName; String lastName; double weight; double height;
3   boolean equals (Object obj) {
4     if(this == obj) { return true }
5     if(obj == null || this.getClass() != obj.getClass()) {
6       return false; }
7     Person other = (Person) obj;
8     return
9       this.weight == other.weight && this.height == other.height
10      && this.firstName.equals (other.firstName)
11      && this.lastName.equals (other.lastName) } }
```

Q: At L5, if swapping the order of two operands of disjunction:

`this.getClass() != obj.getClass() || obj == null`

Will we get `NullPointerException` if `obj` is `Null`?

A: **Yes** ∴ Evaluation of operands is from left to right.

OOP: Equality (5.3)

Exercise: Persons are *equal* if names and measures are equal.

```
1 class Person {
2   String firstName; String lastName; double weight; double height;
3   boolean equals(Object obj) {
4     if(this == obj) { return true }
5     if(obj == null || this.getClass() != obj.getClass()) {
6       return false; }
7     Person other = (Person) obj;
8     return
9       this.weight == other.weight && this.height == other.height
10      && this.firstName.equals(other.firstName)
11      && this.lastName.equals(other.lastName) } }
```

L10 & L11 call equals method defined in the String class.

When defining equals method for your own class, **reuse** equals methods defined in other classes wherever possible.

OOP: Equality (6)

Two notions of **equality** for variables of *reference* types:

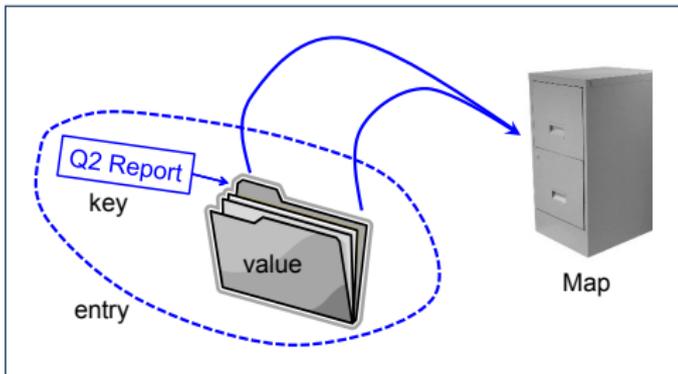
- **Reference Equality**: use `==` to compare *addresses*
- **Object Equality**: define `equals` method to compare *contents*

```
1 Point p1 = new Point(3, 4);
2 Point p2 = new Point(3, 4);
3 Point p3 = new Point(4, 5);
4 System.out.println(p1 == p1); /* true */
5 System.out.println(p1.equals(p1)); /* true */
6 System.out.println(p1 == p2); /* false */
7 System.out.println(p1.equals(p2)); /* true */
8 System.out.println(p2 == p3); /* false */
9 System.out.println(p2.equals(p3)); /* false */
```

- Being *reference*-equal implies being *object*-equal
- Being *object*-equal does **not** imply being *reference*-equal

Hashing: What is a Map?

- A **map** (a.k.a. table or dictionary) stores a collection of *entries*.



ENTRY	
(SEARCH) KEY	VALUE
1	D
25	C
3	F
14	Z
6	A
39	C
7	Q

- Each **entry** is a pair: a *value* and its (*search*) *key*.
- Each **search key** :
 - Uniquely* identifies an object in the map
 - Should be used to *efficiently* retrieve the associated value
- Search keys must be *unique* (i.e., do not contain duplicates).

Hashing: Arrays are Maps

- Each array *entry* is a pair: an object and its *numerical* index.
e.g., say `String[] a = {"A", "B", "C"}`, how many entries?
3 entries: `(0, "A")`, `(1, "B")`, `(2, "C")`
- Search keys* are the set of numerical index values.
- The set of index values are *unique* [e.g., $0 \dots (a.length - 1)$]
- Given a *valid* index value i , we can
 - Uniquely* determines where the object is $[(i + 1)^{th} \text{ item}]$
 - Efficiently* retrieves that object $[a[i] \approx \text{fast memory access}]$
- Maps in general may have *non-numerical* key values:
 - Student ID [student record]
 - Social Security Number [resident record]
 - Passport Number [citizen record]
 - Residential Address [household record]
 - Media Access Control (MAC) Address [PC/Laptop record]
 - Web URL [web page]

Hashing: Naive Implementation of Map

- **Problem:** Support the construction of this simple map:

ENTRY	
(SEARCH) KEY	VALUE
1	D
25	C
3	F
14	Z
6	A
39	C
7	Q

Let's just assume that the maximum map capacity is 100.

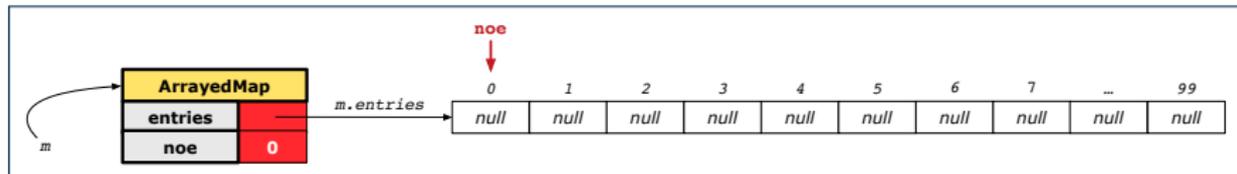
- **Naive Solution:**

Let's understand the expected runtime structures before seeing the Java code!

Hashing: Naive Implementation of Map (0)

After executing `ArrayedMap m = new ArrayedMap()`:

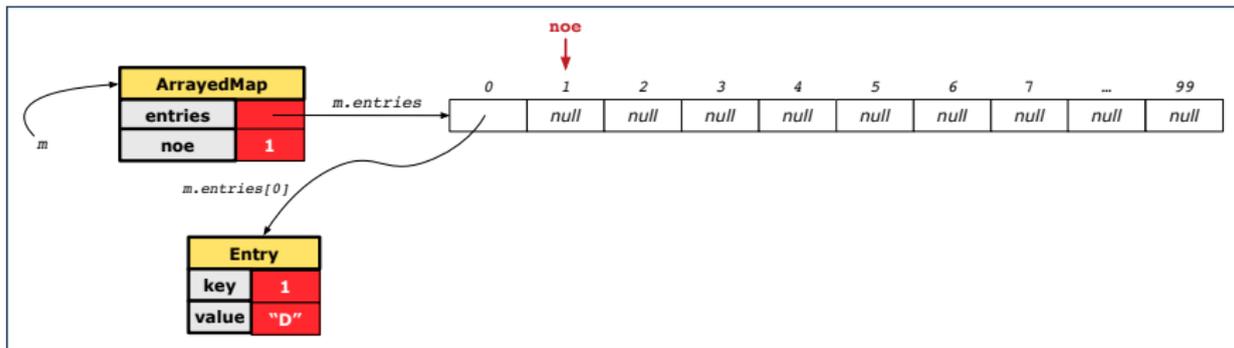
- Attribute `m.entries` initialized as an array of 100 `null` slots.
- Attribute `m.noe` is 0, meaning:
 - Current number of entries stored in the map is 0.
 - Index for storing the next new entry is 0.



Hashing: Naive Implementation of Map (1)

After executing `m.put(new Entry(1, "D"))`:

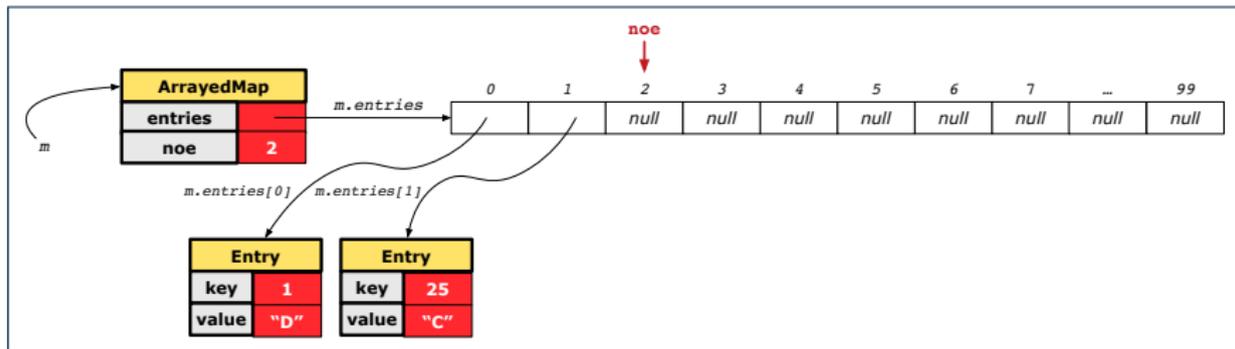
- Attribute `m.entries` has 99 null slots.
- Attribute `m.noE` is 1, meaning:
 - Current number of entries stored in the map is 1.
 - Index for storing the next new entry is 1.



Hashing: Naive Implementation of Map (2)

After executing `m.put(new Entry(25, "C"))`:

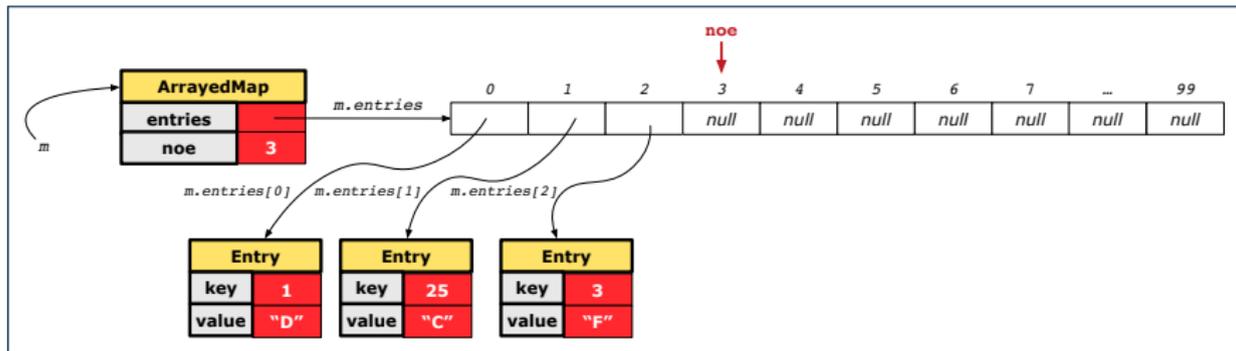
- Attribute `m.entries` has 98 null slots.
- Attribute `m.noe` is 2, meaning:
 - Current number of entries stored in the map is 2.
 - Index for storing the next new entry is 2.



Hashing: Naive Implementation of Map (3)

After executing `m.put(new Entry(3, "F"))`:

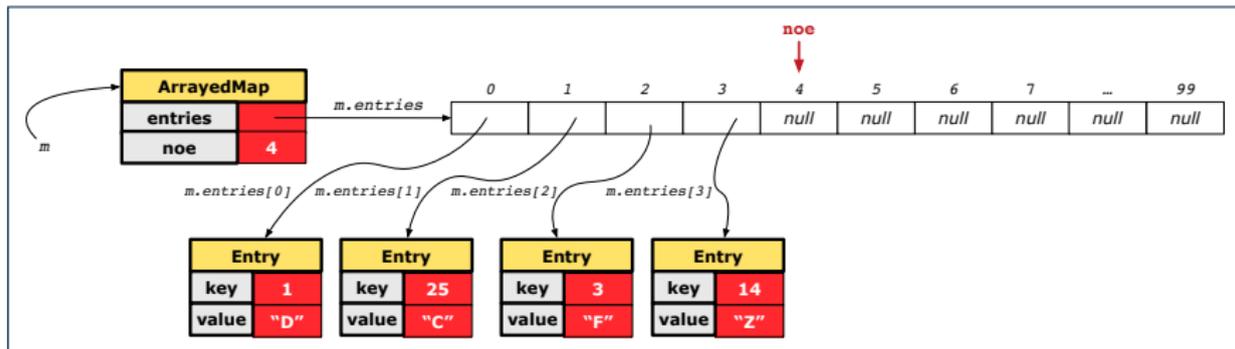
- Attribute `m.entries` has 97 null slots.
- Attribute `m.noe` is 3, meaning:
 - Current number of entries stored in the map is 3.
 - Index for storing the next new entry is 3.



Hashing: Naive Implementation of Map (4)

After executing `m.put(new Entry(14, "Z"))`:

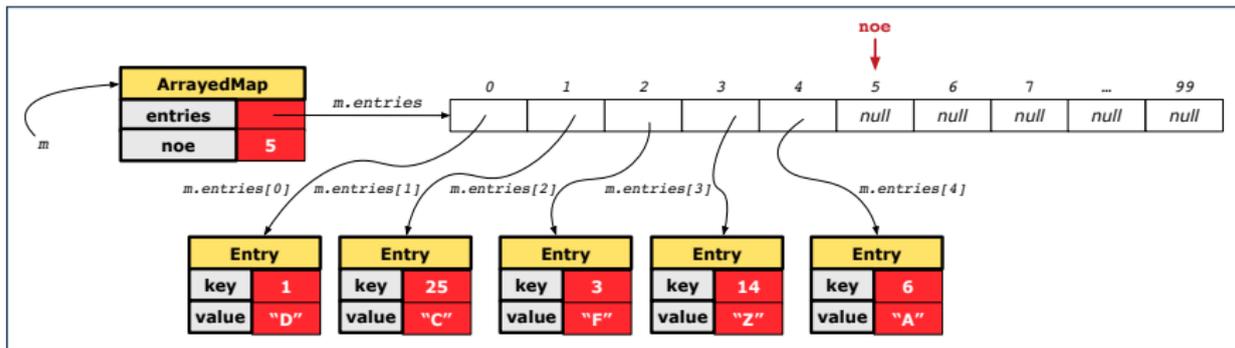
- Attribute `m.entries` has 96 null slots.
- Attribute `m.noE` is 4, meaning:
 - Current number of entries stored in the map is 4.
 - Index for storing the next new entry is 4.



Hashing: Naive Implementation of Map (5)

After executing `m.put(new Entry(6, "A"))`:

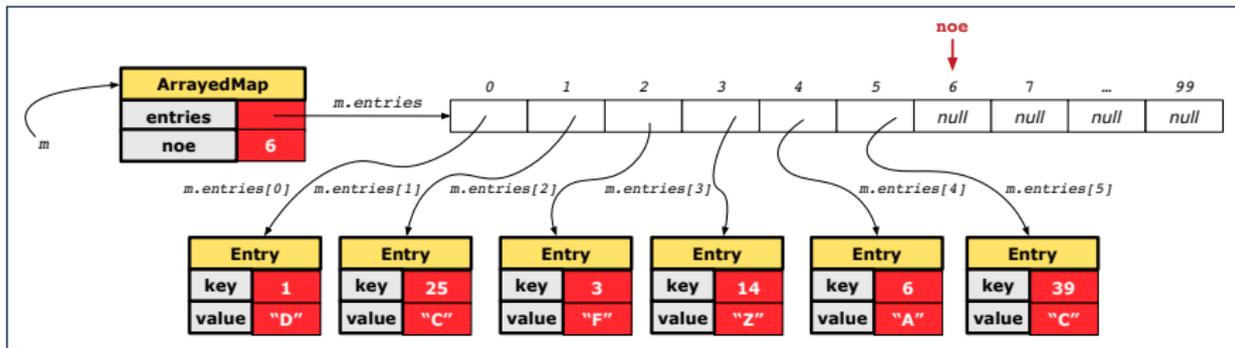
- Attribute `m.entries` has 95 null slots.
- Attribute `m.noë` is 5, meaning:
 - Current number of entries stored in the map is 5.
 - Index for storing the next new entry is 5.



Hashing: Naive Implementation of Map (6)

After executing `m.put(new Entry(39, "C"))`:

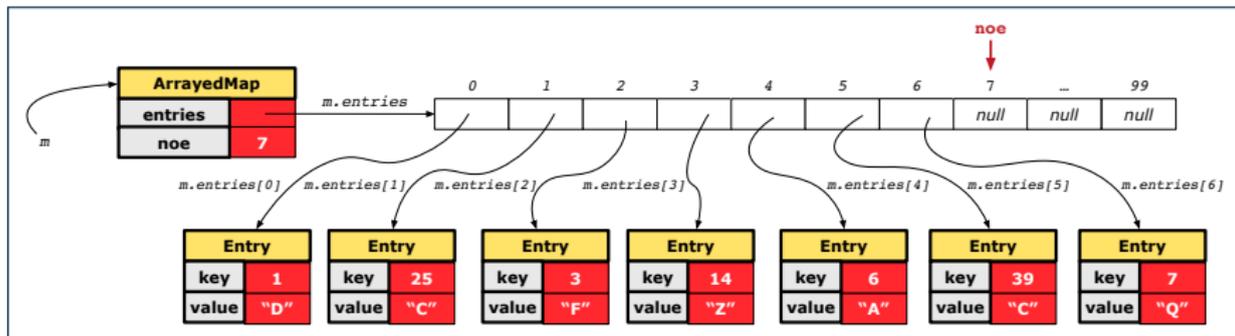
- Attribute `m.entries` has 94 null slots.
- Attribute `m.noë` is 6, meaning:
 - Current number of entries stored in the map is 6.
 - Index for storing the next new entry is 6.



Hashing: Naive Implementation of Map (7)

After executing `m.put(new Entry(7, "Q"))`:

- Attribute `m.entries` has 93 null slots.
- Attribute `m.noE` is 7, meaning:
 - Current number of entries stored in the map is 7.
 - Index for storing the next new entry is 7.



Hashing: Naive Implementation of Map (8.1)

```
public class Entry {  
    private int key;  
    private String value;  
  
    public Entry(int key, String value) {  
        this.key = key;  
        this.value = value;  
    }  
    /* Getters and Setters for key and value */  
}
```

Hashing: Naive Implementation of Map (8.2)

```
public class ArrayedMap {
    private final int MAX_CAPACITY = 100;
    private Entry[] entries;
    private int noe; /* number of entries */
    public ArrayedMap() {
        entries = new Entry[MAX_CAPACITY];
        noe = 0;
    }
    public int size() {
        return noe;
    }
    public void put(int key, String value) {
        Entry e = new Entry(key, value);
        entries[noe] = e;
        noe ++;
    }
}
```

Required Reading: Point and PointCollector

Hashing: Naive Implementation of Map (8.3)

```
@Test
public void testArrayedMap() {
    ArrayedMap m = new ArrayedMap();
    assertTrue(m.size() == 0);
    m.put(1, "D");
    m.put(25, "C");
    m.put(3, "F");
    m.put(14, "Z");
    m.put(6, "A");
    m.put(39, "C");
    m.put(7, "Q");
    assertTrue(m.size() == 7);
    /* inquiries of existing key */
    assertTrue(m.get(1).equals("D"));
    assertTrue(m.get(7).equals("Q"));
    /* inquiry of non-existing key */
    assertTrue(m.get(31) == null);
}
```

Hashing: Naive Implementation of Map (8.4)

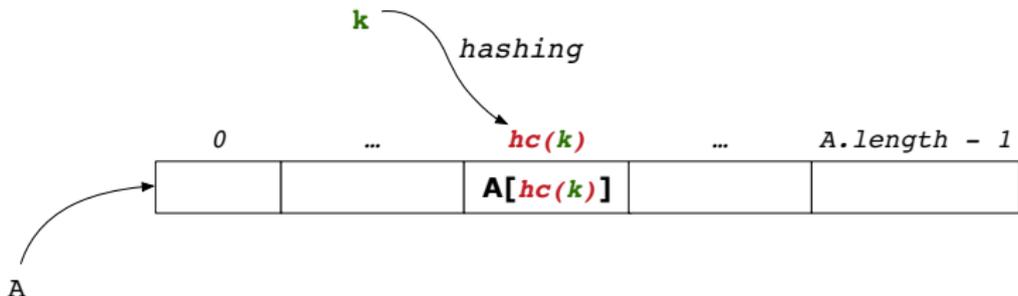
```
public class ArrayedMap {
    private final int MAX_CAPACITY = 100;
    public String getValue(int key) {
        for(int i = 0; i < noe; i++) {
            Entry e = entries[i];
            int k = e.getKey();
            if(k == key) { return e.getValue(); }
        }
        return null;
    }
}
```

Say entries is: {(1, D), (25, C), (3, F), (14, Z), (6, A), (39, C), (7, Q), null, ... }

- o How efficient is `m.get(1)`? [1 iteration]
 - o How efficient is `m.get(7)`? [7 iterations]
 - o If `m` is full, worst case of `m.get(k)`? [100 iterations]
 - o If `m` with 10^6 entries, worst case of `m.get(k)`? [10^6 iterations]
- ⇒ `get`'s worst-case performance is **linear** on size of `m.entries`!

A much **faster** (and **correct**) solution is possible!

Hashing: Hash Table (1)

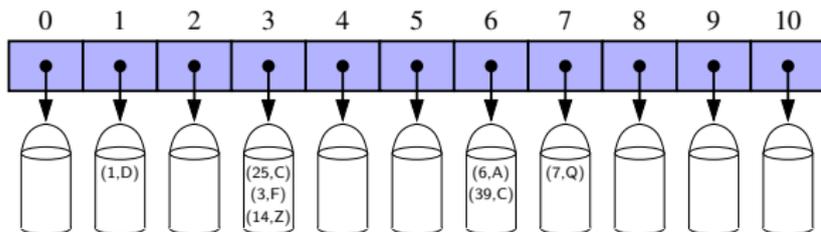


- Given a (numerical or non-numerical) search key k :
 - Apply a function hc so that $hc(k)$ returns an integer.
 - We call $hc(k)$ the **hash code** of key k .
 - Value of $hc(k)$ denotes a **valid index** of some array A .
 - Rather than searching through array A , go directly to $A[hc(k)]$ to get the associated value.
- Both computations are fast:
 - Converting k to $hc(k)$
 - Indexing into $A[hc(k)]$

Hashing: Hash Table as a Bucket Array (2)

For illustration, assume $A.length$ is 10 and $hc(k) = k \% 11$.

$hc(k) = k \% 11$	(SEARCH) KEY	VALUE
1	1	D
3	25	C
3	3	F
3	14	Z
6	6	A
6	39	C
7	7	Q



- Collision:** unequal keys have same hash code (e.g., 25, 3, 14)
 \Rightarrow Unavoidable as number of entries \uparrow , but a *good* hash function should have sizes of the buckets uniformly distributed.

Hashing: Contract of Hash Function

- Principle of defining a hash function *hc*:

$$k1.equals(k2) \Rightarrow hc(k1) == hc(k2)$$

Equal keys always have the same hash code.

- Equivalently, according to contrapositive:

$$hc(k1) \neq hc(k2) \Rightarrow \neg k1.equals(k2)$$

Different hash codes must be generated from unequal keys.

inconsistent hashCode and equals

Hashing: Defining Hash Function in Java (1)

The `Object` class (common super class of all classes) has the method for redefining the hash function for your own class:

```
public class IntegerKey {
    private int k;
    public IntegerKey(int k) { this.k = k; }
    @Override
    public int hashCode() { return k % 11; }
    @Override
    public boolean equals(Object obj) {
        if(this == obj) { return true; }
        if(obj == null) { return false; }
        if(this.getClass() != obj.getClass()) { return false; }
        IntegerKey other = (IntegerKey) obj;
        return this.k == other.k;
    }
}
```

Q: Can we define `equals` as `return this.hashCode == other.hashCode () ?` [**No** :: Collision; see contract of `equals`]

Hashing: Defining Hash Function in Java (2)

```
@Test
public void testCustomizedHashFunction() {
    IntegerKey ik1 = new IntegerKey(1);
    /* 1 % 11 == 1 */
    assertTrue(ik1.hashCode() == 1);

    IntegerKey ik39_1 = new IntegerKey(39);
    /* 39 % 11 == 3 */
    assertTrue(ik39_1.hashCode() == 6);

    IntegerKey ik39_2 = new IntegerKey(39);
    assertTrue(ik39_1.equals(ik39_2));
    assertTrue(ik39_1.hashCode() == ik39_2.hashCode());
}
```

Hashing: Using Hash Table in Java

```
@Test
public void testHashTable() {
    Hashtable<IntegerKey, String> table = new Hashtable<>();
    IntegerKey k1 = new IntegerKey(39);
    IntegerKey k2 = new IntegerKey(39);
    assertTrue(k1.equals(k2));
    assertTrue(k1.hashCode() == k2.hashCode());
    table.put(k1, "D");
    assertTrue(table.get(k2).equals("D"));
}
```

Hashing: Defining Hash Function in Java (3)



- When you are given instructions as to how the `hashCode` method of a class should be defined, override it manually.
- Otherwise, use Eclipse to generate the `equals` and `hashCode` methods for you.
 - Right click on the class.
 - Select `Source`.
 - Select `Generate hashCode() and equals()`.
 - Select the relevant attributes that will be used to compute the hash value.

Hashing: Defining Hash Function in Java (4.1)

Caveat: Always make sure that the `hashCode` and `equals` are redefined/overridden to work together consistently.
e.g., Consider an alternative version of the `IntegerKey` class:

```
public class IntegerKey {
    private int k;
    public IntegerKey(int k) { this.k = k; }
    /* hashCode() inherited from Object NOT overridden. */
    @Override
    public boolean equals(Object obj) {
        if(this == obj) { return true; }
        if(obj == null) { return false; }
        if(this.getClass() != obj.getClass()) { return false; }
        IntegerKey other = (IntegerKey) obj;
        return this.k == other.k;
    }
}
```

Problem?

[**Hint:** Contract of `hashCode ()`]

Hashing: Defining Hash Function in Java (4.2)

```
1  @Test
2  public void testDefaultHashFunction() {
3      IntegerKey ik39_1 = new IntegerKey(39);
4      IntegerKey ik39_2 = new IntegerKey(39);
5      assertTrue(ik39_1.equals(ik39_2));
6      assertTrue(ik39_1.hashCode() != ik39_2.hashCode()); }
7  @Test
8  public void testHashTable() {
9      Hashtable<IntegerKey, String> table = new Hashtable<>();
10     IntegerKey k1 = new IntegerKey(39);
11     IntegerKey k2 = new IntegerKey(39);
12     assertTrue(k1.equals(k2));
13     assertTrue(k1.hashCode() != k2.hashCode());
14     table.put(k1, "D");
15     assertTrue(table.get(k2) == null); }
```

L3, 4, 11, 12: Default version of `hashCode`, inherited from `Object`, returns a *distinct* integer for every new object, *despite its contents*. [**Fix:** Override `hashCode` of your classes!]

Why Ordering Between Objects? (1)

Each employee has their numerical id and salary.

e.g., (*alan*, 2, 4500.34), (*mark*, 3, 3450.67), (*tom*, 1, 3450.67)

- **Problem**: To facilitate an annual review on their statuses, we want to arrange them so that ones with smaller id's come before ones with larger id's.s
e.g., $\langle \textit{tom}, \textit{alan}, \textit{mark} \rangle$
- Even better, arrange them so that ones with larger salaries come first; only compare id's for employees with equal salaries.
e.g., $\langle \textit{alan}, \textit{tom}, \textit{mark} \rangle$
- **Solution** :
 - Define **ordering** of Employee objects.
[Comparable interface, compareTo method]
 - Use the library method `Arrays.sort`.

Why Ordering Between Objects? (2)

```
class Employee {  
    int id; double salary;  
    Employee(int id) { this.id = id; }  
    void setSalary(double salary) { this.salary = salary; } }
```

```
1 @Test  
2 public void testUncomparableEmployees() {  
3     Employee alan = new Employee(2);  
4     Employee mark = new Employee(3);  
5     Employee tom = new Employee(1);  
6     Employee[] es = {alan, mark, tom};  
7     Arrays.sort(es);  
8     Employee[] expected = {tom, alan, mark};  
9     assertEquals(expected, es); }
```

L8 triggers a *java.lang.ClassCastException*:
Employee cannot be cast to java.lang.Comparable

∴ `Arrays.sort` expects an array whose element type defines a precise **ordering** of its instances/objects.

Defining Ordering Between Objects (1.1)

```
class CEmployee1 implements Comparable<CEmployee1> {  
    ... /* attributes, constructor, mutator similar to Employee */  
    @Override  
    public int compareTo(CEmployee1 e) { return this.id - e.id; }  
}
```

- Given two CEmployee1 objects ce1 and ce2:
 - `ce1.compareTo(ce2) > 0` [ce1 “is greater than” ce2]
 - `ce1.compareTo(ce2) == 0` [ce1 “is equal to” ce2]
 - `ce1.compareTo(ce2) < 0` [ce1 “is smaller than” ce2]
- Say ces is an array of CEmployee1 (CEmployee1[] ces), calling `Arrays.sort(ces)` re-arranges ces, so that:

$$\underbrace{ces[0]}_{\text{CEmployee1 object}} \leq \underbrace{ces[1]}_{\text{CEmployee1 object}} \leq \dots \leq \underbrace{ces[ces.length - 1]}_{\text{CEmployee1 object}}$$

Defining Ordering Between Objects (1.2)

```
@Test
public void testComparableEmployees_1() {
    /*
     * CEmployee1 implements the Comparable interface.
     * Method compareTo compares id's only.
     */
    CEmployee1 alan = new CEmployee1(2);
    CEmployee1 mark = new CEmployee1(3);
    CEmployee1 tom = new CEmployee1(1);
    alan.setSalary(4500.34);
    mark.setSalary(3450.67);
    tom.setSalary(3450.67);
    CEmployee1[] es = {alan, mark, tom};
    /* When comparing employees,
     * their salaries are irrelevant.
     */
    Arrays.sort(es);
    CEmployee1[] expected = {tom, alan, mark};
    assertEquals(expected, es);
}
```

Defining Ordering Between Objects (2.1)

Let's now make the comparison more sophisticated:

- Employees with higher salaries come before those with lower salaries.
- When two employees have same salary, whoever with lower id comes first.

```
1 class CEmployee2 implements Comparable<CEmployee2> {
2     ... /* attributes, constructor, mutator similar to Employee */
3     @Override
4     public int compareTo(CEmployee2 other) {
5         int salaryDiff = Double.compare(this.salary, other.salary);
6         int idDiff = this.id - other.id;
7         if(salaryDiff != 0) { return -salaryDiff; }
8         else { return idDiff; } } }
```

- **L5:** `Double.compare(d1, d2)` returns
- $(d1 < d2)$, 0 ($d1 == d2$), or $+$ ($d1 > d2$).
- **L7:** Why inverting the sign of `salaryDiff`?
 - $this.salary > other.salary \Rightarrow Double.compare(this.salary, other.salary) > 0$
 - But we should consider employee with *higher* salary as “smaller”.
∴ We want that employee to come *before* the other one!

Defining Ordering Between Objects (2.2)

Alternatively, we can use extra `if` statements to express the logic more clearly.

```
1 class CEmployee2 implements Comparable<CEmployee2> {
2     ... /* attributes, constructor, mutator similar to Employee */
3     @Override
4     public int compareTo(CEmployee2 other) {
5         if(this.salary > other.salary) {
6             return -1;
7         }
8         else if (this.salary < other.salary) {
9             return 1;
10        }
11        else { /* equal salaries */
12            return this.id - other.id;
13        }
14    }
```

Defining Ordering Between Objects (2.3)

```
1  @Test
2  public void testComparableEmployees_2() {
3      /*
4       * CEmployee2 implements the Comparable interface.
5       * Method compareTo first compares salaries, then
6       * compares id's for employees with equal salaries.
7       */
8      CEmployee2 alan = new CEmployee2(2);
9      CEmployee2 mark = new CEmployee2(3);
10     CEmployee2 tom = new CEmployee2(1);
11     alan.setSalary(4500.34);
12     mark.setSalary(3450.67);
13     tom.setSalary(3450.67);
14     CEmployee2[] es = {alan, mark, tom};
15     Arrays.sort(es);
16     CEmployee2[] expected = {alan, tom, mark};
17     assertEquals(expected, es);
18 }
```

Defining Ordering Between Objects (3)

When you have your class `C` implement the interface `Comparable<C>`, you should design the `compareTo` method, such that given objects `c1`, `c2`, `c3` of type `C`:

- **Asymmetric** :

$$\neg(c1.compareTo(c2) < 0 \wedge c2.compareTo(c1) < 0)$$

$$\neg(c1.compareTo(c2) > 0 \wedge c2.compareTo(c1) > 0)$$

\therefore We don't have $c1 < c2$ and $c2 < c1$ at the same time!

- **Transitive** :

$$c1.compareTo(c2) < 0 \wedge c2.compareTo(c3) < 0 \Rightarrow c1.compareTo(c3) < 0$$

$$c1.compareTo(c2) > 0 \wedge c2.compareTo(c3) > 0 \Rightarrow c1.compareTo(c3) > 0$$

\therefore We have $c1 < c2 \wedge c2 < c3 \Rightarrow c1 < c3$

Q. How would you define the `compareTo` method for the `Player` class of a rock-paper-scissor game? [**Hint:** Transitivity]

Static Variables (1)

```
class Account {  
    int id;  
    String owner;  
    Account(int id, String owner) {  
        this.id = id;  
        this.owner = owner;  
    }  
}
```

```
class AccountTester {  
    Account acc1 = new Account(1, "Jim");  
    Account acc2 = new Account(2, "Jeremy");  
    System.out.println(acc1.id != acc2.id);  
}
```

But, managing the unique id's *manually* is **error-prone**!

Static Variables (2)

```
class Account {  
    static int globalCounter = 1;  
    int id; String owner;  
    Account(String owner) {  
        this.id = globalCounter; globalCounter++;  
        this.owner = owner; } }  
}
```

```
class AccountTester {  
    Account acc1 = new Account("Jim");  
    Account acc2 = new Account("Jeremy");  
    System.out.println(acc1.id != acc2.id); }  
}
```

- Each instance of a class (e.g., acc1, acc2) has a *local* copy of each attribute or instance variable (e.g., id).
 - Changing acc1.id does not affect acc2.id.
- A **static** variable (e.g., globalCounter) belongs to the class.
 - All instances of the class share a *single* copy of the **static** variable.
 - Change to globalCounter via c1 is also visible to c2.

Static Variables (3)

```
class Account {  
    static int globalCounter = 1;  
    int id; String owner;  
    Account(String owner) {  
        this.id = globalCounter;  
        globalCounter++;  
        this.owner = owner;  
    }  
}
```

- **Static** variable `globalCounter` is not instance-specific like **instance** variable (i.e., attribute) `id` is.
- To access a **static** variable:
 - **No** context object is needed.
 - Use of the class name suffices, e.g., `Account.globalCounter`.
- Each time `Account`'s constructor is called to create a new instance, the increment effect is **visible to all existing objects** of `Account`.

Static Variables (4.1): Common Error

```
class Client {  
    Account[] accounts;  
    static int numberOfAccounts = 0;  
    void addAccount(Account acc) {  
        accounts[numberOfAccounts] = acc;  
        numberOfAccounts ++;  
    }  
}
```

```
class ClientTester {  
    Client bill = new Client("Bill");  
    Client steve = new Client("Steve");  
    Account acc1 = new Account();  
    Account acc2 = new Account();  
    bill.addAccount(acc1);  
    /* correctly added to bill.accounts[0] */  
    steve.addAccount(acc2);  
    /* mistakenly added to steve.accounts[1]! */  
}
```

Static Variables (4.2): Common Error

- Attribute `numberOfAccounts` should **not** be declared as `static` as its value should be specific to the client object.
- If it were declared as `static`, then every time the `addAccount` method is called, although on different objects, the increment effect of `numberOfAccounts` will be visible to all `Client` objects.
- Here is the correct version:

```
class Client {  
    Account[] accounts;  
    int numberOfAccounts = 0;  
    void addAccount(Account acc) {  
        accounts[numberOfAccounts] = acc;  
        numberOfAccounts ++;  
    }  
}
```

Static Variables (5.1): Common Error

```
1 public class Bank {  
2     public string branchName;  
3     public static int nextAccountNumber = 1;  
4     public static void useAccountNumber() {  
5         System.out.println (branchName + ...);  
6         nextAccountNumber ++;  
7     }  
8 }
```

- *Non-static method cannot be referenced from a static context*
- **Line 4** declares that we **can** call the method `useAccountNumber` without instantiating an object of the class `Bank`.
- However, in **Lined 5**, the *static* method references a *non-static* attribute, for which we **must** instantiate a `Bank` object.

Static Variables (5.2): Common Error

```
1 public class Bank {  
2     public string branchName;  
3     public static int nextAccountNumber = 1;  
4     public static void useAccountNumber() {  
5         System.out.println (branchName + ...);  
6         nextAccountNumber ++;  
7     }  
8 }
```

- To call `useAccountNumber()`, no instances of `Bank` are required:

```
Bank.useAccountNumber();
```

- *Contradictorily*, to access `branchName`, a *context object* is required:

```
Bank b1 = new Bank(); b1.setBranch("Songdo IBK");  
System.out.println(b1.branchName);
```

Static Variables (5.3): Common Error

There are two possible ways to fix:

1. Remove all uses of *non-static* variables (i.e., `branchName`) in the *static* method (i.e., `useAccountNumber`).
2. Declare `branchName` as a *static* variable.
 - This does not make sense.
 - ∴ `branchName` should be a value specific to each `Bank` instance.

OOP: Helper Methods (1)

- After you complete and test your program, feeling confident that it is *correct*, you may find that there are lots of *repetitions*.
- When similar fragments of code appear in your program, we say that your code “*smells*”!
- We may eliminate *repetitions* of your code by:
 - **Factoring out** recurring code fragments into a new method.
 - This new method is called a **helper method** :
 - You can replace every occurrence of the recurring code fragment by a **call** to this helper method, with appropriate argument values.
 - That is, we **reuse** the body implementation, rather than repeating it over and over again, of this helper method via calls to it.
- This process is called **refactoring** of your code:
Modify the code structure **without** compromising *correctness*.

OOP: Helper (Accessor) Methods (2.1)

```
class PersonCollector {
    Person[] ps;
    final int MAX = 100; /* max # of persons to be stored */
    int nop; /* number of persons */
    PersonCollector() {
        ps = new Person[MAX];
    }
    void addPerson(Person p) {
        ps[nop] = p;
        nop++;
    }
    /* Tasks:
     * 1. An accessor: boolean personExists(String n)
     * 2. A mutator: void changeWeightOf(String n, double w)
     * 3. A mutator: void changeHeightOf(String n, double h)
     */
}
```

OOP: Helper (Accessor) Methods (2.2.1)

```
class PersonCollector {
    /* ps, MAX, nop, PersonCollector(), addPerson */
    boolean personExists(String n) {
        boolean found = false;
        for(int i = 0; i < nop; i ++ ) {
            if(ps[i].name.equals(n)) { found = true; } }
        return found;
    }
    void changeWeightOf(String n, double w) {
        for(int i = 0; i < nop; i ++ ) {
            if(ps[i].name.equals(n)) { ps[i].setWeight(w); } }
    }
    void changeHeightOf(String n, double h) {
        for(int i = 0; i < nop; i ++ ) {
            if(ps[i].name.equals(n)) { ps[i].setHeight(h); } }
    }
}
```

OOP: Helper (Accessor) Methods (2.2.2)

```
class PersonCollector { /* code smells: repetitions! */
    /* ps, MAX, nop, PersonCollector(), addPerson */
    boolean personExists(String n) {
        boolean found = false;
        for(int i = 0; i < nop; i ++) {
            if(ps[i].name.equals(n)) { found = true; } }
        return found;
    }
    void changeWeightOf(String n, double w) {
        for(int i = 0; i < nop; i ++) {
            if(ps[i].name.equals(n)) { ps[i].setWeight(w); } }
    }
    void changeHeightOf(String n, double h) {
        for(int i = 0; i < nop; i ++) {
            if(ps[i].name.equals(n)) { ps[i].setHeight(h); } }
    }
}
```

OOP: Helper (Accessor) Methods (2.3)

```
class PersonCollector { /* Eliminate code smell. */
    /* ps, MAX, nop, PersonCollector(), addPerson */
    int indexOf (String n) { /* Helper Methods */
        int i = -1;
        for(int j = 0; j < nop; j++) {
            if(ps[j].name.equals(n)) { i = j; }
        }
        return i; /* -1 if not found; >= 0 if found. */
    }
    boolean personExists(String n) { return indexOf (n) >= 0; }
    void changeWeightOf(String n, double w) {
        int i = indexOf (n); if(i >= 0) { ps[i].setWeight(w); }
    }
    void changeHeightOf(String n, double h) {
        int i = indexOf (n); if(i >= 0) { ps[i].setHeight(h); }
    }
}
```

OOP: Helper (Accessor) Methods (3.1)

Problems:

- A `Point` class with `x` and `y` coordinate values.
- Accessor `double` `getDistanceFromOrigin()`.
`p.getDistanceFromOrigin()` returns the distance between `p` and `(0, 0)`.
- Accessor `double` `getDistancesTo(Point p1, Point p2)`.
`p.getDistancesTo(p1, p2)` returns the sum of distances between `p` and `p1`, and between `p` and `p2`.
- Accessor `double` `getTriDistances(Point p1, Point p2)`.
`p.getDistancesTo(p1, p2)` returns the sum of distances between `p` and `p1`, between `p` and `p2`, and between `p1` and `p2`.

OOP: Helper (Accessor) Methods (3.2)

```
class Point {
    double x; double y;
    double getDistanceFromOrigin() {
        return Math.sqrt(Math.pow(x - 0, 2) + Math.pow(y - 0, 2)); }
    double getDistancesTo(Point p1, Point p2) {
        return
            Math.sqrt(Math.pow(x - p1.x, 2) + Math.pow(y - p1.y, 2))
            +
            Math.sqrt(Math.pow(x - p2.x, 2) + Math.pow(y - p2.y, 2)); }
    double getTriDistances(Point p1, Point p2) {
        return
            Math.sqrt(Math.pow(x - p1.x, 2) + Math.pow(y - p1.y, 2))
            +
            Math.sqrt(Math.pow(x - p2.x, 2) + Math.pow(y - p2.y, 2))
            +
            Math.sqrt(Math.pow(p1.x - p2.x, 2)
                +
                Math.pow(p1.y - p2.y, 2));
    }
}
```

OOP: Helper (Accessor) Methods (3.3)

- The code pattern

```
Math.sqrt(Math.pow(... - ..., 2) + Math.pow(... - ..., 2))
```

is written down explicitly every time we need to use it.

- Create a **helper method** out of it, with the right *parameter* and *return* types:

```
double getDistanceFrom(double otherX, double otherY) {  
    return  
        Math.sqrt(Math.pow(otherX - this.x, 2)  
            +  
            Math.pow(otherY - this.y, 2));  
}
```

OOP: Helper (Accessor) Methods (3.4)

```
class Point {
    double x; double y;
    double getDistanceFrom(double otherX, double otherY) {
        return Math.sqrt(Math.pow(ohterX - this.x, 2) +
            Math.pow(otherY - this.y, 2));
    }
    double getDistanceFromOrigin() {
        return this.getDistanceFrom(0, 0);
    }
    double getDistancesTo(Point p1, Point p2) {
        return this.getDistanceFrom(p1.x, p1.y) +
            this.getDistanceFrom(p2.x, p2.y);
    }
    double getTriDistances(Point p1, Point p2) {
        return this.getDistanceFrom(p1.x, p1.y) +
            this.getDistanceFrom(p2.x, p2.y) +
            p1.getDistanceFrom(p2.x, p2.y)
    }
}
```

OOP: Helper (Mutator) Methods (4.1)

```
class Student {
    String name;
    double balance;
    Student(String n, double b) {
        name = n;
        balance = b;
    }

    /* Tasks:
     * 1. A mutator void receiveScholarship(double val)
     * 2. A mutator void payLibraryOverdue(double val)
     * 3. A mutator void payCafeCoupons(double val)
     * 4. A mutator void transfer(Student other, double val)
     */
}
```

OOP: Helper (Mutator) Methods (4.2.1)

```
class Student {  
    /* name, balance, Student(String n, double b) */  
    void receiveScholarship(double val) {  
        balance = balance + val;  
    }  
    void payLibraryOverdue(double val) {  
        balance = balance - val;  
    }  
    void payCafeCoupons(double val) {  
        balance = balance - val;  
    }  
    void transfer(Student other, double val) {  
        balance = balance - val;  
        other.balance = other.balance + val;  
    }  
}
```

OOP: Helper (Mutator) Methods (4.2.2)

```
class Student { /* code smells: repetitions! */
  /* name, balance, Student(String n, double b) */
  void receiveScholarship(double val) {
    balance = balance + val;
  }
  void payLibraryOverdue(double val) {
    balance = balance - val;
  }
  void payCafeCoupons(double val) {
    balance = balance - val;
  }
  void transfer(Student other, double val) {
    balance = balance - val;
    balance = other.balance + val;
  }
}
```

OOP: Helper (Mutator) Methods (4.3)

```
class Student { /* Eliminate code smell. */
    /* name, balance, Student(String n, double b) */
    void deposit(double val) { /* Helper Method */
        balance = balance + val;
    }
    void withdraw(double val) { /* Helper Method */
        balance = balance - val;
    }
    void receiveScholarship(double val) { this.deposit(val); }
    void payLibraryOverdue(double val) { this.withdraw(val); }
    void payCafeCoupons(double val) { this.withdraw(val) }
    void transfer(Student other, double val) {
        this.withdraw(val);
        other.deposit(val);
    }
}
```

Index (1)

Separation of Concerns: App vs. Model

Object Orientation:

Observe, Model, and Execute

Object-Oriented Programming (OOP)

OO Thinking: Templates vs. Instances (1.1)

OO Thinking: Templates vs. Instances (1.2)

OO Thinking: Templates vs. Instances (2.1)

OO Thinking: Templates vs. Instances (2.2)

OO Thinking: Templates vs. Instances (3)

OOP: Classes \approx Templates

OOP:

Define Constructors for Creating Objects (1.1)

OOP:

Define Constructors for Creating Objects (1.2)

The `this` Reference (1)

Index (2)

The `this` Reference (2)

The `this` Reference (3)

The `this` Reference (4)

The `this` Reference (5)

The `this` Reference (6.1): Common Error

The `this` Reference (6.2): Common Error

OOP:

Define Constructors for Creating Objects (2.1)

OOP:

Define Constructors for Creating Objects (2.2)

OOP: Methods (1.1)

OOP: Methods (1.2)

OOP: Methods (2)

OOP: Methods (3)

Index (3)

OOP: The Dot Notation (1)

OOP: The Dot Notation (2)

OOP: Method Calls

OOP: Class Constructors (1)

OOP: Class Constructors (2)

OOP: Class Constructors (3)

OOP: Class Constructors (4)

OOP: Object Creation (1)

OOP: Object Creation (2)

OOP: Object Creation (3)

OOP: Object Creation (4)

OOP: Object Creation (5)

OOP: Object Creation (6)

OOP: Mutator Methods

Index (4)

OOP: Accessor Methods

OOP: Use of Mutator vs. Accessor Methods

OOP: Method Parameters

The `this` Reference (7.1): Exercise

The `this` Reference (7.2): Exercise

Java Data Types (1)

Java Data Types (2)

Java Data Types (3.1)

Java Data Types (3.2.1)

Java Data Types (3.2.2)

Java Data Types (3.3.1)

Java Data Types (3.3.2)

OOP: Object Alias (1)

OOP: Object Alias (2.1)

Index (5)

OOP: Object Alias (2.2)

Call by Value vs. Call by Reference (1)

Call by Value vs. Call by Reference (2.1)

Call by Value vs. Call by Reference (2.2.1)

Call by Value vs. Call by Reference (2.2.2)

Call by Value vs. Call by Reference (2.3.1)

Call by Value vs. Call by Reference (2.3.2)

Call by Value vs. Call by Reference (2.4.1)

Call by Value vs. Call by Reference (2.4.2)

Aggregation vs. Composition: Terminology

Aggregation: Independent Containees

Shared by Containers (1.1)

Aggregation: Independent Containees

Shared by Containers (1.2)

Index (6)

**Aggregation: Independent Containees
Shared by Containers (2.1)**

**Aggregation: Independent Containees
Shared by Containers (2.2)**

OOP: The Dot Notation (3.1)

OOP: The Dot Notation (3.2)

OOP: The Dot Notation (3.3)

OOP: The Dot Notation (3.4)

**Composition: Dependent Containees
Owned by Containers (1.1)**

**Composition: Dependent Containees
Owned by Containers (1.2.1)**

**Composition: Dependent Containees
Owned by Containers (1.2.2)**

Index (7)

**Composition: Dependent Containees
Owned by Containers (1.3)**

**Composition: Dependent Containees
Owned by Containers (1.4.1)**

**Composition: Dependent Containees
Owned by Containers (1.4.2)**

**Composition: Dependent Containees
Owned by Containers (1.5.1)**

**Composition: Dependent Containees
Owned by Containers (1.5.2)**

**Composition: Dependent Containees
Owned by Containers (1.6)**

Aggregation vs. Composition (1)

Aggregation vs. Composition (2)

OOP: Equality (1)

Index (8)

OOP: Equality (2)

OOP: Equality (3)

OOP: Contract of equals

OOP: Equality (4.1)

OOP: Equality (4.2)

OOP: Equality (4.3)

OOP: Equality (5.1)

OOP: Equality (5.2)

OOP: Equality (5.3)

OOP: Equality (6)

Hashing: What is a Map?

Hashing: Arrays are Maps

Hashing: Naive Implementation of Map

Hashing: Naive Implementation of Map (0)

Index (9)

Hashing: Naive Implementation of Map (1)

Hashing: Naive Implementation of Map (2)

Hashing: Naive Implementation of Map (3)

Hashing: Naive Implementation of Map (4)

Hashing: Naive Implementation of Map (5)

Hashing: Naive Implementation of Map (6)

Hashing: Naive Implementation of Map (7)

Hashing: Naive Implementation of Map (8.1)

Hashing: Naive Implementation of Map (8.2)

Hashing: Naive Implementation of Map (8.3)

Hashing: Naive Implementation of Map (8.4)

Hashing: Hash Table (1)

Hashing: Hash Table as a Bucket Array (2)

Hashing: Contract of Hash Function

Index (10)

Hashing: Defining Hash Function in Java (1)

Hashing: Defining Hash Function in Java (2)

Hashing: Using Hash Table in Java

Hashing: Defining Hash Function in Java (3)

Hashing: Defining Hash Function in Java (4.1)

Hashing: Defining Hash Function in Java (4.2)

Why Ordering Between Objects? (1)

Why Ordering Between Objects? (2)

Defining Ordering Between Objects (1.1)

Defining Ordering Between Objects (1.2)

Defining Ordering Between Objects (2.1)

Defining Ordering Between Objects (2.2)

Defining Ordering Between Objects (2.3)

Defining Ordering Between Objects (3)

Index (11)

Static Variables (1)

Static Variables (2)

Static Variables (3)

Static Variables (4.1): Common Error

Static Variables (4.2): Common Error

Static Variables (5.1): Common Error

Static Variables (5.2): Common Error

Static Variables (5.3): Common Error

OOP: Helper Methods (1)

OOP: Helper (Accessor) Methods (2.1)

OOP: Helper (Accessor) Methods (2.2.1)

OOP: Helper (Accessor) Methods (2.2.2)

OOP: Helper (Accessor) Methods (2.3)

OOP: Helper (Accessor) Methods (3.1)

Index (12)

OOP: Helper (Accessor) Methods (3.2)

OOP: Helper (Accessor) Methods (3.3)

OOP: Helper (Accessor) Methods (3.4)

OOP: Helper (Mutator) Methods (4.1)

OOP: Helper (Mutator) Methods (4.2.1)

OOP: Helper (Mutator) Methods (4.2.2)

OOP: Helper (Mutator) Methods (4.3)

Asymptotic Analysis of Algorithms



EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Algorithm and Data Structure

- A **data structure** is:
 - A systematic way to store and organize data in order to facilitate *access* and *modifications*
 - Never suitable for all purposes: it is important to know its *strengths* and *limitations*
- A **well-specified computational problem** precisely describes the desired *input/output relationship*.
 - **Input:** A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$
 - **Output:** A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$
 - An *instance* of the problem: $\langle 3, 1, 2, 5, 4 \rangle$
- An **algorithm** is:
 - A solution to a well-specified *computational problem*
 - A *sequence of computational steps* that takes value(s) as *input* and produces value(s) as *output*
- Steps in an *algorithm* manipulate well-chosen *data structure(s)*.

Measuring “Goodness” of an Algorithm

1. *Correctness* :

- Does the algorithm produce the expected output?
- Use JUnit to ensure this.

2. Efficiency:

- *Time Complexity*: processor time required to complete
- *Space Complexity*: memory space required to store data

Correctness is always the priority.

How about efficiency? Is time or space more of a concern?

Measuring Efficiency of an Algorithm

- *Time* is more of a concern than is *storage*.
- Solutions that are meant to be run on a computer should run *as fast as possible*.
- Particularly, we are interested in how *running time* depends on two *input factors*:
 1. size
e.g., sorting an array of 10 elements vs. 1m elements
 2. structure
e.g., sorting an already-sorted array vs. a hardly-sorted array
- *How do you determine the running time of an algorithm?*
 1. Measure time via *experiments*
 2. Characterize time as a *mathematical function* of the input size

Measure Running Time via Experiments

- Once the algorithm is implemented in Java:
 - Execute the program on *test inputs* of various *sizes* and *structures*.
 - For each test, record the *elapsed time* of the execution.

```
long startTime = System.currentTimeMillis();  
/* run the algorithm */  
long endTime = System.currenctTimeMillis();  
long elapsed = endTime - startTime;
```

- *Visualize* the result of each test.
- To make **sound statistical claims** about the algorithm's *running time*, the set of input tests must be “reasonably” *complete*.

Example Experiment

- *Computational Problem:*
 - **Input:** A character c and an integer n
 - **Output:** A string consisting of n repetitions of character c
e.g., Given input `'*` and 15, output `*****`.
- *Algorithm 1* using *String* Concatenations:

```
public static String repeat1(char c, int n) {  
    String answer = "";  
    for (int i = 0; i < n; i++) { answer += c; }  
    return answer; }
```

- *Algorithm 2* using *StringBuilder* append's:

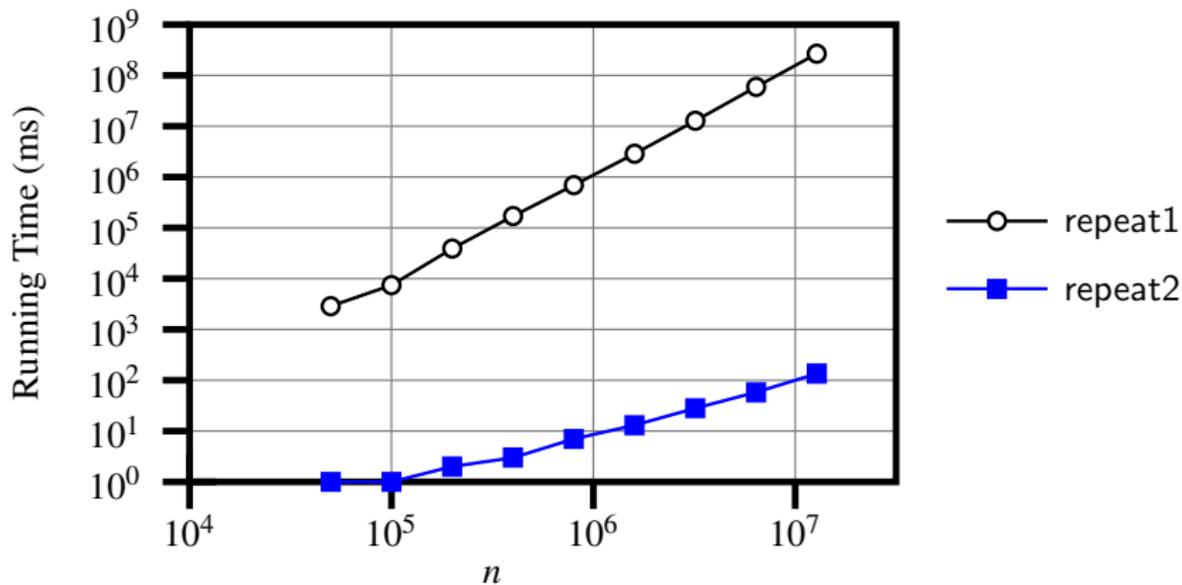
```
public static String repeat2(char c, int n) {  
    StringBuilder sb = new StringBuilder();  
    for (int i = 0; i < n; i++) { sb.append(c); }  
    return sb.toString(); }
```

Example Experiment: Detailed Statistics

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,847,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421 (\approx 3 days)	135

- As *input size* is doubled, **rates of increase** for both algorithms are *linear*:
 - Running time* of repeat1 increases by \approx 5 times.
 - Running time* of repeat2 increases by \approx 2 times.

Example Experiment: Visualization



Experimental Analysis: Challenges

1. An algorithm must be *fully implemented* (i.e., translated into valid Java syntax) in order to study its runtime behaviour *experimentally*.
 - What if our purpose is to *choose among alternative* data structures or algorithms to implement?
 - Can there be a **higher-level analysis** to determine that one algorithm or data structure is *superior* than others?
2. Comparison of multiple algorithms is only *meaningful* when experiments are conducted under the same environment of:
 - *Hardware*: CPU, running processes
 - *Software*: OS, JVM version
3. Experiments can be done only on *a limited set of test inputs*.
 - What if “*important*” inputs were not included in the experiments?

Moving Beyond Experimental Analysis

- A better approach to analyzing the *efficiency* (e.g., *running times*) of algorithms should be one that:
 - Allows us to calculate the *relative efficiency* (rather than absolute elapsed time) of algorithms in a ways that is *independent of* the hardware and software environment.
 - Can be applied using a *high-level description* of the algorithm (without fully implementing it).
 - Considers *all* possible inputs.
- We will learn a better approach that contains 3 ingredients:
 1. Counting *primitive operations*
 2. Approximating running time as *a function of input size*
 3. Focusing on the *worst-case* input (requiring the most running time)

Counting Primitive Operations

- A *primitive operation* corresponds to a low-level instruction with a *constant execution time*.
 - Assignment [e.g., `x = 5;`]
 - Indexing into an array [e.g., `a[i]`]
 - Arithmetic, relational, logical op. [e.g., `a + b`, `z > w`, `b1 && b2`]
 - Accessing a field of an object [e.g., `acc.balance`]
 - Returning from a method [e.g., `return result;`]
 - Why is a method call is in general *not* a primitive operation?
- The *number of primitive operations* required by an algorithm should be *proportional* to its *actual running time* on a specific environment: $RT = \sum_{i=1}^N t(i)$ [$N = \#$ of PO's]
 - Say c is the *absolute* time of executing a *primitive operation* on a specific computer platform.
 - $RT = \sum_{i=1}^N t(i) = c \times N \approx N$
 - ⇒ *approximate* # of primitive operations that its steps contain.

Example: Counting Primitive Operations

```
1  findMax (int[] a, int n) {  
2    currentMax = a[0];  
3    for (int i = 1; i < n; ) {  
4      if (a[i] > currentMax) {  
5        currentMax = a[i]; }  
6      i ++ }  
7    return currentMax; }
```

- # of times $i < n$ in **Line 3** is executed? $[n]$
- # of times the loop body (**Line 4** to **Line 6**) is executed? $[n - 1]$
- **Line 2:** 2 [1 indexing + 1 assignment]
 - **Line 3:** $n + 1$ [1 assignment + n comparisons]
 - **Line 4:** $(n - 1) \cdot 2$ [1 indexing + 1 comparison]
 - **Line 5:** $(n - 1) \cdot 2$ [1 indexing + 1 assignment]
 - **Line 6:** $(n - 1) \cdot 2$ [1 addition + 1 assignment]
 - **Line 7:** 1 [1 return]
 - **Total # of Primitive Operations:** $7n - 2$

Example: Approx. # of Primitive Operations

- Given # of primitive operations counted precisely as $7n^1 - 2$, we view it as

$$7 \cdot n - 2 \cdot n^0$$

- We say
 - n is the **highest power**
 - 7 and 2 are the **multiplicative constants**
 - 2 is the **lower term**
- When approximating a function (considering that input size may be very large):
 - Only** the **highest power** matters.
 - multiplicative constants** and **lower terms** can be dropped.

$\Rightarrow 7n - 2$ is approximately n

Exercise: Consider $7n + 2n \cdot \log n + 3n^2$:

- highest power?**
- multiplicative constants?**
- lower terms?**

$$[n^2]$$

$$[7, 2, 3]$$

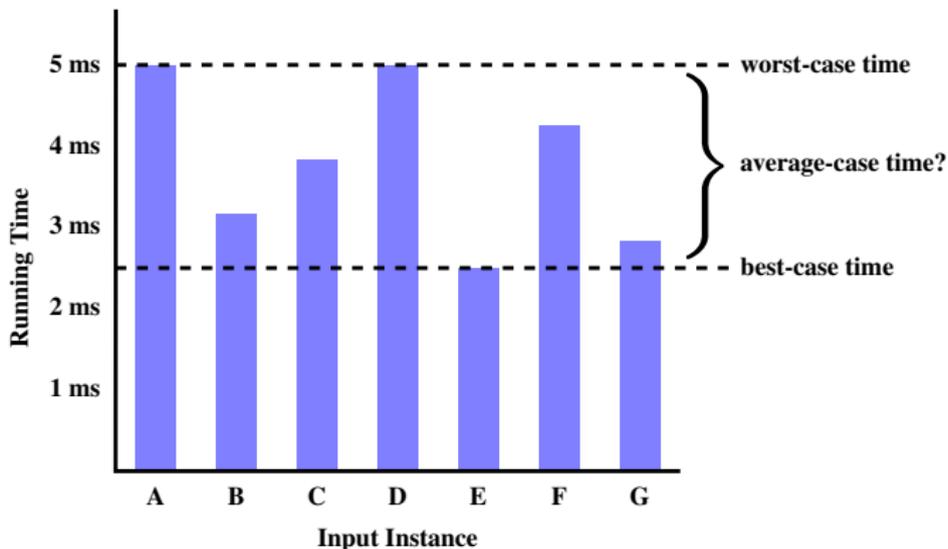
$$[7n + 2n \cdot \log n]$$

Approximating Running Time as a Function of Input Size

Given the *high-level description* of an algorithm, we associate it with a function f , such that $f(n)$ returns the *number of primitive operations* that are performed on an *input of size n* .

- $f(n) = 5$ [constant]
- $f(n) = \log_2 n$ [logarithmic]
- $f(n) = 4 \cdot n$ [linear]
- $f(n) = n^2$ [quadratic]
- $f(n) = n^3$ [cubic]
- $f(n) = 2^n$ [exponential]

Focusing on the Worst-Case Input



- *Average-case* analysis calculates the *expected running times* based on the probability distribution of input values.
- *worst-case* analysis or *best-case* analysis?

What is Asymptotic Analysis?

Asymptotic analysis

- Is a method of describing *behaviour in the limit*:
 - How the *running time* of the algorithm under analysis changes as the *input size* changes without bound
 - e.g., contrast $RT_1(n) = n$ with $RT_2(n) = n^2$
- Allows us to compare the *relative* performance of alternative algorithms:
 - For large enough inputs, the *multiplicative constants* and *lower-order* terms of an exact running time can be disregarded.
 - e.g., $RT_1(n) = 3n^2 + 7n + 18$ and $RT_2(n) = 100n^2 + 3n - 100$ are considered **equally efficient**, *asymptotically*.
 - e.g., $RT_1(n) = n^3 + 7n + 18$ is considered **less efficient** than $RT_2(n) = 100n^2 + 100n + 2000$, *asymptotically*.

Three Notions of Asymptotic Bounds

We may consider three kinds of *asymptotic bounds* for the *running time* of an algorithm:

- Asymptotic *upper* bound $[O]$
- Asymptotic lower bound $[\Omega]$
- Asymptotic tight bound $[\Theta]$

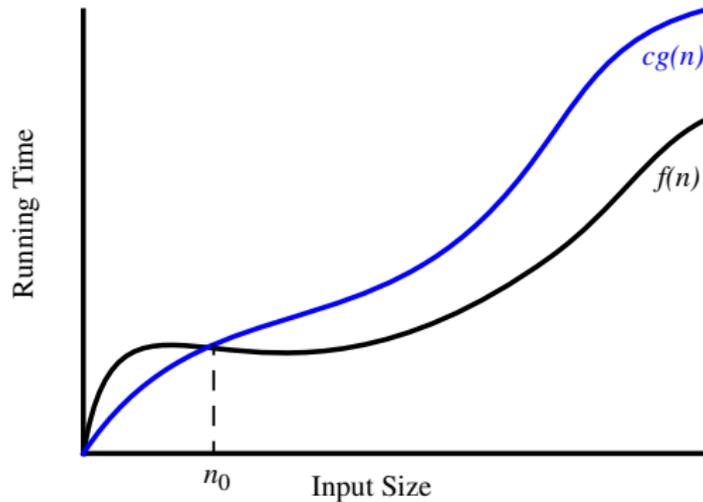
Asymptotic Upper Bound: Definition

- Let $f(n)$ and $g(n)$ be functions mapping positive integers (input size) to positive real numbers (running time).
 - $f(n)$ characterizes the running time of some algorithm.
 - $O(g(n))$ denotes *a collection of* functions.
- $O(g(n))$ consists of *all* functions that can be upper bounded by $g(n)$, starting at some point, using some constant factor.
- $f(n) \in O(g(n))$ if there are:
 - A real *constant* $c > 0$
 - An integer *constant* $n_0 \geq 1$
 such that:

$$f(n) \leq c \cdot g(n) \quad \text{for } n \geq n_0$$

- For each member function $f(n)$ in $O(g(n))$, we say that:
 - $f(n) \in O(g(n))$ [f(n) is a member of “big-Oh of g(n)”]
 - $f(n)$ **is** $O(g(n))$ [f(n) is “big-Oh of g(n)”]
 - $f(n)$ **is order of** $g(n)$

Asymptotic Upper Bound: Visualization



From n_0 , $f(n)$ is upper bounded by $c \cdot g(n)$, so $f(n)$ is $O(g(n))$.

Asymptotic Upper Bound: Example (1)

Prove: The function $8n + 5$ is $O(n)$.

Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$8n + 5 \leq c \cdot n$$

Can we choose $c = 9$? What should the corresponding n_0 be?

n	$8n + 5$	$9n$
1	13	9
2	21	18
3	29	27
4	37	36
5	45	45
6	53	54

...

Therefore, we prove it by choosing $c = 9$ and $n_0 = 5$.

We may also prove it by choosing $c = 13$ and $n_0 = 1$. Why?

Asymptotic Upper Bound: Example (2)

Prove: The function $f(n) = 5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.

Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq c \cdot n^4$$

$$f(1) = 5 + 3 + 2 + 4 + 1 = 15$$

Choose $c = 15$ and $n_0 = 1$!

Asymptotic Upper Bound: Proposition (1)

If $f(n)$ is a polynomial of degree d , i.e.,

$$f(n) = a_0 \cdot n^0 + a_1 \cdot n^1 + \cdots + a_d \cdot n^d$$

and a_0, a_1, \dots, a_d are integers (i.e., negative, zero, or positive), then $f(n)$ is $O(n^d)$.

Proof:

1. We know that for $n \geq 1$:

$$n^0 \leq n^1 \leq n^2 \leq \cdots \leq n^d$$

2. By choosing $c = |a_0| + |a_1| + \cdots + |a_d|$:

$$a_0 \cdot n^0 + a_1 \cdot n^1 + \cdots + a_d \cdot n^d \leq |a_0| \cdot n^d + |a_1| \cdot n^d + \cdots + |a_d| \cdot n^d$$

3. By choosing $n_0 = 1$:

$$a_0 \cdot 1^0 + a_1 \cdot 1^1 + \cdots + a_d \cdot 1^d \leq |a_0| \cdot 1^d + |a_1| \cdot 1^d + \cdots + |a_d| \cdot 1^d$$

That is, we prove by choosing

$$\begin{aligned} c &= |a_0| + |a_1| + \cdots + |a_d| \\ n_0 &= 1 \end{aligned}$$

Asymptotic Upper Bound: Proposition (2)

$$O(n^0) \subset O(n^1) \subset O(n^2) \subset \dots$$

If a function $f(n)$ is *upper bounded* by another function $g(n)$ of degree d , $d \geq 0$, then $f(n)$ is also upper bounded by all other functions of a *strictly higher degree* (i.e., $d + 1$, $d + 2$, etc.).

Asymptotic Upper Bound: More Examples

- $5n^2 + 3n \cdot \log n + 2n + 5$ is $O(n^2)$ [$c = 15, n_0 = 1$]
- $20n^3 + 10n \cdot \log n + 5$ is $O(n^3)$ [$c = 35, n_0 = 1$]
- $3 \cdot \log n + 2$ is $O(\log n)$ [$c = 5, n_0 = 2$]
 - Why can't n_0 be 1?
 - Choosing $n_0 = 1$ means $\Rightarrow f(1)$ **is** upper-bounded by $c \cdot \log 1$:
 - We have $f(1) = 3 \cdot \log 1 + 2$, which is 2.
 - We have $c \cdot \log 1$, which is 0. $\Rightarrow f(1)$ **is not** upper-bounded by $c \cdot \log 1$ [Contradiction!]
- 2^{n+2} is $O(2^n)$ [$c = 4, n_0 = 1$]
- $2n + 100 \cdot \log n$ is $O(n)$ [$c = 102, n_0 = 1$]

Using Asymptotic Upper Bound Accurately

- Use the big-Oh notation to characterize a function (of an algorithm's running time) *as closely as possible*.

For example, say $f(n) = 4n^3 + 3n^2 + 5$:

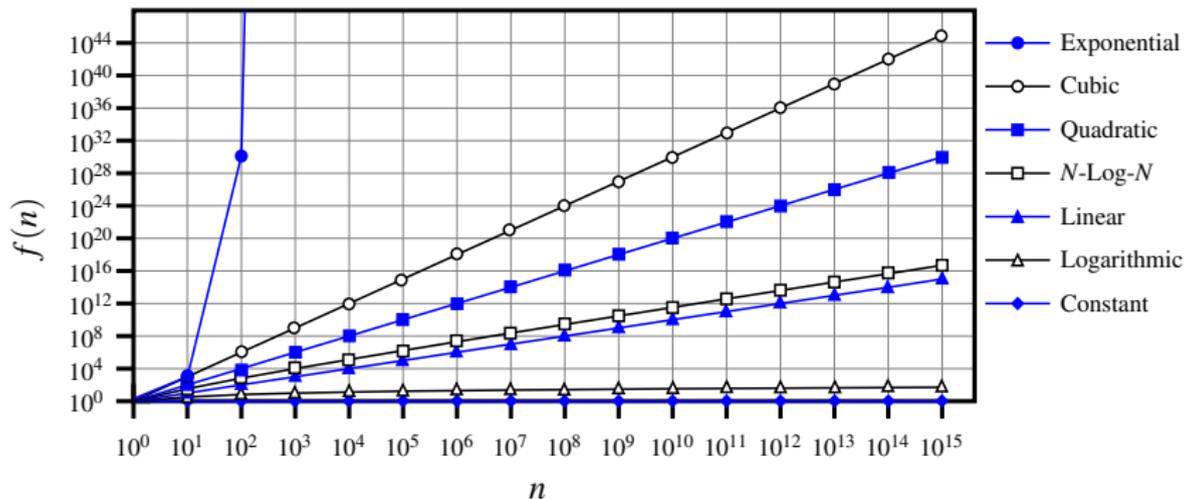
- Recall: $O(n^3) \subset O(n^4) \subset O(n^5) \subset \dots$
 - It is the *most accurate* to say that $f(n)$ is $O(n^3)$.
 - It is also true, but not very useful, to say that $f(n)$ is $O(n^4)$ and that $f(n)$ is $O(n^5)$.
- Do not include *constant factors* and *lower-order terms* in the big-Oh notation.

For example, say $f(n) = 2n^2$ is $O(n^2)$, do not say $f(n)$ is $O(4n^2 + 6n + 9)$.

Classes of Functions

upper bound	class	cost
$O(1)$	constant	<i>cheapest</i>
$O(\log(n))$	logarithmic	
$O(n)$	linear	
$O(n \cdot \log(n))$	"n-log-n"	
$O(n^2)$	quadratic	
$O(n^3)$	cubic	
$O(n^k), k \geq 1$	polynomial	
$O(a^n), a > 1$	exponential	<i>most expensive</i>

Rates of Growth: Comparison



Upper Bound of Algorithm: Example (1)

```
1  maxOf (int x, int y) {  
2      int max = x;  
3      if (y > x) {  
4          max = y;  
5      }  
6      return max;  
7  }
```

- # of primitive operations: 4
 2 assignments + 1 comparison + 1 return = 4
- Therefore, the running time is $O(1)$.
- That is, this is a *constant-time* algorithm.

Upper Bound of Algorithm: Example (2)

```
1 findMax (int[] a, int n) {  
2   currentMax = a[0];  
3   for (int i = 1; i < n; ) {  
4     if (a[i] > currentMax) {  
5       currentMax = a[i]; }  
6     i ++ }  
7   return currentMax; }
```

- From last lecture, we calculated that the # of primitive operations is $7n - 2$.
- Therefore, the running time is $O(n)$.
- That is, this is a *linear-time* algorithm.

Upper Bound of Algorithm: Example (3)

```
1 containsDuplicate (int[] a, int n) {  
2   for (int i = 0; i < n; ) {  
3     for (int j = 0; j < n; ) {  
4       if (i != j && a[i] == a[j]) {  
5         return true; }  
6       j ++; }  
7     i ++; }  
8   return false; }
```

- Worst case is when we reach Line 8.
- # of primitive operations $\approx c_1 + n \cdot n \cdot c_2$, where c_1 and c_2 are some constants.
- Therefore, the running time is $O(n^2)$.
- That is, this is a *quadratic* algorithm.

Upper Bound of Algorithm: Example (4)

```
1  sumMaxAndCrossProducts (int[] a, int n) {  
2    int max = a[0];  
3    for(int i = 1; i < n; ) {  
4      if (a[i] > max) { max = a[i]; }  
5    }  
6    int sum = max;  
7    for (int j = 0; j < n; j ++ ) {  
8      for (int k = 0; k < n; k ++ ) {  
9        sum += a[j] * a[k]; } }  
10   return sum; }
```

- # of primitive operations $\approx (c_1 \cdot n + c_2) + (c_3 \cdot n \cdot n + c_4)$, where c_1 , c_2 , c_3 , and c_4 are some constants.
- Therefore, the running time is $O(n + n^2) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

Upper Bound of Algorithm: Example (5)

```
1 triangularSum (int[] a, int n) {  
2   int sum = 0;  
3   for (int i = 0; i < n; i++) {  
4     for (int j = i; j < n; j++) {  
5       sum += a[j]; } }  
6   return sum; }
```

- # of primitive operations $\approx n + (n - 1) + \dots + 2 + 1 = \frac{n \cdot (n + 1)}{2}$
- Therefore, the running time is $O\left(\frac{n^2 + n}{2}\right) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

Index (1)

Algorithm and Data Structure

Measuring “Goodness” of an Algorithm

Measuring Efficiency of an Algorithm

Measure Running Time via Experiments

Example Experiment

Example Experiment: Detailed Statistics

Example Experiment: Visualization

Experimental Analysis: Challenges

Moving Beyond Experimental Analysis

Counting Primitive Operations

Example: Counting Primitive Operations

Example: Approx. # of Primitive Operations

Approximating Running Time

as a Function of Input Size

Index (2)

Focusing on the Worst-Case Input

What is Asymptotic Analysis?

Three Notions of Asymptotic Bounds

Asymptotic Upper Bound: Definition

Asymptotic Upper Bound: Visualization

Asymptotic Upper Bound: Example (1)

Asymptotic Upper Bound: Example (2)

Asymptotic Upper Bound: Proposition (1)

Asymptotic Upper Bound: Proposition (2)

Asymptotic Upper Bound: More Examples

Using Asymptotic Upper Bound Accurately

Classes of Functions

Rates of Growth: Comparison

Upper Bound of Algorithm: Example (1)

Index (3)

Upper Bound of Algorithm: Example (2)

Upper Bound of Algorithm: Example (3)

Upper Bound of Algorithm: Example (4)

Upper Bound of Algorithm: Example (5)

ADTs, Arrays, and Linked-Lists

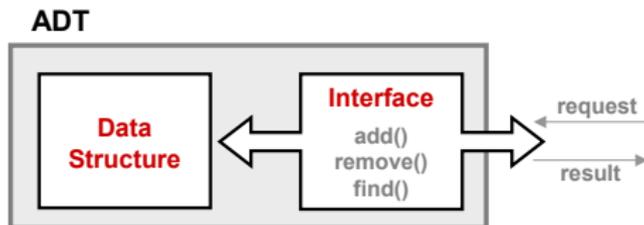


EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Abstract Data Types (ADTs)

- Given a problem, you are required to filter out *irrelevant* details.
- The result is an **abstract data type (ADT)**, whose *interface* consists of a list of (unimplemented) operations.



- Supplier's Obligations:**
 - Implement all operations
 - Choose the "right" data structure (DS)
- Client's Benefits:**
 - Correct output
 - Efficient performance
- The internal details of an *implemented ADT* should be **hidden**.

Standard ADTs

- *Standard* ADTs are **reusable components** that have been adopted in solving many real-world problems.
e.g., Stacks, Queues, Lists, Tables, Trees, Graphs
- You will be required to:
 - *Implement* standard ADTs
 - *Design* algorithms that make use of standard ADTs
- For each standard ADT, you are required to know:
 - The list of supported operations (i.e., **interface**)
 - Time (and sometimes space) **complexity** of each operation
- In this lecture, we learn about two *basic data structures*:
 - arrays
 - linked lists

Basic Data Structure: Arrays

- An array is a sequence of indexed elements.
- *Size* of an array is **fixed** at the time of its construction.
- Supported *operations* on an array:
 - *Accessing*: e.g., `int max = a[0];`
Time Complexity: $O(1)$ [constant operation]
 - *Updating*: e.g., `a[i] = a[i + 1];`
Time Complexity: $O(1)$ [constant operation]
 - *Inserting/Removing*:

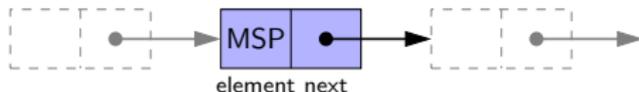
```
insertAt(String[] a, int n, String e, int i)
    String[] result = new String[n + 1];
    for(int j = 0; j < i; j++){ result[j] = a[j]; }
    result[i] = e;
    for(int j = i + 1; j < n; j++){ result[j] = a[j - 1]; }
    return result;
```

Time Complexity: $O(n)$ [linear operation]

Basic Data Structure: Singly-Linked Lists

- We know that **arrays** perform:
 - *well* in indexing
 - *badly* in inserting and deleting
- We now introduce an alternative data structure to arrays.
- A **linked list** is a series of connected *nodes* that collectively form a *linear sequence*.
- Each node in a **singly-linked** list has:
 - A *reference* to an *element of the sequence*
 - A *reference* to the *next node* in the list

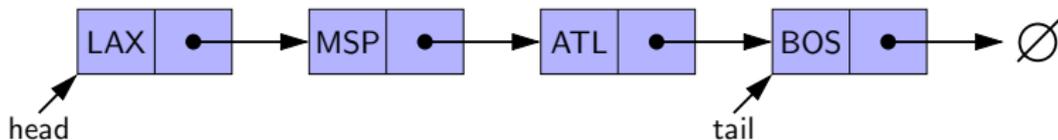
Contrast this *relative* positioning with the *absolute* indexing of arrays.



- The *last element* in a *singly-linked* list is different from others. How so? Its reference to the next node is simply `null`.

Singly-Linked List: How to Keep Track?

- Due to its “chained” structure, we can use a singly-linked list to *dynamically* store as many elements as we desire.
 - By creating a *new node* and setting the relevant *references*.
 - e.g., inserting an element to the beginning/middle/end of a list
 - e.g., deleting an element from the list requires a similar procedure
- **Contrary to the case of arrays**, we simply *cannot* keep track of all nodes in a linked list *directly* by indexing the *next* references.
- Instead, we only store a reference to the *head* (i.e., *first node*), and find other parts of the list *indirectly*.



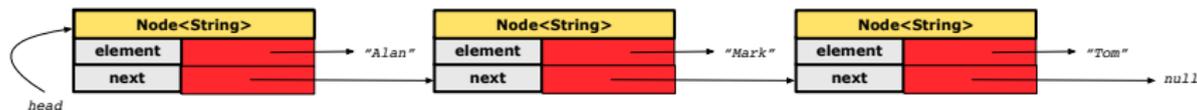
- **Exercise:** Given the *head* reference of a singly-linked list:
 - Count the number of nodes currently in the list [Running Time?]
 - Find the reference to its *tail* (i.e., last element) [Running Time?]

Singly-Linked List: Java Implementation

```
public class Node {  
    private String element;  
    private Node next;  
    public Node(String e, Node n) { element = e; next = n; }  
    public String getElement() { return element; }  
    public void setElement(String e) { element = e; }  
    public Node getNext() { return next; }  
    public void setNext(Node n) { next = n; }  
}
```

```
public class SinglyLinkedList {  
    private Node head = null;  
    public void addFirst(String e) { ... }  
    public void removeLast() { ... }  
    public void addAt(int i, String e) { ... }  
}
```

Singly-Linked List: A Running Example



Approach 1

```
Node tom = new Node("Tom", null);  
Node mark = new Node("Mark", tom);  
Node alan = new Node("Alan", mark);
```

Approach 2

```
Node alan = new Node("Alan", null);  
Node mark = new Node("Mark", null);  
Node tom = new Node("Tom", null);  
alan.setNext(mark);  
mark.setNext(tom);
```

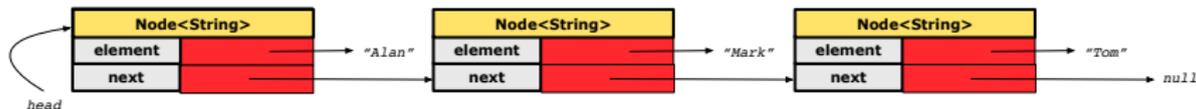
Singly-Linked List: Counting # of Nodes (1)

- Assume we are in the context of class `SinglyLinkedList`.

```
1  int getSize() {  
2      int size = 0;  
3      Node current = head;  
4      while (current != null) {  
5          /* exit when current == null */  
6          current = current.getNext();  
7          size ++;  
8      }  
9      return size;  
10 }
```

- When does the *while loop* (Line 4) terminate? `current` is null
- Only the *last node* has a null *next* reference.
- RT of `getSize` $O(n)$ [linear operation]
- Contrast:** RT of `a.length` is $O(1)$ [constant]

Singly-Linked List: Counting # of Nodes (2)



```
1 int getSize() {  
2     int size = 0;  
3     Node current = head;  
4     while (current != null) { /* exit when current == null */  
5         current = current.getNext();  
6         size ++;  
7     }  
8     return size;  
9 }
```

current	current != null	Beginning of Iteration	size
Alan	<i>true</i>	1	1
Mark	<i>true</i>	2	2
Tom	<i>true</i>	3	3
null	<i>false</i>	—	—

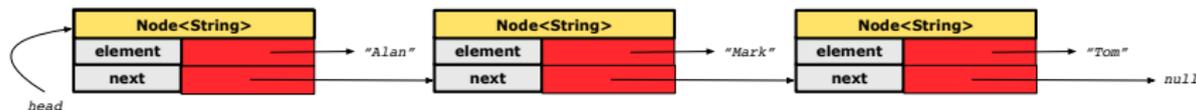
Singly-Linked List: Finding the Tail (1)

- Assume we are in the context of class `SinglyLinkedList`.

```
1  Node getTail() {  
2      Node current = head;  
3      Node tail = null;  
4      while (current != null) {  
5          /* exit when current == null */  
6          tail = current;  
7          current = current.getNext();  
8      }  
9      return tail;  
10 }
```

- When does the *while loop* (Line 4) terminate? `current` is null
- Only the *last node* has a null *next* reference.
- RT of `getTail` is $O(n)$ [linear operation]
- Contrast:** RT of `a[a.length - 1]` is $O(1)$ [constant]

Singly-Linked List: Finding the Tail (2)



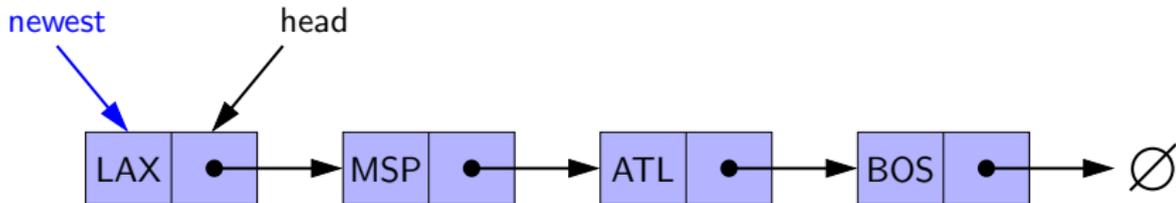
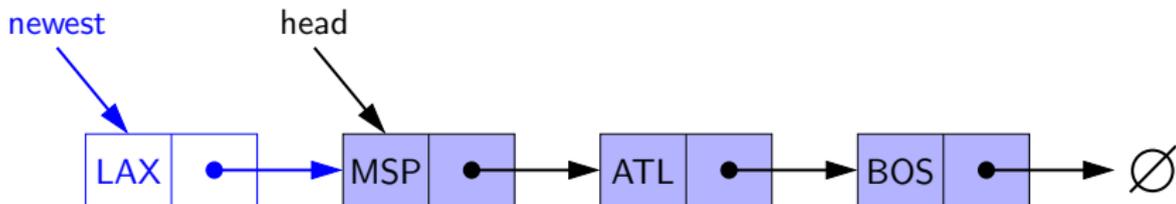
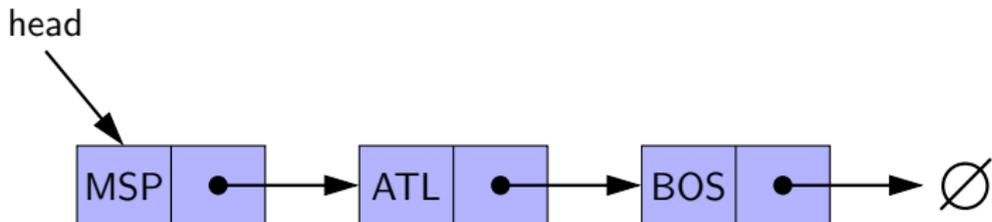
```
1 Node getTail() {  
2     Node current = head;  
3     Node tail = null;  
4     while (current != null) { /* exit when current == null */  
5         tail = current;  
6         current = current.getNext();  
7     }  
8     return tail;  
9 }
```

current	current != null	Beginning of Iteration	tail
Alan	true	1	Alan
Mark	true	2	Mark
Tom	true	3	Tom
null	false	—	—

Singly-Linked List: Can We Do Better?

- It is frequently needed to
 - access the **tail** of list [e.g., a new customer joins service queue]
 - query about its **size** [e.g., is the service queue full?]
- How can we improve the *running time* of these two operations?
- We may trade *space* for *time*.
- In addition to **head**, we also declare:
 - A variable **tail** that points to the end of the list
 - A variable **size** that keeps tracks of the number of nodes in list
 - Running time of these operations are both **$O(1)$** !
- Nonetheless, we cannot declare variables to store references to **nodes in-between** the head and tail. Why?
 - At the *time of declarations*, we simply do not know how many nodes there will be at *runtime*.

Singly-Linked List: Inserting to the Front (1)



Singly-Linked List: Inserting to the Front (2)

- Assume we are in the context of class `SinglyLinkedList`.

```
1 void addFirst (String e) {  
2     head = new Node(e, head);  
3     if (size == 0) {  
4         tail = head;  
5     }  
6     size ++;  
7 }
```

- Remember that RT of accessing *head* or *tail* is $O(1)$
- RT of `addFirst` is $O(1)$ [constant operation]
- **Contrast:** RT of inserting into an array is $O(n)$ [linear]

Your Homework

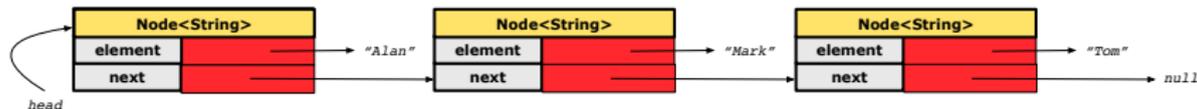
- Complete the Java *implementations* and *running time analysis* for `removeFirst()`, `addLast(E e)`.
- **Question:** *The `removeLast()` method may not be completed in the same way as is `addLast(String e)`. Why?*

Singly-Linked List: Accessing the Middle (1)

- Assume we are in the context of class `SinglyLinkedList`.

```
1  Node getNodeAt (int i) {
2      if (i < 0 || i >= size) {
3          throw IllegalArgumentException("Invalid Index");
4      }
5      else {
6          int index = 0;
7          Node current = head;
8          while (index < i) { /* exit when index == i */
9              index ++;
10             /* current is set to node at index i
11              * last iteration: index incremented from i - 1 to i
12              */
13             current = current.getNext();
14         }
15         return current;
16     }
17 }
```

Singly-Linked List: Accessing the Middle (2)



```
1 Node getNodeAt (int i) {  
2   if (i < 0 || i >= size) { /* print error */ }  
3   else {  
4     int index = 0;  
5     Node current = head;  
6     while (index < i) { /* exit when index == i */  
7       index ++;  
8       current = current.getNext();  
9     }  
10    return current;  
11  }  
12 }
```

Let's now consider `list.getNodeAt (2)` :

current	index	index < 2	Beginning of Iteration
Alan	0	<i>true</i>	1
Mark	1	<i>true</i>	2
Tom	2	<i>false</i>	-

Singly-Linked List: Accessing the Middle (3)

- What is the *worst case* of the index i for `getNodeAt(i)` ?
- Worst case: `list.getNodeAt(list.size - 1)`
- RT of `getNodeAt` is $O(n)$ [linear operation]
- **Contrast:** RT of accessing an array element is $O(1)$ [constant]

Singly-Linked List: Inserting to the Middle (1)

- Assume we are in the context of class `SinglyLinkedList`.

```
1 void addAt (int i, String e) {
2     if (i < 0 || i >= size) {
3         throw IllegalArgumentException("Invalid Index.");
4     }
5     else {
6         if (i == 0) {
7             addFirst(e);
8         }
9         else {
10            Node nodeBefore = getNodeAt(i - 1);
11            newNode = new Node(e, nodeBefore.getNext());
12            nodeBefore.setNext(newNode);
13            size ++;
14        }
15    }
16 }
```

Singly-Linked List: Inserting to the Middle (2)

- A call to `addAt(i, e)` may end up executing:
 - Line 3 (throw exception) $[O(1)]$
 - Line 7 (`addFirst`) $[O(1)]$
 - Lines 10 (`getNodeAt`) $[O(n)]$
 - Lines 11 – 13 (setting references) $[O(1)]$
- What is the *worst case* of the index `i` for `addAt(i, e)`?
- Worst case: `list.addAt(list.getSize() - 1, e)`
- RT of `addAt` is $O(n)$ [linear operation]
- **Contrast:** RT of inserting into an array is $O(n)$ [linear]
- On the other hand, for arrays, when given the *index* to an element, the RT of inserting an element is always $O(n)$!

Singly-Linked List: Removing from the End

- Assume we are in the context of class `SinglyLinkedList`.

```
1 void removeLast () {
2     if (size == 0) {
3         System.err.println("Empty List.");
4     }
5     else if (size == 1) {
6         removeFirst();
7     }
8     else {
9         Node secondLastNode = getNodeAt(size - 2);
10        secondLastNode.setNext(null);
11        tail = secondLastNode;
12        size --;
13    }
14 }
```

Running time? $O(n)$

Singly-Linked List: Exercises

Consider the following two linked-list operations, where a *reference node* is given as an input parameter:

- `void insertAfter(Node n, String e)`

- Steps?

- *Create a new node nn.*
- *Set nn's next to n's next.*
- *Set n's next to nn.*

- Running time?

[$O(1)$]

- `void insertBefore(Node n, String e)`

- Steps?

- *Iterate from the head, until `current.next == n`.*
- *Create a new node nn.*
- *Set nn's next to current's next (which is n).*
- *Set current's next to nn.*

- Running time?

[$O(n)$]

Your Homework

- Complete the Java *implementation* and *running time analysis* for `removeAt(int i)`.

Arrays vs. Singly-Linked Lists

DATA STRUCTURE		ARRAY	SINGLY-LINKED LIST
OPERATION			
get size			$O(1)$
get first/last element			
get element at index i		$O(1)$	$O(n)$
remove last element			
add/remove first element, add last element			
add/remove j^{th} element	given reference to $(i - 1)^{\text{th}}$ element	$O(n)$	$O(1)$
	not given		$O(n)$

Index (1)

Abstract Data Types (ADTs)

Standard ADTs

Basic Data Structure: Arrays

Basic Data Structure: Singly-Linked Lists

Singly-Linked List: How to Keep Track?

Singly-Linked List: Java Implementation

Singly-Linked List: A Running Example

Singly-Linked List: Counting # of Nodes (1)

Singly-Linked List: Counting # of Nodes (2)

Singly-Linked List: Finding the Tail (1)

Singly-Linked List: Finding the Tail (2)

Singly-Linked List: Can We Do Better?

Singly-Linked List: Inserting to the Front (1)

Singly-Linked List: Inserting to the Front (2)

Index (2)

Your Homework

Singly-Linked List: Accessing the Middle (1)

Singly-Linked List: Accessing the Middle (2)

Singly-Linked List: Accessing the Middle (3)

Singly-Linked List: Inserting to the Middle (1)

Singly-Linked List: Inserting to the Middle (2)

Singly-Linked List: Removing from the End

Singly-Linked List: Exercises

Your Homework

Arrays vs. Singly-Linked Lists

Stacks and Queues



EECS2030: Advanced
Object Oriented Programming
Fall 2017

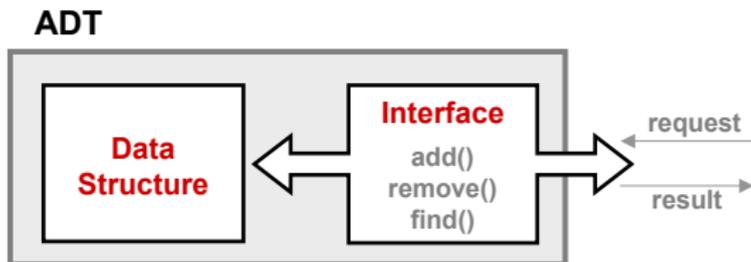
CHEN-WEI WANG

What is a Stack?

- A **stack** is a collection of objects.
- Objects in a **stack** are inserted and removed according to the **last-in, first-out (LIFO)** principle.
 - **Cannot** access **arbitrary** elements of a stack
 - **Can** only access or remove the **most-recently inserted** element



The Stack ADT



- **Accessors**
 - *top*
 - *size*
 - *isEmpty*
- **Mutators**
 - *push*
 - *pop*

Stack: Illustration

OPERATION	RETURN VALUE	STACK CONTENTS
-	-	∅
isEmpty	<i>true</i>	∅
push(5)	-	5
push(3)	-	$\frac{3}{5}$
push(1)	-	$\frac{1}{\frac{3}{5}}$
size	3	$\frac{1}{\frac{10}{5}}$
top	1	$\frac{1}{\frac{3}{5}}$
pop	1	$\frac{3}{5}$
pop	3	5
pop	5	∅

Implementing Stack ADT: Array (1)

```
public class ArrayedStack {
    private static final int MAX_CAPACITY = 1000;
    private String[] data;
    private int t; /* top index */
    public ArrayedStack() {
        data = new String[MAX_CAPACITY];
        t = -1; }
    public int size() { return (t + 1); }
    public boolean isEmpty() { return (t == -1); }
    public String top() {
        if (isEmpty()) { /* Error: Empty Stack. */ }
        else { return data[t]; } }
    public void push(String e) {
        if (size() == MAX_CAPACITY) { /* Error: Stack Full. */ }
        else { t++; data[t] = e; } }
    public String pop() {
        String result;
        if (isEmpty()) { /* Error: Empty Stack */ }
        else { result = data[t]; data[t] = null; t--; }
        return result; }
}
```

Implementing Stack ADT: Array (2)

```
@Test
public void testArrayedStack() {
    ArrayedStack s = new ArrayedStack();
    assertTrue(s.size() == 0 && s.isEmpty());
    try { String top = s.top();
        fail("Empty stack should have caused an exception."); }
    catch (IllegalArgumentException e) { }
    s.push("Alan");
    s.push("Mark");
    s.push("Tom");
    assertTrue(s.size() == 3 && !s.isEmpty());
    assertEquals("Tom", s.top());
    String oldTop = s.pop();
    assertEquals("Tom", oldTop);
    String newTop = s.top();
    assertEquals("Mark", newTop);
    oldTop = s.pop();
    assertEquals("Mark", oldTop);
    newTop = s.top();
    assertEquals("Alan", newTop);
}
```

Implementing Stack ADT: Array (3)

Running Times of *Array*-Based *Stack* Operations?

<i>ArrayedStack</i> Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Q: What if the preset capacity turns out to be insufficient?

A: $O(n)$ time to grow the array size and copy existing contents!

Implementing Stack ADT: Singly-Linked List (1)

```
public class LinkedStack {
    private SinglyLinkedList list; /* assumed: head, tail, size */
    ...
}
```

Question:

Stack Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
top	list.first	list.last
push	list.addFirst	list.addLast
pop	list.removeFirst	list.removeLast

Which *implementation strategy* should be chosen? Either?

Implementing Stack ADT: Singly-Linked List (2)

- If the *front of list* is treated as the *top of stack*, then:
 - All stack operations remain $O(1)$.
 - *No resizing* is necessary!
- If the *back of list* is treated as the *top of stack*, then:
 - Still *no resizing* is necessary!
 - The pop operation (via `removeLast`) takes $O(n)$!

Application (1): Reversing an Array

```
public static void reverse(String[] a) {  
    ArrayedStack buffer = new ArrayedStack();  
    for (int i = 0; i < a.length; i++) {  
        buffer.push(a[i]);  
    }  
    for (int i = 0; i < a.length; i++) {  
        a[i] = buffer.pop();  
    }  
}
```

```
@Test  
public void testReverseViaStack() {  
    String[] names = {"Alan", "Mark", "Tom"};  
    String[] reverseOfNames = {"Tom", "Mark", "Alan"};  
    StackUtilities.reverse(names);  
    assertEquals(reverseOfNames, names);  
}
```

Application (2): Matching Delimiters

• Problem

Opening delimiters: (, [, {

Closing delimiters:),], }

e.g., *Correct*: “()()([()])”

e.g., *Incorrect*:

- “[()]” [mismatched opening and closing]
- “{ }” [more openings than closings]
- “{ }” [more closings than openings]

• Can we simply say $s.equals(reverseOf(s)) \Rightarrow isMatched(s)$?

- e.g., “[()]” is matched, and its reverse are equal.
- **NO!** e.g., “[()]([()])” matched, but different from its reverse.

• Sketch of Solution

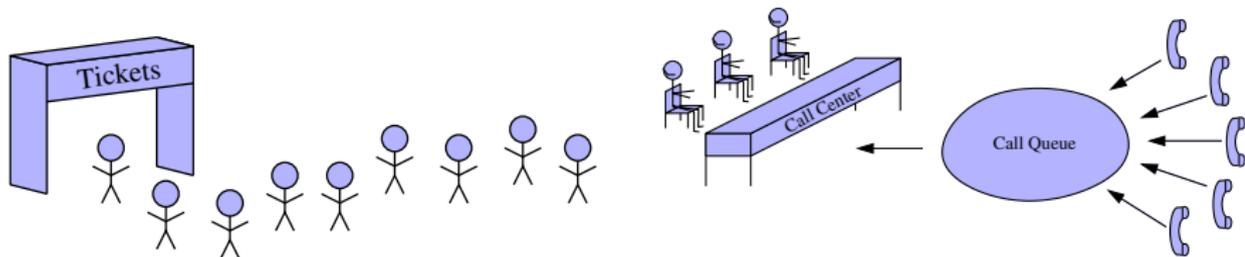
- When a new *opening* delimiter is found, *push* it to the **stack**.
- When a new *closing* delimiter is found:
 - If it matches the *top* of the **stack**, then *pop* off the stack.
 - Otherwise, an error is found!
- Finishing reading the input, an empty **stack** means a success!

Application (2): Matching Delimiters in Java

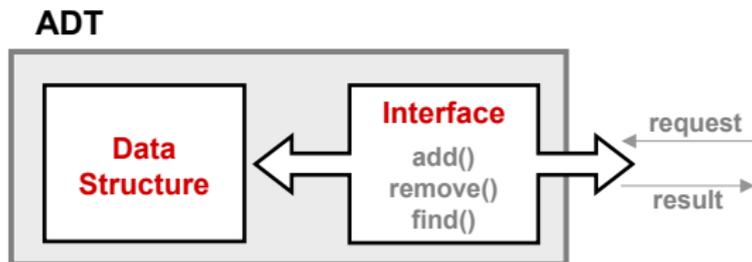
```
public static boolean isMatched(String expression) {
    final String open = "[{";
    final String close = ")]";
    ArrayedStack openings = new ArrayedStack();
    for (int i = 0; i < expression.length(); i++) {
        String c = Character.toString(expression.charAt(i));
        if(open.indexOf(c) != -1) { openings.push(c); }
        else if (close.indexOf(c) != -1) {
            if (openings.isEmpty()) { return false; /* e.g., } */ }
            else {
                if (open.indexOf(openings.top()) == close.indexOf(c)) {
                    openings.pop(); }
                else { return false; /* e.g., ( */ }
            }
        }
    }
    return openings.isEmpty(); /* e.g., {{ */ }
}
```

What is a Queue?

- A **queue** is a collection of objects.
- Objects in a **queue** are inserted and removed according to the **first-in, first-out (FIFO)** principle.
 - Each new element joins at the **back** of the queue.
 - **Cannot** access **arbitrary** elements of a queue
 - **Can** only access or remove the **front** of queue: **least-recently (or longest) inserted** element



The Queue ADT



- **Accessors**

- *first*
- *size*
- *isEmpty*

[compare: *top* of stack]

- **Mutators**

- *enqueue*
- *dequeue*

[compare: *push* of stack]
[compare: *pop* of stack]

Queue: Illustration

Operation	Return Value	Queue Contents
–	–	∅
isEmpty	<i>true</i>	∅
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
enqueue(1)	–	(5, 3, 1)
size	3	(5, 3, 1)
dequeue	5	(3, 1)
dequeue	3	1
dequeue	1	∅

Implementing Queue ADT: Array (1)

```
public class ArrayedQueue {
    private static final int MAX_CAPACITY = 1000;
    private String[] data;
    private int r; /* rear index */
    public ArrayedQueue() { data = new String[MAX_CAPACITY]; r = -1; }
    public int size() { return (r + 1); }
    public boolean isEmpty() { return (r == -1); }
    public String first() {
        if (isEmpty()) { /* Error: Empty Queue */ }
        else { return data[0]; } }
    public void enqueue(String e) {
        if (size() == MAX_CAPACITY) { /* Error: Queue Full. */ }
        else { r++; data[r] = e; } }
    public String dequeue() {
        String result;
        if (isEmpty()) { /* Error: Empty Queue. */ }
        else {
            result = data[0];
            for (int i = 0; i < r; i++) { data[i] = data[i + 1]; }
            r--; }
        return result; }
}
```

Implementing Queue ADT: Array (2)

```
@Test
public void testArrayedQueue() {
    ArrayedQueue q = new ArrayedQueue();
    assertTrue(q.size() == 0 && q.isEmpty());
    try { String first = q.first();
        fail("Empty queue should have caused an exception."); }
    catch (IllegalArgumentException e) { }
    q.enqueue("Alan");
    q.enqueue("Mark");
    q.enqueue("Tom");
    assertTrue(q.size() == 3 && !q.isEmpty());
    assertEquals("Alan", q.first());
    String oldFirst = q.dequeue();
    assertEquals("Alan", oldFirst);
    String newFirst = q.first();
    assertEquals("Mark", newFirst);
    oldFirst = q.dequeue();
    assertEquals("Mark", oldFirst);
    newFirst = q.first();
    assertEquals("Tom", newFirst);
}
```

Implementing Queue ADT: Array (3)

Running Times of *Array*-Based *Queue* Operations?

<i>ArrayQueue</i> Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(n)$

Q: What if the preset capacity turns out to be insufficient?

A: $O(n)$ time to grow the array size and copy existing contents!

Implementing Queue ADT: Singly-Linked List (1)

```
public class LinkedQueue {
    private SinglyLinkedList list; /* assumed: head, tail, size */
    ...
}
```

Question:

Queue Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
first	list.first	list.last
enqueue	list.addLast	list.addFirst
dequeue	list.removeFirst	list.removeLast

Which *implementation strategy* should be chosen? Either?

Implementing Queue ADT: Singly-Linked List (2)

- If the *front of list* is treated as the *first of queue*, then:
 - All queue operations remain $O(1)$.
 - *No resizing* is necessary!
- If the *back of list* is treated as the *first of queue*, then:
 - Still *no resizing* is necessary!
 - The dequeue operation (via `removeLast`) takes $O(n)$!

Index (1)

What is a Stack?

The Stack ADT

Stack: Illustration

Implementing Stack ADT: Array (1)

Implementing Stack ADT: Array (2)

Implementing Stack ADT: Array (3)

Implementing Stack ADT:

Singly-Linked List (1)

Implementing Stack ADT:

Singly-Linked List (2)

Application (1): Reversing an Array

Application (2): Matching Delimiters

Application (2): Matching Delimiters in Java

What is a Queue?

Index (2)

The Queue ADT

Queue: Illustration

Implementing Queue ADT: Array (1)

Implementing Queue ADT: Array (2)

Implementing Queue ADT: Array (3)

Implementing Queue ADT: Singly-Linked List (1)

Implementing Queue ADT: Singly-Linked List (2)

Recursion



EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Recursion: Principle

- **Recursion** is useful in expressing solutions to problems that can be **recursively** defined:
 - **Base Cases:** Small problem instances immediately solvable.
 - **Recursive Cases:**
 - Large problem instances *not immediately solvable*.
 - Solve by reusing *solution(s) to strictly smaller problem instances*.
- Similar idea learnt in high school: [**mathematical induction**]
- Recursion can be easily expressed programmatically in Java:
 - In the body of a method m , there might be *a call or calls to m itself*.
 - Each such self-call is said to be a **recursive call**.
 - Inside the execution of $m(i)$, a recursive call $m(j)$ must be that $j < i$.

```
m(i) {  
    ...  
    m(j); /* recursive call with strictly smaller value */  
    ...  
}
```

Recursion: Factorial (1)

- Recall the formal definition of calculating the n factorial:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

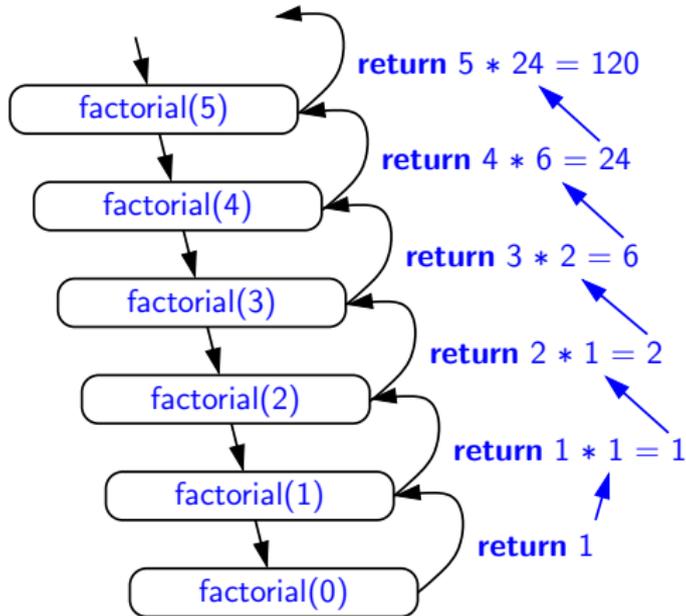
- How do you define the same problem *recursively*?

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

- To solve $n!$, we combine n and the solution to $(n - 1)!$.

```
int factorial(int n) {
    int result;
    if(n == 0) { /* base case */ result = 1; }
    else { /* recursive case */
        result = n * factorial(n - 1);
    }
    return result;
}
```

Recursion: Factorial (2)



Recursion: Factorial (3)

- When running *factorial(5)*, a *recursive call factorial(4)* is made. Call to *factorial(5)* suspended until *factorial(4)* returns a value.
- When running *factorial(4)*, a *recursive call factorial(3)* is made. Call to *factorial(4)* suspended until *factorial(3)* returns a value.
- ...
- *factorial(0)* returns 1 back to *suspended call factorial(1)*.
- *factorial(1)* receives 1 from *factorial(0)*, multiplies 1 to it, and returns 1 back to the *suspended call factorial(2)*.
- *factorial(2)* receives 1 from *factorial(1)*, multiplies 2 to it, and returns 2 back to the *suspended call factorial(3)*.
- *factorial(3)* receives 2 from *factorial(1)*, multiplies 3 to it, and returns 6 back to the *suspended call factorial(4)*.
- *factorial(4)* receives 6 from *factorial(3)*, multiplies 4 to it, and returns 24 back to the *suspended call factorial(5)*.
- *factorial(5)* receives 24 from *factorial(4)*, multiplies 5 to it, and returns 120 as the result.

Recursion: Factorial (4)

- When the execution of a method (e.g., *factorial(5)*) leads to a nested method call (e.g., *factorial(4)*):
 - The execution of the current method (i.e., *factorial(5)*) is *suspended*, and a structure known as an *activation record* or *activation frame* is created to store information about the progress of that method (e.g., values of parameters and local variables).
 - The nested methods (e.g., *factorial(4)*) may call other nested methods (*factorial(3)*).
 - When all nested methods complete, the activation frame of the *latest suspended* method is re-activated, then continue its execution.
- What kind of data structure does this activation-suspension process correspond to? [LIFO Stack]

Tracing Recursion using a Stack

- When a method is called, it is **activated** (and becomes **active**) and **pushed** onto the stack.
- When the body of a method makes a (helper) method call, that (helper) method is **activated** (and becomes **active**) and **pushed** onto the stack.
 - ⇒ The stack contains activation records of all **active** methods.
 - **Top** of stack denotes the current point of execution.
 - Remaining parts of stack are (temporarily) **suspended**.
- When entire body of a method is executed, stack is **popped**.
 - ⇒ The current point of execution is returned to the new **top** of stack (which was **suspended** and just became **active**).
- Execution terminates when the stack becomes **empty**.

Recursion: Fibonacci (1)

Recall the formal definition of calculating the n_{th} number in a Fibonacci series (denoted as F_n), which is already itself recursive:

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

```
int fib (int n) {
    int result;
    if(n == 1) { /* base case */ result = 1; }
    else if(n == 2) { /* base case */ result = 1; }
    else { /* recursive case */
        result = fib (n - 1) + fib (n - 2);
    }
    return result;
}
```

Recursion: Fibonacci (2)

```

fib(5)
= {fib(5) = fib(4) + fib(3); push(fib(5)); suspended: {fib(5)}; active: fib(4)}
  fib(4) + fib(3)
= {fib(4) = fib(3) + fib(2); suspended: {fib(4), fib(5)}; active: fib(3)}
  ( fib(3) + fib(2) ) + fib(3)
= {fib(3) = fib(2) + fib(1); suspended: {fib(3), fib(4), fib(5)}; active: fib(2)}
  (( fib(2) + fib(1) ) + fib(2) ) + fib(3)
= {fib(2) returns 1; suspended: {fib(3), fib(4), fib(5)}; active: fib(1)}
  ((1 + fib(1) ) + fib(2) ) + fib(3)
= {fib(1) returns 1; suspended: {fib(3), fib(4), fib(5)}; active: fib(3)}
  ((1 + 1) + fib(2) ) + fib(3)
= {fib(3) returns 1 + 1; pop(); suspended: {fib(4), fib(5)}; active: fib(2)}
  (2 + fib(2) ) + fib(3)
= {fib(2) returns 1; suspended: {fib(4), fib(5)}; active: fib(4)}
  (2 + 1) + fib(3)
= {fib(4) returns 2 + 1; pop(); suspended: {fib(5)}; active: fib(3)}
  3 + fib(3)
= {fib(3) = fib(2) + fib(1); suspended: {fib(3), fib(5)}; active: fib(2)}
  3 + ( fib(2) + fib(1) )
= {fib(2) returns 1; suspended: {fib(3), fib(5)}; active: fib(1)}
  3 + (1 + fib(1) )
= {fib(1) returns 1; suspended: {fib(3), fib(5)}; active: fib(3)}
  3 + (1 + 1)
= {fib(3) returns 1 + 1; pop(); suspended: {fib(5)}; active: fib(5)}
  3 + 2
= {fib(5) returns 3 + 2; suspended: {}}

```

Java Library: String

```
public class StringTester {
    public static void main(String[] args) {
        String s = "abcd";
        System.out.println(s.isEmpty()); /* false */
        /* Characters in index range [0, 0) */
        String t0 = s.substring(0, 0);
        System.out.println(t0); /* "" */
        /* Characters in index range [0, 4) */
        String t1 = s.substring(0, 4);
        System.out.println(t1); /* "abcd" */
        /* Characters in index range [1, 3) */
        String t2 = s.substring(1, 3);
        System.out.println(t2); /* "bc" */
        String t3 = s.substring(0, 2) + s.substring(2, 4);
        System.out.println(s.equals(t3)); /* true */
        for(int i = 0; i < s.length(); i++) {
            System.out.print(s.charAt(i));
        }
        System.out.println();
    }
}
```

Recursion: Palindrome (1)

Problem: A palindrome is a word that reads the same forwards and backwards. Write a method that takes a string and determines whether or not it is a palindrome.

```
System.out.println(isPalindrome("")); true
System.out.println(isPalindrome("a")); true
System.out.println(isPalindrome("madam")); true
System.out.println(isPalindrome("racecar")); true
System.out.println(isPalindrome("man")); false
```

Base Case 1: Empty string \rightarrow Return *true* immediately.

Base Case 2: String of length 1 \rightarrow Return *true* immediately.

Recursive Case: String of length ≥ 2 \rightarrow

- 1st and last characters match, **and**
- *the rest (i.e., middle) of the string* is a palindrome .

Recursion: Palindrome (2)

```
boolean isPalindrome (String word) {
    if (word.length() == 0 || word.length() == 1) {
        /* base case */
        return true;
    }
    else {
        /* recursive case */
        char firstChar = word.charAt(0);
        char lastChar = word.charAt(word.length() - 1);
        String middle = word.substring(1, word.length() - 1);
        return
            firstChar == lastChar
            /* See the API of java.lang.String.substring. */
            && isPalindrome (middle);
    }
}
```

Recursion: Reverse of String (1)

Problem: The reverse of a string is written backwards. Write a method that takes a string and returns its reverse.

```
System.out.println(reverseOf("")); /* "" */  
System.out.println(reverseOf("a")); "a"  
System.out.println(reverseOf("ab")); "ba"  
System.out.println(reverseOf("abc")); "cba"  
System.out.println(reverseOf("abcd")); "dcba"
```

Base Case 1: Empty string \rightarrow Return *empty string*.

Base Case 2: String of length 1 \rightarrow Return *that string*.

Recursive Case: String of length $\geq 2 \rightarrow$

- 1) Head of string (i.e., first character)
- 2) Reverse of the tail of string (i.e., all but the first character)

Return the concatenation of **1)** and **2)**.

Recursion: Reverse of a String (2)

```
String reverseOf (String s) {  
    if(s.isEmpty()) { /* base case 1 */  
        return "";  
    }  
    else if(s.length() == 1) { /* base case 2 */  
        return s;  
    }  
    else { /* recursive case */  
        String tail = s.substring(1, s.length());  
        String reverseOfTail = reverseOf (tail);  
        char head = s.charAt(0);  
        return reverseOfTail + head;  
    }  
}
```

Recursion: Number of Occurrences (1)

Problem: Write a method that takes a string s and a character c , then count the number of occurrences of c in s .

```
System.out.println(occurrencesOf("", 'a')); /* 0 */  
System.out.println(occurrencesOf("a", 'a')); /* 1 */  
System.out.println(occurrencesOf("b", 'a')); /* 0 */  
System.out.println(occurrencesOf("baaba", 'a')); /* 3 */  
System.out.println(occurrencesOf("baaba", 'b')); /* 2 */  
System.out.println(occurrencesOf("baaba", 'c')); /* 0 */
```

Base Case: Empty string \rightarrow Return 0 .

Recursive Case: String of length $\geq 1 \rightarrow$

- 1) Head of s (i.e., first character)
- 2) Number of occurrences of c in the tail of s (i.e., all but the first character)

If head is equal to c , return $1 + 2$).

If head is not equal to c , return $0 + 2$).

Recursion: Number of Occurrences (2)

```
int occurrencesOf (String s, char c) {
    if (s.isEmpty()) {
        /* Base Case */
        return 0;
    }
    else {
        /* Recursive Case */
        char head = s.charAt(0);
        String tail = s.substring(1, s.length());
        if (head == c) {
            return 1 + occurrencesOf (tail, c);
        }
        else {
            return 0 + occurrencesOf (tail, c);
        }
    }
}
```

Recursion: All Positive (1)

Problem: Determine if an array of integers are all positive.

```
System.out.println(allPositive({})); /* true */  
System.out.println(allPositive({1, 2, 3, 4, 5})); /* true */  
System.out.println(allPositive({1, 2, -3, 4, 5})); /* false */
```

Base Case: Empty array \rightarrow Return *true* immediately.

The base case is *true* \because we can *not* find a counter-example (i.e., a number *not* positive) from an empty array.

Recursive Case: Non-Empty array \rightarrow

- 1st element positive, **and**
- *the rest of the array* is all positive.

Exercise: Write a method `boolean somePositive(int[]`

a) which *recursively* returns *true* if there is some positive number in a, and *false* if there are no positive numbers in a.

Hint: What to return in the base case of an empty array? [*false*]

\because No witness (i.e., a positive number) from an empty array

Making Recursive Calls on an Array

- Recursive calls denote solutions to *smaller* sub-problems.
- *Naively*, explicitly create a new, smaller array:

```
void m(int[] a) {
    int[] subArray = new int[a.length - 1];
    for(int i = 1; i < a.length; i++) { subArray[0] = a[i - 1]; }
    m(subArray) }
```

- For *efficiency*, we pass the same array **by reference** and specify the *range of indices* to be considered:

```
void m(int[] a, int from, int to) {
    if(from == to) { /* base case */ }
    else { m(a, from + 1, to) } }
```

- $m(a, 0, a.length - 1)$ [Initial call; entire array]
- $m(a, 1, a.length - 1)$ [1st r.c. on array of size $a.length - 1$]
- $m(a, 2, a.length - 1)$ [2nd r.c. on array of size $a.length - 2$]
- ...
- $m(a, a.length-1, a.length-1)$ [Last r.c. on array of size 1]

Recursion: All Positive (2)

```
boolean allPositive(int[] a) {  
    return allPositiveHelper(a, 0, a.length - 1);  
}  
  
boolean allPositiveHelper(int[] a, int from, int to) {  
    if (from > to) { /* base case 1: empty range */  
        return true;  
    }  
    else if (from == to) { /* base case 2: range of one element */  
        return a[from] > 0;  
    }  
    else { /* recursive case */  
        return a[from] > 0 && allPositiveHelper(a, from + 1, to);  
    }  
}
```

Recursion: Is an Array Sorted? (1)

Problem: Determine if an array of integers are sorted in a non-descending order.

```
System.out.println(isSorted({})); true
```

```
System.out.println(isSorted({1, 2, 2, 3, 4})); true
```

```
System.out.println(isSorted({1, 2, 2, 1, 3})); false
```

Base Case: Empty array → Return *true* immediately.

The base case is *true* ∵ we can *not* find a counter-example (i.e., a pair of adjacent numbers that are *not* sorted in a non-descending order) from an empty array.

Recursive Case: Non-Empty array →

- 1st and 2nd elements are sorted in a non-descending order, **and**
- *the rest of the array*, starting from the 2nd element, **are sorted in a non-descending order**.

Recursion: Is an Array Sorted? (2)

```
boolean isSorted(int[] a) {  
    return isSortedHelper(a, 0, a.length - 1);  
}  
  
boolean isSortedHelper(int[] a, int from, int to) {  
    if (from > to) { /* base case 1: empty range */  
        return true;  
    }  
    else if (from == to) { /* base case 2: range of one element */  
        return true;  
    }  
    else {  
        return a[from] <= a[from + 1]  
            && isSortedHelper(a, from + 1, to);  
    }  
}
```

Recursion: Sorting an Array (1)

Problem: Sort an array into a non-descending order, using the *selection-sort* strategy.

Base Case: Arrays of size 0 or 1 \rightarrow Completed immediately.

Recursive Case: Non-Empty array $a \rightarrow$

Run an iteration from indices $i = 0$ to $a.length - 1$.

In each iteration:

- In index range $[i, a.length - 1]$, *recursively* compute the *minimum* element e .
- Swap $a[i]$ and e if $e < a[i]$.

Recursion: Sorting an Array (2)

```
public static int getMinIndex(int[] a, int from, int to) {  
    if(from == to) { return from; }  
    else {  
        int minIndexOfTail = getMinIndex(a, from + 1, to);  
        if(a[from] < a[minIndexOfTail]) { return from; }  
        else { return minIndexOfTail; }  
    }  
}  
  
public static void selectionSort(int[] a) {  
    if(a.length == 0 || a.length == 1) { /* sorted, do nothing */ }  
    else {  
        for(int i = 0; i < a.length; i++) {  
            int minIndex = getMinIndex(a, i, a.length - 1);  
            /* swap a[i] and a[minIndex] */  
            int temp = a[i];  
            a[i] = a[minIndex];  
            a[minIndex] = temp;  
        }  
    }  
}
```

Recursion: Binary Search (1)

- **Searching Problem**

Input: A number a and a **sorted** list of n numbers $\langle a_1, a_2, \dots, a_n \rangle$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Output: Whether or not a exists in the input list

- **An Efficient Recursive Solution**

Base Case: Empty list \rightarrow *False*.

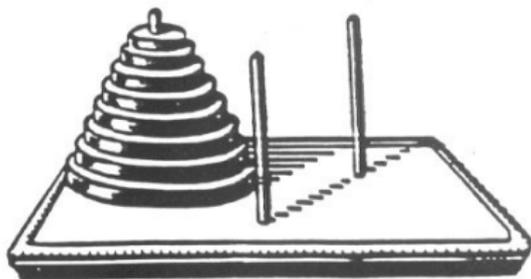
Recursive Case: List of size $\geq 1 \rightarrow$

- **Compare** the *middle* element against a .
 - All elements to the left of *middle* are $\leq a$
 - All elements to the right of *middle* are $\geq a$
- If the *middle* element *is* equal to $a \rightarrow$ *True*.
- If the *middle* element *is not* equal to a :
 - If $a < \text{middle}$, recursively find a on the left half.
 - If $a > \text{middle}$, recursively find a on the right half.

Recursion: Binary Search (2)

```
boolean binarySearch(int[] sorted, int key) {  
    return binarySearchHelper(sorted, 0, sorted.length - 1, key);  
}  
boolean binarySearchHelper(int[] sorted, int from, int to, int key) {  
    if (from > to) { /* base case 1: empty range */  
        return false; }  
    else if (from == to) { /* base case 2: range of one element */  
        return sorted[from] == key; }  
    else {  
        int middle = (from + to) / 2;  
        int middleValue = sorted[middle];  
        if (key < middleValue) {  
            return binarySearchHelper(sorted, from, middle - 1, key);  
        }  
        else if (key > middleValue) {  
            return binarySearchHelper(sorted, middle + 1, to, key);  
        }  
        else { return true; }  
    }  
}
```

Tower of Hanoi: Specification



- *Given:* A tower of 8 disks, initially stacked in decreasing size on one of 3 pegs
- *Rules:*
 - Move only one disk at a time
 - Never move a larger disk onto a smaller one
- *Problem:* Transfer the entire tower to one of the other pegs.

Tower of Hanoi: Strategy

- Generalize the problem by considering n disks.
- Introduce appropriate notation:
 - T_n denotes the *minimum* number of moves required to transfer n disks from one to another under the rules.
- General patterns are easier to perceive when the extreme or base cases are well understood.
 - Look at small cases first:
 - $T_1 = 1$
 - $T_2 = 3$
 - How about T_3 ? Does it help us perceive the general case of n ?

Tower of Hanoi: A General Solution Pattern

A possible (yet to be proved as *optimal*) solution requires 3 steps:

1. Transfer the $n - 1$ smallest disks to a different peg.
2. Move the largest to the remaining free peg.
3. Transfer the $n - 1$ disks back onto the largest disk.

How many moves are required from the above 3 steps?

$$(2 \times T_{n-1}) + 1$$

However, we have only proved that the # moves required by this solution are *sufficient*:

$$T_n \leq (2 \times T_{n-1}) + 1$$

But are the above steps all *necessary*? Can you justify?

$$T_n \geq (2 \times T_{n-1}) + 1$$

Tower of Hanoi: Recurrence Relation for T_n



We end up with the following recurrence relation that allows us to compute T_n for any n we like:

$$\begin{aligned}T_0 &= 0 \\T_n &= (2 \times T_{n-1}) + 1 \quad \text{for } n > 0\end{aligned}$$

However, the above relation only gives us *indirect* information:

To calculate T_n , first calculate T_{n-1} , which requires the calculation of T_{n-2} , and so on.

Instead, we need a *closed-form solution* to the above recurrence relation, which allows us to *directly* calculate the value of T_n .

Tower of Hanoi: A Hypothesized Closed Form Solution to T_n

$$\begin{aligned}T_0 &= 0 \\T_1 &= 2 \times T_0 + 1 = 1 \\T_2 &= 2 \times T_1 + 1 = 3 \\T_3 &= 2 \times T_2 + 1 = 7 \\T_4 &= 2 \times T_3 + 1 = 15 \\T_5 &= 2 \times T_4 + 1 = 31 \\T_6 &= 2 \times T_5 + 1 = 63 \\&\dots\end{aligned}$$

Guess:

$$T_n = 2^n - 1 \quad \text{for } n \geq 0$$

Prove by *mathematical induction*.

Tower of Hanoi: Prove by Mathematical Induction

Basis:

$$T_0 = 2^0 - 1 = 0$$

Induction:

Assume that

$$T_{n-1} = 2^{n-1} - 1$$

then

$$\begin{aligned} & T_n \\ = & \{ \text{Recurrence relation for } T_n \} \\ & (2 \times T_{n-1}) + 1 \\ = & \{ \text{Inductive assumption} \} \\ & (2 \times (2^{n-1} - 1)) + 1 \\ = & \{ \text{Arithmetic} \} \\ & 2^n - 1 \end{aligned}$$

Revisiting the Tower of Hanoi

Given: A tower of 8 disks, initially stacked in decreasing size on one of 3 pegs.

This shall require

$$T_8 = 2^8 - 1 = 255$$

moves to complete.

Tower of Hanoi in Java (1)

```
void towerOfHanoi(String[] disks) {
    toHelper (disks, 0, disks.length - 1, 1, 3);
}
void toHelper(String[] disks, int from, int to, int p1, int p2) {
    if(from > to) { }
    else if(from == to) {
        print("move " + disks[to] + " from " + p1 + " to " + p2);
    }
    else {
        int intermediate = 6 - p1 - p2;
        toHelper (disks, from, to - 1, p1, intermediate);
        print("move " + disks[to] + " from " + p1 + " to " + p2);
        toHelper (disks, from, to - 1, intermediate, p2);
    }
}
```

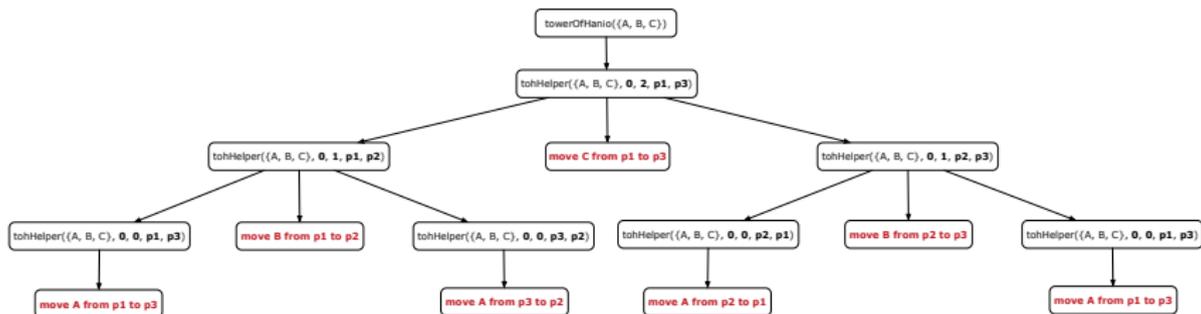
- `toHelper(disks, from, to, p1, p2)` moves disks $\{disks[from], disks[from + 1], \dots, disks[to]\}$ from peg $p1$ to peg $p2$.
- Peg id's are 1, 2, and 3 \Rightarrow The intermediate one is $6 - p1 - p2$.

Tower of Hanoi in Java (2)

Say ds (disks) is $\{A, B, C\}$, where $A < B < C$.

$$\text{tohH}(ds, \underbrace{0, 2}_{\{A, B, C\}}, p1, p3) = \left\{ \begin{array}{l} \text{Move C: } p1 \text{ to } p3 \\ \text{tohH}(ds, \underbrace{0, 1}_{\{A, B\}}, p1, p2) = \left\{ \begin{array}{l} \text{tohH}(ds, \underbrace{0, 0}_{\{A\}}, p1, p3) = \left\{ \text{Move A: } p1 \text{ to } p3 \right. \\ \text{Move B: } p1 \text{ to } p2 \\ \text{tohH}(ds, \underbrace{0, 0}_{\{A\}}, p3, p2) = \left\{ \text{Move A: } p3 \text{ to } p2 \right. \end{array} \right. \\ \text{tohH}(ds, \underbrace{0, 1}_{\{A, B\}}, p2, p3) = \left\{ \begin{array}{l} \text{tohH}(ds, \underbrace{0, 0}_{\{A\}}, p2, p1) = \left\{ \text{Move A: } p2 \text{ to } p1 \right. \\ \text{Move B: } p2 \text{ to } p3 \\ \text{tohH}(ds, \underbrace{0, 0}_{\{A\}}, p1, p3) = \left\{ \text{Move A: } p1 \text{ to } p3 \right. \end{array} \right. \end{array} \right.$$

Tower of Hanoi in Java (3)



Recursive Methods: Correctness Proofs

```
1 boolean allPositive(int[] a) { return allPosH(a, 0, a.length - 1); }
2 boolean allPosH(int[] a, int from, int to) {
3     if (from > to) { return true; }
4     else if (from == to) { return a[from] > 0; }
5     else { return a[from] > 0 && allPosH(a, from + 1, to); } }
```

- Via mathematical induction, prove that `allPosH` is correct:

Base Cases

- In an empty array, there is no non-positive number \therefore result is **true**. [L3]
- In an array of size 1, the only one elements determines the result. [L4]

Inductive Cases

- **Inductive Hypothesis:** `allPosH(a, from + 1, to)` returns **true** if `a[from + 1]`, `a[from + 2]`, ..., `a[to]` are all positive; **false** otherwise.
- `allPosH(a, from, to)` should return **true** if: **1)** `a[from]` is positive; **and 2)** `a[from + 1]`, `a[from + 2]`, ..., `a[to]` are all positive.
- By **I.H.**, result is `a[from] > 0` \wedge `allPosH(a, from + 1, to)`. [L5]
- `allPositive(a)` is correct by invoking `allPosH(a, 0, a.length - 1)`, examining the entire array. [L1]

Beyond this lecture ...

- Notes on Recursion:

http://www.eecs.yorku.ca/~jackie/teaching/lectures/2017/F/EECS2030/slides/EECS2030_F17_Notes_Recursion.pdf

- API for String:

<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

- Fantastic resources for sharpening your recursive skills for the exam:

<http://codingbat.com/java/Recursion-1>

<http://codingbat.com/java/Recursion-2>

- The **best** approach to learning about recursion is via a functional programming language:

Haskell Tutorial: <https://www.haskell.org/tutorial/>

Index (1)

Recursion: Principle

Recursion: Factorial (1)

Recursion: Factorial (2)

Recursion: Factorial (3)

Recursion: Factorial (4)

Tracing Recursion using a Stack

Recursion: Fibonacci (1)

Recursion: Fibonacci (2)

Java Library: String

Recursion: Palindrome (1)

Recursion: Palindrome (2)

Recursion: Reverse of a String (1)

Recursion: Reverse of a String (2)

Recursion: Number of Occurrences (1)

Index (2)

Recursion: Number of Occurrences (2)

Recursion: All Positive (1)

Making Recursive Calls on an Array

Recursion: All Positive (2)

Recursion: Is an Array Sorted? (1)

Recursion: Is an Array Sorted? (2)

Recursion: Sorting an Array (1)

Recursion: Sorting an Array (2)

Recursion: Binary Search (1)

Recursion: Binary Search (2)

Tower of Hanoi: Specification

Tower of Hanoi: Strategy

Tower of Hanoi: A General Solution Pattern

Tower of Hanoi: Recurrence Relation for T_n

Index (3)

Tower of Hanoi:

A Hypothesized Closed Form Solution to T_n

Tower of Hanoi:

Prove by Mathematical Induction

Revisiting the Tower of Hanoi

Tower of Hanoi in Java (1)

Tower of Hanoi in Java (2)

Tower of Hanoi in Java (3)

Recursive Methods: Correctness Proofs

Beyond this lecture ...

Binary Trees



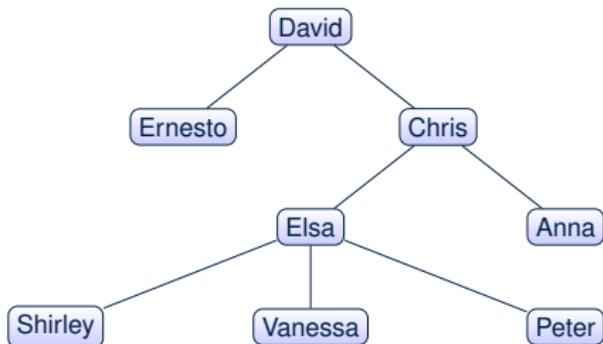
EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

General Trees

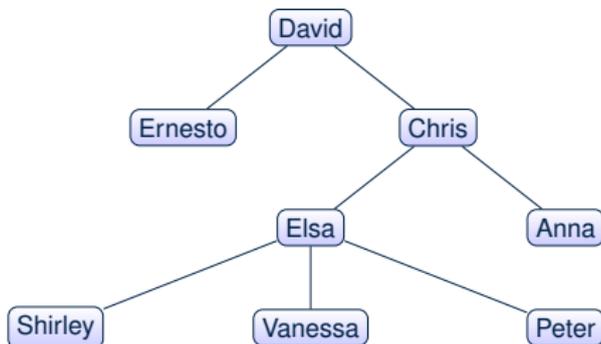
- A *linear* data structure is a sequence, where stored objects can be related via the “*before*” and “*after*” relationships.
e.g., arrays, singly-linked lists, and doubly-linked lists
- A **tree** is a *non-linear* collection of nodes.
 - Each node stores some data object.
 - Nodes stored in a **tree** is organized in a *non-linear* manner.
 - In a **tree**, the relationships between stored objects are **hierarchical**: some objects are “*above*” others, and some are “*below*” others.
- The main terminology for the **tree** data structure comes from that of family trees: parents, siblings, children, ancestors, descendants.

General Trees: Terminology (1)



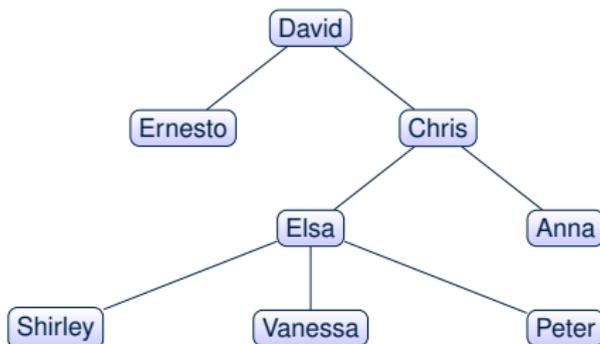
- **root of tree**: top element of the tree
e.g., *root* of the above family tree: David
- **parent of node v** : node immediately above and connected to v
e.g., *parent* of Vanessa: Elsa
- **children of node v** : nodes immediately below and connected to v
e.g., *children* of Elsa: Shirley, Vanessa, and Peter
e.g., *children* of Ernesto: \emptyset

General Trees: Terminology (2)



- **ancestors of node v** : v + v 's parent + v 's grand parent + ...
e.g., **ancestors** of Vanessa: **Vanessa**, Elsa, Chris, and David
e.g., **ancestors** of David: David
- **descendants of node v** : v + v 's children + v 's grand children + ...
e.g., **descendants** of Vanessa: Vanessa
e.g., **descendants** of David: the entire family tree

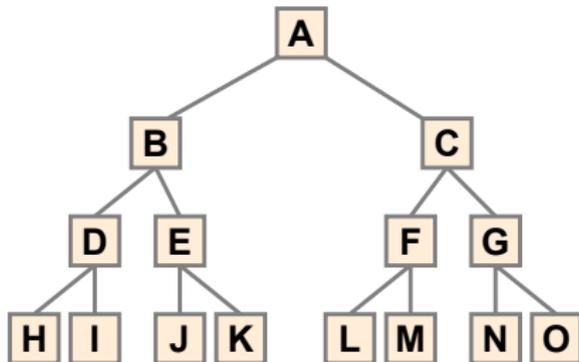
General Trees: Terminology (3)



- **siblings of node v** : nodes whose parents are the same as v 's
e.g., **siblings** of Vanessa: Shirley and Peter
- **subtree rooted at v** : a tree formed by all descendant of v
- **external nodes (leaves)** : nodes that have no children
e.g., **leaves** of the above tree: Ernesto, Anna, Shirley, Vanessa, Peter
- **internal nodes** : nodes that has at least one children
e.g., **non-leaves** of the above tree: David, Chris, Elsa

Exercise: Identifying Subtrees

How many subtrees are there?



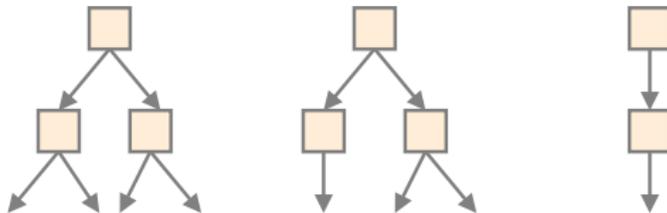
15 subtrees

[i.e., subtrees rooted at each node]

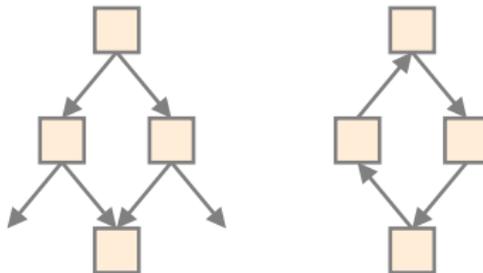
SIZE OF SUBTREE	ROOTS OF SUBTREES
1	H, I, J, K, L, M, N, O
3	D, E, F, G
7	B, C
15	A

General Tree: Important Characteristics

There is a *single unique path* along the edges from the *root* to any particular node.



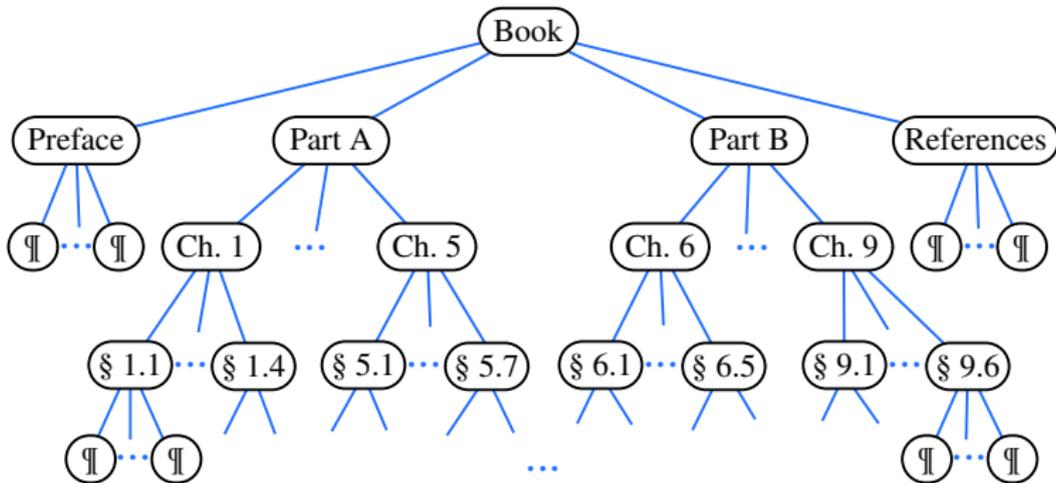
legal tree organization



illegal tree organization (nontrees)

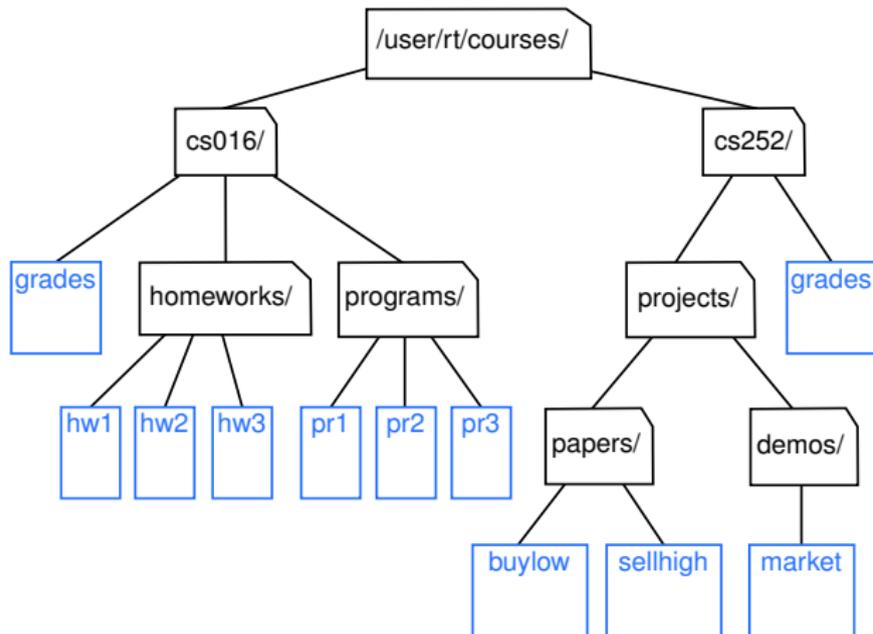
General Trees: Ordered Trees

A tree is **ordered** if there is a meaningful *linear* order among the *children* of each node.



General Trees: Unordered Trees

A tree is **unordered** if the order among the *children* of each node does not matter.



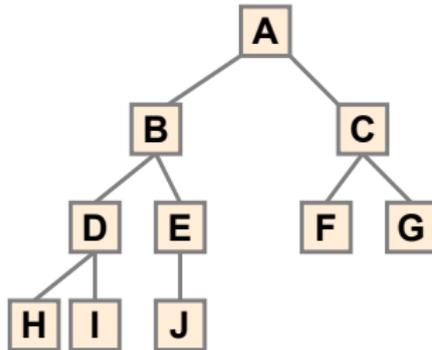
Binary Trees

- A **binary tree** is an *ordered* tree which satisfies the following properties:
 1. Each node has *at most two* children.
 2. Each child node is labeled as either a **left child** or a **right child**.
 3. A *left child* precedes a *right child* in the order of children of a node.

Binary Trees: Terminology (1)

For an *internal* node n :

- Subtree rooted at its *left child* is called **left subtree**.
 n has no left child \Rightarrow n 's left subtree is **empty**
- Subtree rooted at its *right child* is called **right subtree**.
 n has no right child \Rightarrow n 's right subtree is **empty**



A 's *left subtree* is rooted at B and *right subtree* rooted at C .

H 's *left subtree* and *right subtree* are both empty.

Binary Trees: Recursive Definition

A **binary** tree is either:

- An **empty** tree; or
- A **nonempty** tree with a root node r that
 - has a **left binary subtree** rooted at its left child
 - has a **right binary subtree** rooted at its right child

⇒ To solve problems **recursively** on a binary tree rooted at r :

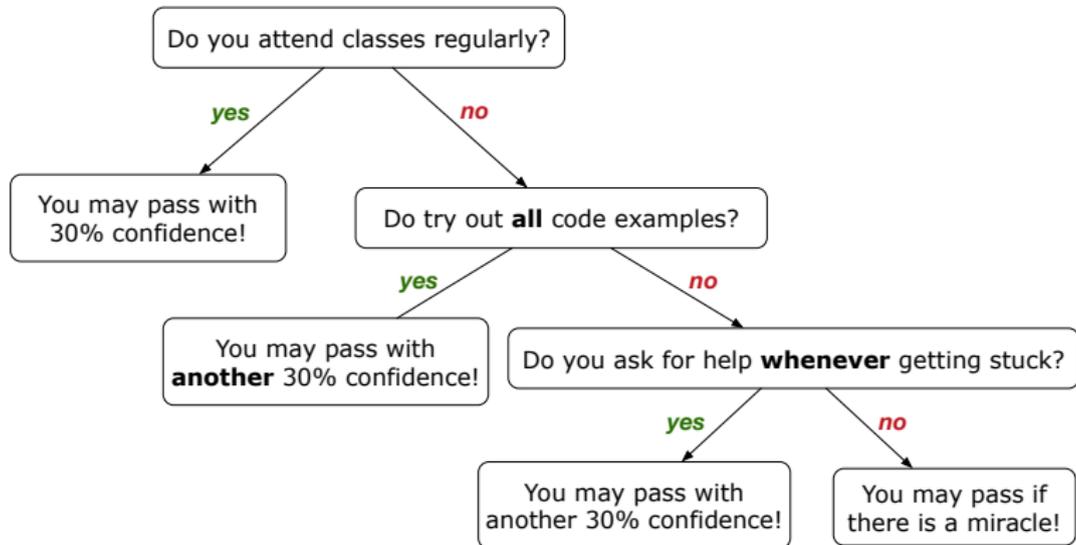
- Do something with root r .
- Recur on r 's **left subtree**. [strictly smaller problem]
- Recur on r 's **right subtree**. [strictly smaller problem]

Similar to how we **recur on subarrays** (by passing the `from` and `to` indices), we **recur on subtrees** by passing their roots (i.e., the current root's left child and right child).

Binary Trees: Application (1)

A **decision tree** is a binary tree used to express the decision-making process:

- Each *internal node* has two children (yes and no).
- Each *external node* represents a decision.



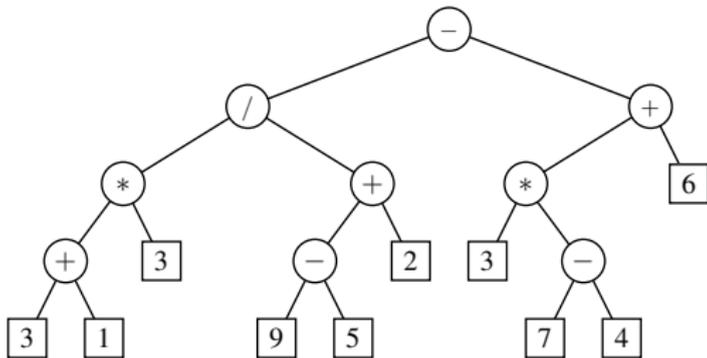
Binary Trees: Application (2)

An **arithmetic expression** can be represented using a binary tree:

- Each **internal node** denotes an operator (unary or binary).
- Each **external node** denotes an operand (i.e., a number).

e.g., Use a binary tree to represent the arithmetic expression

$$((3 + 1) * 3) / ((9 - 5) + 2) - (3 * (7 - 4)) + 6$$



- To print, or to evaluate, the expression that is represented by a binary tree, certain **traversal** over the entire tree is required.

Tree Traversal Algorithms: Definition

- A **traversal** of a tree T is a systematic way of visiting **all** the nodes of T .
- The visit of each node may be associated with an action: e.g.,
 - print the node element
 - determine if the node element satisfies certain property
 - accumulate the node element value to some global counter

Tree Traversal Algorithms: Common Types

- **Inorder:** Visit left subtree, then parent, then right subtree.

```
inorder (r): if (r != null) { /*subtree with root r is not empty*/  
  inorder (r's left child)  
  visit and act on the subtree rooted at r  
  inorder (r's right child) }
```

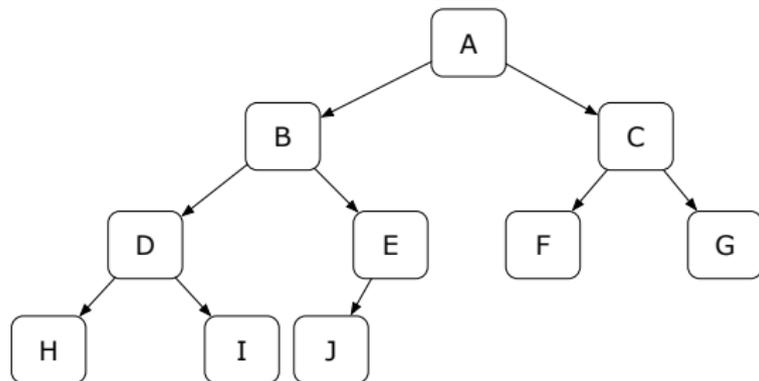
- **Preorder:** Visit parent, then left subtree, then right subtree.

```
preorder (r): if (r != null) { /*subtree with root r is not empty*/  
  visit and act on the subtree rooted at r  
  preorder (r's left child)  
  preorder (r's right child) }
```

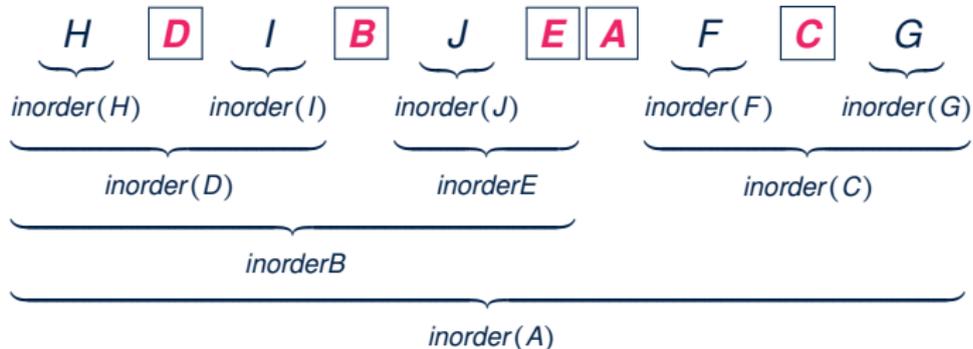
- **Postorder:** Visit left subtree, then right subtree, then parent.

```
postorder (r): if (r != null) { /*subtree with root r is not empty*/  
  postorder (r's left child)  
  postorder (r's right child)  
  visit and act on the subtree rooted at r }
```

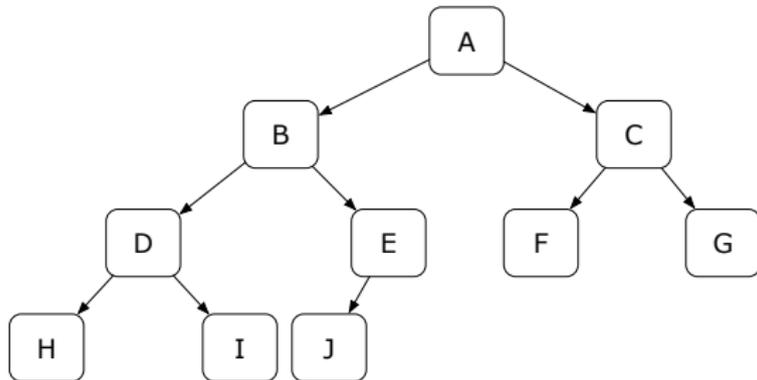
Tree Traversal: Inorder



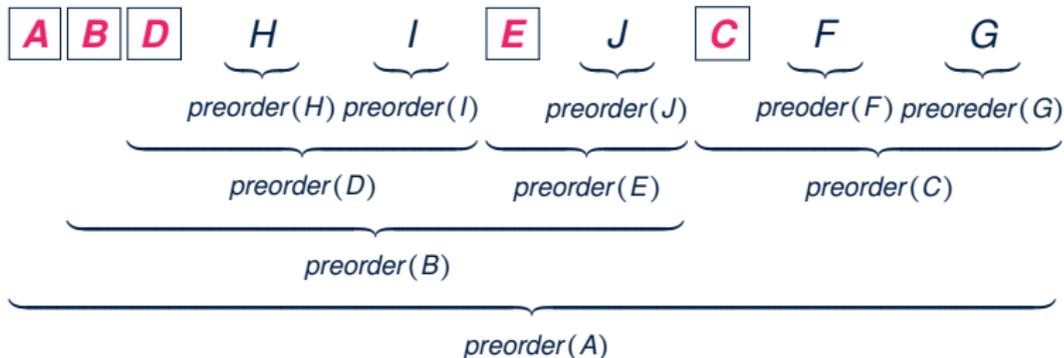
inorder traversal from the root A:



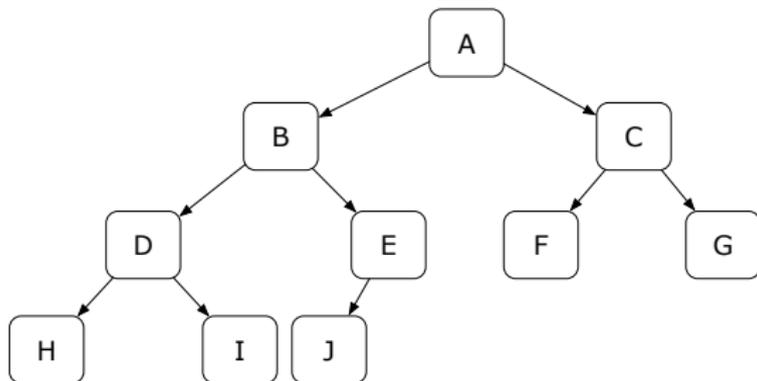
Tree Traversal: Preorder



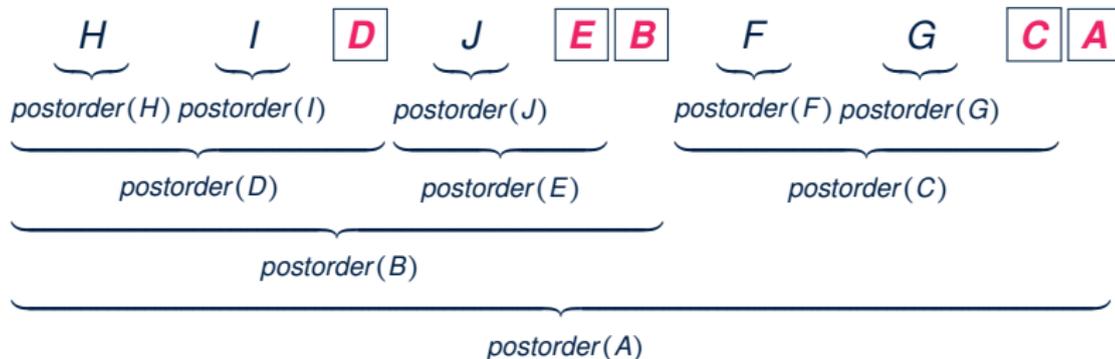
preorder traversal from the root A:



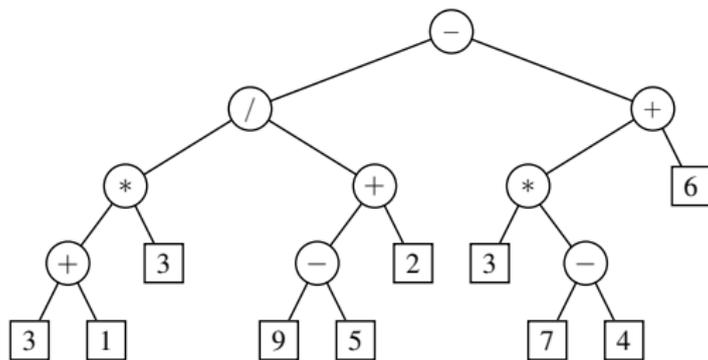
Tree Traversal: Postorder



postorder traversal from the root A:



Tree Traversal: Exercises



- **inorder** traversal from the root:
 $3+1*3/9-5+2-3*7-4+6$
- **preorder** traversal from the root:
 $-/ * + 3 1 3 + - 9 5 2 + * 3 - 7 4 6$
- **postorder** traversal from the root:
 $3 1 + 3 * 9 5 - 2 + / 3 7 4 - * 6 + -$

Binary Tree in Java: Linked Node

```
public class BTNode {  
    private String element;  
    private BTNode left;  
    private BTNode right;  
  
    BTNode(String element) {  
        this.element = element;  
    }  
  
    public String getElement() { return element; }  
    public BTNode getLeft() { return left; }  
    public BTNode getRight() { return right; }  
  
    public void setElement(String element) { this.element = element; }  
    public void setLeft(BTNode left) { this.left = left; }  
    public void setRight(BTNode right) { this.right = right; }  
}
```

Binary Tree in Java: Root Note

```
public class BinaryTree {
    private BTreeNode root;

    public BinaryTree() {
        /* Initialize an empty binary tree with root being null. */
    }

    public void setRoot(BTreeNode root) {
        this.root = root;
    }

    ...
}
```

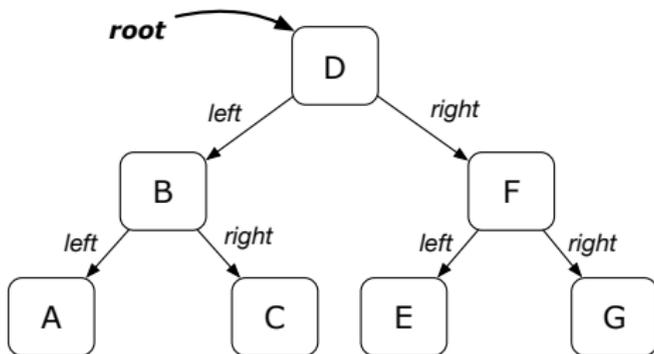
Binary Tree in Java: Adding Nodes (1)

```
public class BinaryTree {
    private BTreeNode root;
    public void addToLeft(BTreeNode n, String element) {
        if(n.getLeft() != null) {
            throw new IllegalArgumentException("Left is already there");
        }
        n.setLeft(new BTreeNode(element));
    }
    public void addToRight(BTreeNode n, String element) {
        if(n.getRight() != null) {
            throw new IllegalArgumentException("Right is already there");
        }
        n.setRight(new BTreeNode(element));
    }
}
```

- The way we implement the add methods is **not** recursive.
- These two add methods assume that the caller calls them by **passing references** of the **parent nodes**.

Binary Tree in Java: Adding Nodes (2)

Exercise: Write Java code to construct the following binary tree:



```

BinaryTree bt = new BinaryTree(); /* empty binary tree */
BTNode root = new BTNode("D"); /* node disconnected from BT */
bt.setRoot(root); /* node connected to BT */
bt.addToLeft(root, "B");
bt.addToRight(root, "F");
bt.addToLeft(root.getLeft(), "A");
bt.addToRight(root.getLeft(), "C");
bt.addToLeft(root.getRight(), "E");
bt.addToRight(root.getRight(), "G");
  
```

Binary Tree in Java: Counting Size (1)

Size of a tree rooted at r is 1 (counting r itself) plus the size of r 's left subtree and plus the size of r 's right subtree.

```
public class BinaryTree {
    private BTreeNode root;
    public int size() { return sizeHelper (root); }
    private int sizeHelper (BTreeNode root) {
        if (root == null) {
            return 0;
        }
        else {
            return
                1
            + sizeHelper (root.getLeft ())
            + sizeHelper (root.getRight ());
        }
    }
}
```

Binary Tree in Java: Counting Size (2)

```
@Test
public void testBTSize() {
    BinaryTree bt = new BinaryTree();
    assertEquals(0, bt.size());

    BTreeNode root = new BTreeNode("D");
    bt.setRoot(root);
    assertEquals(1, bt.size());

    bt.addToLeft(root, "B");
    bt.addToRight(root, "F");
    bt.addToLeft(root.getLeft(), "A");
    bt.addToRight(root.getLeft(), "C");
    bt.addToLeft(root.getRight(), "E");
    bt.addToRight(root.getRight(), "G");
    assertEquals(7, bt.size());
}
```

Binary Tree in Java: Membership (1)

An element e exists in a tree rooted at r if either r contains e ,
or r 's left subtree contains e , or r 's right subtree contains e .

```
public class BinaryTree {
    private BTNode root;

    public boolean has(String e) { return hasHelper (root, e); }
    private boolean hasHelper (BTNode root, String e) {
        if (root == null) {
            return false;
        }
        else {
            return
                root.getElement().equals(e)
                || hasHelper (root.getLeft(), e)
                || hasHelper (root.getRight(), e);
        }
    }
}
```

Binary Tree in Java: Membership (2)

```
@Test
public void testBTMembership() {
    BinaryTree bt = new BinaryTree();
    assertFalse(bt.has("D"));
    BTreeNode root = new BTreeNode("D");
    bt.setRoot(root);
    assertTrue(bt.has("D"));
    assertFalse(bt.has("A"));
    bt.addToLeft(root, "B");
    bt.addToRight(root, "F");
    bt.addToLeft(root.getLeft(), "A");
    bt.addToRight(root.getLeft(), "C");
    bt.addToLeft(root.getRight(), "E");
    bt.addToRight(root.getRight(), "G");
    assertTrue(bt.has("A")); assertTrue(bt.has("B"));
    assertTrue(bt.has("C")); assertTrue(bt.has("D"));
    assertTrue(bt.has("E")); assertTrue(bt.has("F"));
    assertTrue(bt.has("G"));
    assertFalse(bt.has("H"));
    assertFalse(bt.has("I"));
}
```

Binary Tree in Java: Inorder Traversal (1)

```
public class BinaryTree {
    private BTreeNode root;

    public ArrayList<String> inoder() {
        ArrayList<String> list = new ArrayList<>();
        inorderHelper (root, list);
        return list;
    }
    private void inorderHelper (BTreeNode root, ArrayList<String> list) {
        if (root != null) {
            inorderHelper (root.getLeft(), list);
            list.add(root.getElement());
            inorderHelper (root.getRight(), list);
        }
    }
}
```

Binary Tree in Java: Inorder Traversal (2)

```
@Test
public void testBT_inorder() {
    BinaryTree bt = new BinaryTree();
    BTNode root = new BTNode("D");
    bt.setRoot(root);
    bt.addToLeft(root, "B");
    bt.addToRight(root, "F");
    bt.addToLeft(root.getLeft(), "A");
    bt.addToRight(root.getLeft(), "C");
    bt.addToLeft(root.getRight(), "E");
    bt.addToRight(root.getRight(), "G");
    ArrayList<String> list = bt.inroder();
    assertEquals(list.get(0), "A");
    assertEquals(list.get(1), "B");
    assertEquals(list.get(2), "C");
    assertEquals(list.get(3), "D");
    assertEquals(list.get(4), "E");
    assertEquals(list.get(5), "F");
    assertEquals(list.get(6), "G");
}
```

Binary Tree in Java: Preorder Traversal (1)

```
public class BinaryTree {
    private BTreeNode root;

    public ArrayList<String> preorder() {
        ArrayList<String> list = new ArrayList<>();
        preorderHelper (root, list);
        return list;
    }
    private void preorderHelper (BTreeNode root, ArrayList<String> list) {
        if (root != null) {
            list.add(root.getElement());
            preorderHelper (root.getLeft(), list);
            preorderHelper (root.getRight(), list);
        }
    }
}
```

Binary Tree in Java: Preorder Traversal (2)

```
@Test
public void testBT_inorder() {
    BinaryTree bt = new BinaryTree();
    BTNode root = new BTNode("D");
    bt.setRoot(root);
    bt.addToLeft(root, "B");
    bt.addToRight(root, "F");
    bt.addToLeft(root.getLeft(), "A");
    bt.addToRight(root.getLeft(), "C");
    bt.addToLeft(root.getRight(), "E");
    bt.addToRight(root.getRight(), "G");
    ArrayList<String> list = bt.preorder();
    assertEquals(list.get(0), "D");
    assertEquals(list.get(1), "B");
    assertEquals(list.get(2), "A");
    assertEquals(list.get(3), "C");
    assertEquals(list.get(4), "F");
    assertEquals(list.get(5), "E");
    assertEquals(list.get(6), "G");
}
```

Binary Tree in Java: Postorder Traversal (1)

```
public class BinaryTree {
    private BTreeNode root;

    public ArrayList<String> preorder() {
        ArrayList<String> list = new ArrayList<>();
        postorderHelper (root, list);
        return list;
    }
    private void postorderHelper (BTreeNode root, ArrayList<String> list) {
        if (root != null) {
            list.add(root.getElement());
            postorderHelper (root.getLeft(), list);
            postorderHelper (root.getRight(), list);
        }
    }
}
```

Binary Tree in Java: Postorder Traversal (2)

```
@Test
public void testBT_inorder() {
    BinaryTree bt = new BinaryTree();
    BTNode root = new BTNode("D");
    bt.setRoot(root);
    bt.addToLeft(root, "B");
    bt.addToRight(root, "F");
    bt.addToLeft(root.getLeft(), "A");
    bt.addToRight(root.getLeft(), "C");
    bt.addToLeft(root.getRight(), "E");
    bt.addToRight(root.getRight(), "G");
    ArrayList<String> list = bt.postorder();
    assertEquals(list.get(0), "A");
    assertEquals(list.get(1), "C");
    assertEquals(list.get(2), "B");
    assertEquals(list.get(3), "E");
    assertEquals(list.get(4), "G");
    assertEquals(list.get(5), "F");
    assertEquals(list.get(6), "D");
}
```

Index (1)

General Trees

General Trees: Terminology (1)

General Trees: Terminology (2)

General Trees: Terminology (3)

Exercise: Identifying Subtrees

General Tree: Important Characteristics

General Trees: Ordered Trees

General Trees: Unordered Trees

Binary Trees

Binary Trees: Terminology (1)

Binary Trees: Recursive Definition

Binary Trees: Application (1)

Binary Trees: Application (2)

Tree Traversal Algorithms: Definition

Index (2)

Tree Traversal Algorithms: Common Types

Tree Traversal: Inorder

Tree Traversal: Preorder

Tree Traversal: Postorder

Tree Traversal: Exercises

Binary Tree in Java: Linked Node

Binary Tree in Java: Root Node

Binary Tree in Java: Adding Nodes (1)

Binary Tree in Java: Adding Nodes (2)

Binary Tree in Java: Counting Size (1)

Binary Tree in Java: Counting Size (2)

Binary Tree in Java: Membership (1)

Binary Tree in Java: Membership (2)

Binary Tree in Java: Inorder Traversal (1)

Index (3)

Binary Tree in Java: Inorder Traversal (2)

Binary Tree in Java: Preorder Traversal (1)

Binary Tree in Java: Preorder Traversal (2)

Binary Tree in Java: Postorder Traversal (1)

Binary Tree in Java: Postorder Traversal (2)

Inheritance



EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Why Inheritance: A Motivating Example

Problem: A *student management system* stores data about students. There are two kinds of university students: *resident* students and *non-resident* students. Both kinds of students have a *name* and a list of *registered courses*. Both kinds of students are restricted to *register* for no more than 10 courses. When *calculating the tuition* for a student, a base amount is first determined from the list of courses they are currently registered (each course has an associated fee). For a non-resident student, there is a *discount rate* applied to the base amount to waive the fee for on-campus accommodation. For a resident student, there is a *premium rate* applied to the base amount to account for the fee for on-campus accommodation and meals.

Tasks: Write Java classes that satisfy the above problem statement. At runtime, each type of student must be able to register a course and calculate their tuition fee.

No Inheritance: ResidentStudent Class

```
class ResidentStudent {
    String name;
    Course[] registeredCourses;
    int numberOfCourses;
    double premiumRate; /* there's a mutator method for this */
    ResidentStudent (String name) {
        this.name = name;
        registeredCourses = new Course[10];
    }
    void register(Course c) {
        registeredCourses[numberOfCourses] = c;
        numberOfCourses ++;
    }
    double getTuition() {
        double tuition = 0;
        for(int i = 0; i < numberOfCourses; i ++ ) {
            tuition += registeredCourses[i].fee;
        }
        return tuition * premiumRate ;
    }
}
```

No Inheritance: NonResidentStudent Class

```
class NonResidentStudent {
    String name;
    Course[] registeredCourses;
    int numberOfCourses;
    double discountRate; /* there's a mutator method for this */
    NonResidentStudent (String name) {
        this.name = name;
        registeredCourses = new Course[10];
    }
    void register(Course c) {
        registeredCourses[numberOfCourses] = c;
        numberOfCourses ++;
    }
    double getTuition() {
        double tuition = 0;
        for(int i = 0; i < numberOfCourses; i ++ ) {
            tuition += registeredCourses[i].fee;
        }
        return tuition * discountRate ;
    }
}
```

No Inheritance: Testing Student Classes

```
class Course {
    String title;
    double fee;
    Course(String title, double fee) {
        this.title = title; this.fee = fee; } }

```

```
class StudentTester {
    static void main(String[] args) {
        Course c1 = new Course("EECS2030", 500.00); /* title and fee */
        Course c2 = new Course("EECS3311", 500.00); /* title and fee */
        ResidentStudent jim = new ResidentStudent("J. Davis");
        jim.setPremiumRate(1.25);
        jim.register(c1); jim.register(c2);
        NonResidentStudent jeremy = new NonResidentStudent("J. Gibbons");
        jeremy.setDiscountRate(0.75);
        jeremy.register(c1); jeremy.register(c2);
        System.out.println("Jim pays " + jim.getTuition());
        System.out.println("Jeremy pays " + jeremy.getTuition());
    }
}

```

No Inheritance: Issues with the Student Classes

- Implementations for the two student classes seem to work. But can you see any potential problems with it?
- The code of the two student classes share a lot in common.
- *Duplicates of code make it hard to maintain your software!*
- This means that when there is a change of policy on the common part, we need modify *more than one places*.

No Inheritance: Maintainability of Code (1)

What if the way for registering a course changes?

e.g.,

```
void register(Course c) {  
    if (numberOfCourses >= MAX_ALLOWANCE) {  
        throw new IllegalArgumentException("Maximum allowance reached.");  
    }  
    else {  
        registeredCourses[numberOfCourses] = c;  
        numberOfCourses++;  
    }  
}
```

We need to change the `register` method in *both* student classes!

No Inheritance: Maintainability of Code (2)

What if the way for calculating the base tuition changes?

e.g.,

```
double getTuition() {  
    double tuition = 0;  
    for(int i = 0; i < numberOfCourses; i++) {  
        tuition += registeredCourses[i].fee;  
    }  
    /* ... can be premiumRate or discountRate */  
    return tuition * inflationRate * ...;  
}
```

We need to change the `getTuition` method in *both* student classes.

No Inheritance: A Collection of Various Kinds of Students

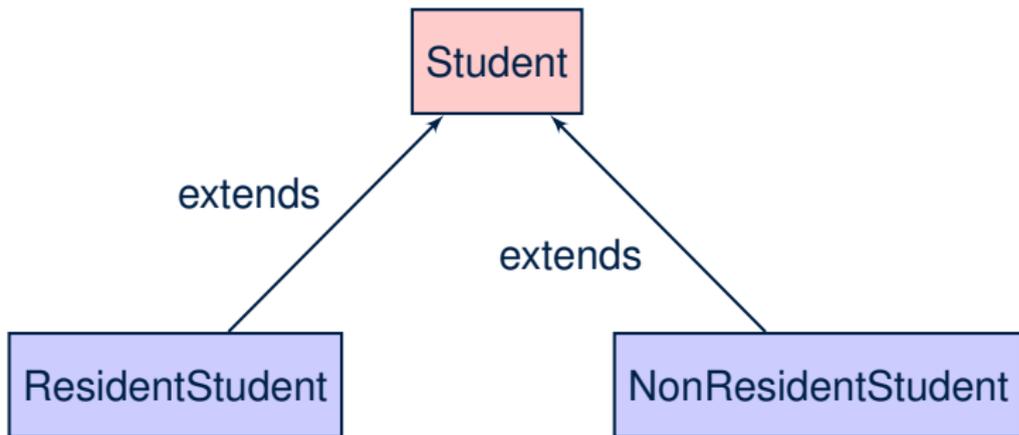
How do you define a class `StudentManagementSystem` that contains a list of *resident* and *non-resident* students?

```
class StudentManagementSystem {
    ResidentStudent[] rss;
    NonResidentStudent[] nrss;
    int nors; /* number of resident students */
    int nonrs; /* number of non-resident students */
    void addRS (ResidentStudent rs){ rss[nors]=rs; nors++; }
    void addNRS (NonResidentStudent nrs){ nrss[nonrs]=nrs; nonrs++; }
    void registerAll (Course c) {
        for(int i = 0; i < nors; i ++){ rss[i].register(c); }
        for(int i = 0; i < nonrs; i ++){ nrss[i].register(c); }
    }
}
```

But what if we later on introduce *more kinds of students*?

Very *inconvenient* to handle each list of students *separately*!

Inheritance Architecture



Inheritance: The Student Parent/Super Class

```
class Student {
    String name;
    Course[] registeredCourses;
    int numberOfCourses;

    Student (String name) {
        this.name = name;
        registeredCourses = new Course[10];
    }
    void register(Course c) {
        registeredCourses[numberOfCourses] = c;
        numberOfCourses ++;
    }
    double getTuition() {
        double tuition = 0;
        for(int i = 0; i < numberOfCourses; i ++) {
            tuition += registeredCourses[i].fee;
        }
        return tuition; /* base amount only */
    }
}
```

Inheritance:

The ResidentStudent Child/Sub Class

```
1 class ResidentStudent extends Student {
2     double premiumRate; /* there's a mutator method for this */
3     ResidentStudent (String name) { super(name); }
4     /* register method is inherited */
5     double getTuition() {
6         double base = super.getTuition();
7         return base * premiumRate;
8     }
9 }
```

- L1 declares that ResidentStudent inherits all attributes and methods (except constructors) from Student.
- There is no need to repeat the register method
- Use of *super* in L4 is as if calling Student (name)
- Use of *super* in L8 returns what getTuition() in Student returns.
- Use *super* to refer to attributes/methods defined in the super class:

`super.name`, `super.register(c)`.

Inheritance:

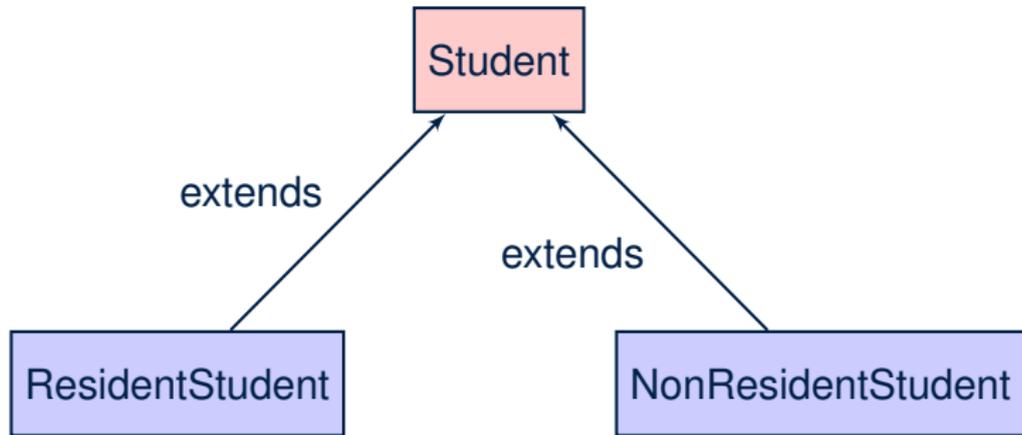
The NonResidentStudent Child/Sub Class

```
1 class NonResidentStudent extends Student {
2     double discountRate; /* there's a mutator method for this */
3     NonResidentStudent (String name) { super(name); }
4     /* register method is inherited */
5     double getTuition() {
6         double base = super.getTuition();
7         return base * discountRate;
8     }
9 }
```

- L1 declares that NonResidentStudent inherits all attributes and methods (except constructors) from Student.
- There is no need to repeat the register method
- Use of *super* in L4 is as if calling Student (name)
- Use of *super* in L8 returns what getTuition() in Student returns.
- Use *super* to refer to attributes/methods defined in the super class:

`super.name`, `super.register(c)`.

Inheritance Architecture Revisited



- The class that defines the common attributes and methods is called the **parent** or **super** class.
- Each “extended” class is called a **child** or **sub** class.

Visualizing Parent/Child Objects (1)

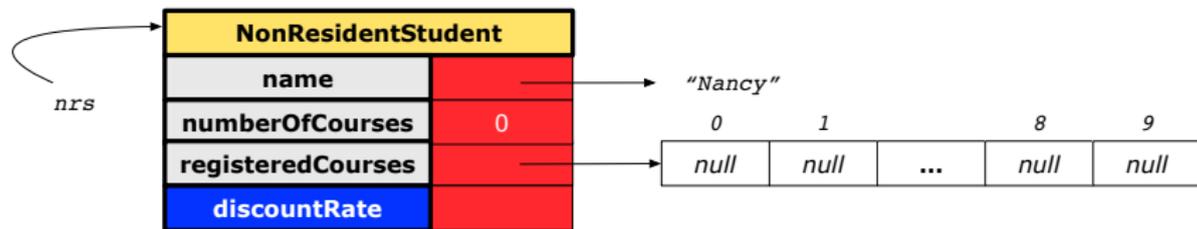
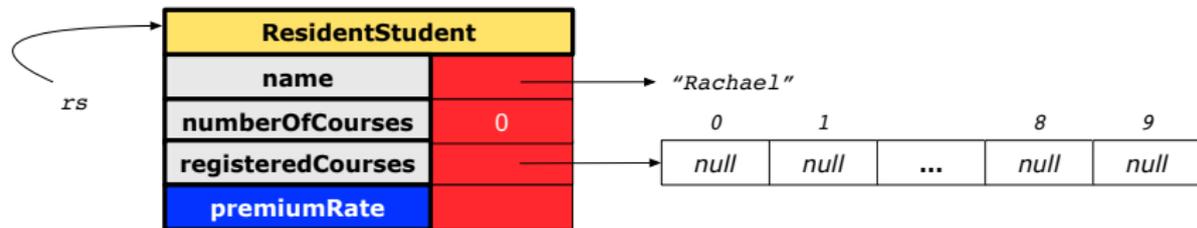
- A child class inherits **all** attributes from its parent class.
⇒ A child instance has **at least as many** attributes as an instance of its parent class.

Consider the following instantiations:

```
Student s = new Student("Stella");  
ResidentStudent rs = new ResidentStudent("Rachael");  
NonResidentStudent nrs = new NonResidentStudent("Nancy");
```

- How will these initial objects look like?

Visualizing Parent/Child Objects (2)

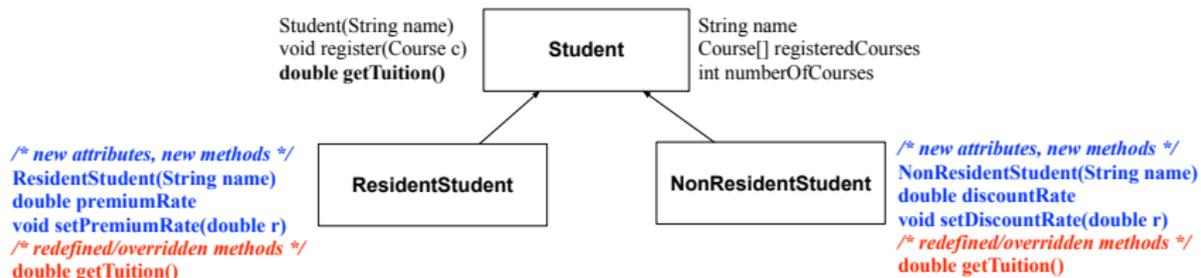


Using Inheritance for Code Reuse

Inheritance in Java allows you to:

- Define *common attributes and methods* in a separate class.
e.g., the `Student` class
- Define an “extended” version of the class which:
 - *inherits* definitions of all attributes and methods
e.g., `name`, `registeredCourses`, `numberOfCourses`
e.g., `register`
e.g., base amount calculation in `getTuition`
This means code reuse and elimination of code duplicates!
 - *defines new* attributes and methods if necessary
e.g., `setPremiumRate` for `ResidentStudent`
e.g., `setDiscountRate` for `NonResidentStudent`
 - *redefines/overrides* methods if necessary
e.g., compounded tuition for `ResidentStudent`
e.g., discounted tuition for `NonResidentStudent`

Inheritance Architecture Revisited



```

Student s = new Student("Stella");
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
    
```

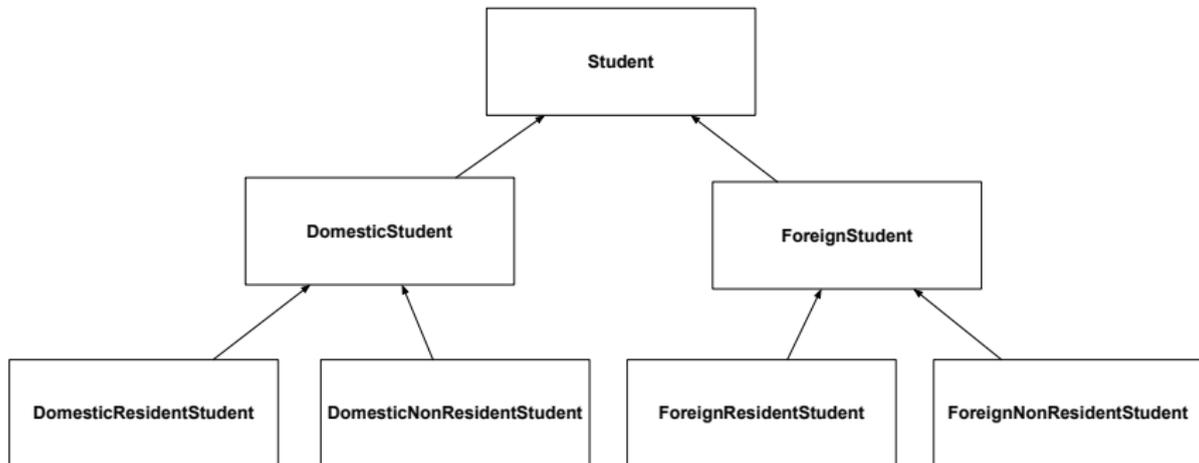
	name	rCs	noC	reg	getT	pr	setPR	dr	setDR
s.			✓					×	
rs.			✓				✓		×
nrs.			✓				×		✓

Testing the Two Student Sub-Classes

```
class StudentTester {
    static void main(String[] args) {
        Course c1 = new Course("EECS2030", 500.00); /* title and fee */
        Course c2 = new Course("EECS3311", 500.00); /* title and fee */
        ResidentStudent jim = new ResidentStudent("J. Davis");
        jim.setPremiumRate(1.25);
        jim.register(c1); jim.register(c2);
        NonResidentStudent jeremy = new NonResidentStudent("J. Gibbons");
        jeremy.setDiscountRate(0.75);
        jeremy.register(c1); jeremy.register(c2);
        System.out.println("Jim pays " + jim.getTuition());
        System.out.println("Jeremy pays " + jeremy.getTuition());
    }
}
```

- The software can be used in exactly the same way as before (because we did not modify *method signatures*).
- But now the internal structure of code has been made *maintainable* using *inheritance*.

Multi-Level Inheritance Architecture



Root of the Java Class Hierarchy

- Implicitly:
 - Every class is a *child/sub* class of the `Object` class.
 - The `Object` class is the *parent/super* class of every class.
- There are two useful *accessor methods* that every class *inherits* from the `Object` class:
 - `boolean equals(Object other)`
Indicates whether some other object is “equal to” this one.
 - The default definition inherited from `Object`:

```
boolean equals(Object other) {  
    return (this == other);  
}
```
 - `String toString()`
Returns a string representation of the object.
- Very often when you define new classes, you want to *redefine* / *override* the inherited definitions of `equals` and `toString`.

Behaviour of the Inherited equals Method (1)

Problem: Define equals method for the Rectangle class

```
class Rectangle{
    double width; double length;
    double getArea() { return width * length; } }
```

and the RectangleCollector class

```
class RectangleCollector{
    Rectangle[] rectangles;
    final int MAX = 100;
    int nor; /* number of rectangles */
    RectangleCollector() { rectangles = new Rectangle[ MAX ]; }
    addRectangle(Rectangle c) { rectangles[ nor ] = c; nor++; }
}
```

Two rectangles are *equal* if their areas are *equal*.

Two rectangle collectors are *equal* if rectangles they contain are *equal* .

Behaviour of the Inherited equals Method (2)

```

1 class RectangleCollectorTester {
2     Rectangle r1 = new Rectangle(3, 6);
3     Rectangle r2 = new Rectangle(2, 9);
4     System.out.println(r1 == r2); /* false */
5     System.out.println(r1.equals(r2)); /* false */
6     RectangleCollector rc1 = new RectangleCollector();
7     rc1.addRectangle(r1);
8     RectangleCollector rc2 = new RectangleCollector();
9     rc2.addRectangle(r2);
10    System.out.println(rc1 == rc2); /* false */
11    System.out.println(rc1.equals(rc2)); /* false */
12 }

```

- Lines 5 and 11 return *false* because we have not **explicitly** redefined/overridden the `equals` method inherited from the `Object` class (which compares addresses by default).
- We need to **redefine / override** the inherited `equals` method in both `Rectangle` and `RectangleCollector`.

Behaviour of the Inherited equals Method (3)

Two rectangles are equal if their areas are equal:

```
class Rectangle{
    double width;
    double length;
    getArea() { ... }
    boolean equals(Object obj) {
        if(this == obj) {
            return true;
        }
        if(obj == null || this.getClass() != obj.getClass()) {
            return false;
        }
        Rectangle other = (Rectangle) obj;
        return getArea() == other.getArea();
    }
}
```

Behaviour of the Inherited equals Method (4)

Rectangle collectors are equal if rectangles collected are equal:

```

1 class RectangleCollector{
2     /* rectangles, RectangleCollector(), nor, addRectangle */
3     boolean equals (Object obj) {
4         if(this == obj) {
5             return true;
6         }
7         if(obj == null || this.getClass() != obj.getClass()) {
8             return false;
9         }
10        RectangleCollector other = (RectangleCollector) obj;
11        boolean soFarEqual = this.nor == other.nor;
12        for(int i = 0; soFarEqual && i < this.nor; i++) {
13            soFarEqual =
14                this.rectangles[i].equals (other.rectangles[i]);
15        }
16        return soFarEqual;
17    }
18 }

```

Behaviour of the Inherited equals Method (5)

Now that we have *redefined* / *overridden* the equals method, inherited from the Object class, in both Rectangle and RectangleCollector, the test results shall be different!

```
class RectangleCollectorTester {
    Rectangle r1 = new Rectangle(3, 6);
    Rectangle r2 = new Rectangle(2, 9);
    System.out.println(r1 == r2); /* false */
    System.out.println(r1.equals(r2)); /* true */
    RectangleCollector rc1 = new RectangleCollector();
    rc1.addRectangle(r1);
    RectangleCollector rc2 = new RectangleCollector();
    rc2.addRectangle(r2);
    System.out.println(rc1 == rc2); /* false */
    System.out.println(rc1.equals(rc2)); /* true */
}
```

Behaviour of Inherited toString Method (1)

```
Point p1 = new Point(2, 4);  
System.out.println(p1);
```

```
Point@677327b6
```

- Implicitly, the toString method is called inside the println method.
- By default, the address stored in p1 gets printed.
- We need to **redefine** / **override** the toString method, inherited from the Object class, in the Point class.

Behaviour of Inherited toString Method (2)

```
class Point {  
    double x;  
    double y;  
    public String toString() {  
        return "(" + this.x + ", " + this.y + ")";  
    }  
}
```

After redefining/overriding the toString method:

```
Point p1 = new Point(2, 4);  
System.out.println(p1);
```

(2, 4)

Behaviour of Inherited `toString` Method (3)



Exercise: Override the `toString` method for the `Rectangle` and `RectangleCollector` classes.

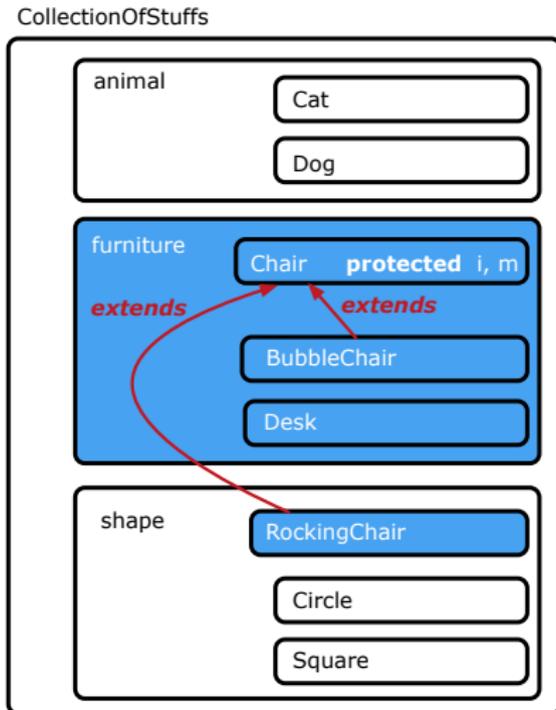
Exercise: Override the `equals` and `toString` methods for the `ResidentStudent` and `NonResidentStudent` classes.

Use of the protected Modifier

- *private* attributes are not inherited to subclasses.
- package-level attributes (i.e., with **no modifier**) and project-level attributes (i.e., *public*) are inherited.
- What if we want attributes to be:
 - *visible* to sub-classes outside the current package, but still
 - *invisible* to other non-sub-classes outside the current package?

Use `protected`!

Visibility of Attr./Meth.: Across All Methods Same Package and Sub-Classes (protected)



Visibility of Attributes/Methods

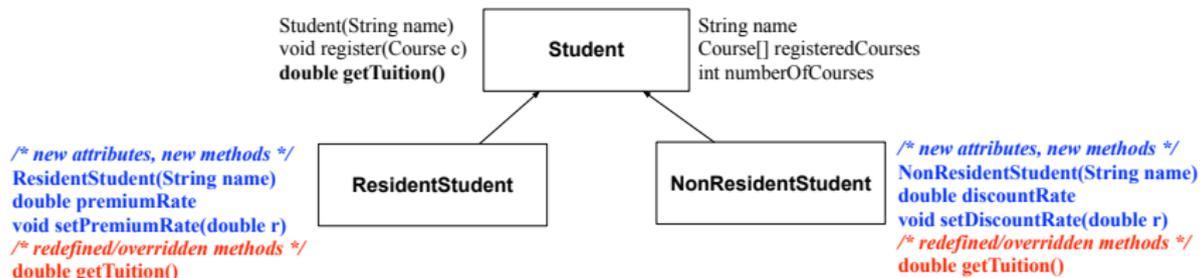
scope modifier	CLASS	PACKAGE	SUBCLASS (same pkg)	SUBCLASS (different pkg)	PROJECT
public	Green	Green	Green	Green	Green
protected	Green	Green	Green	Green	Red
no modifier	Green	Green	Green	Red	Red
private	Green	Red	Red	Red	Red

For the rest of this lecture, for simplicity, we assume that:

All relevant descendant classes are in the same package .

⇒ Attributes with **no modifiers** (package-level visibility) suffice.

Inheritance Architecture Revisited

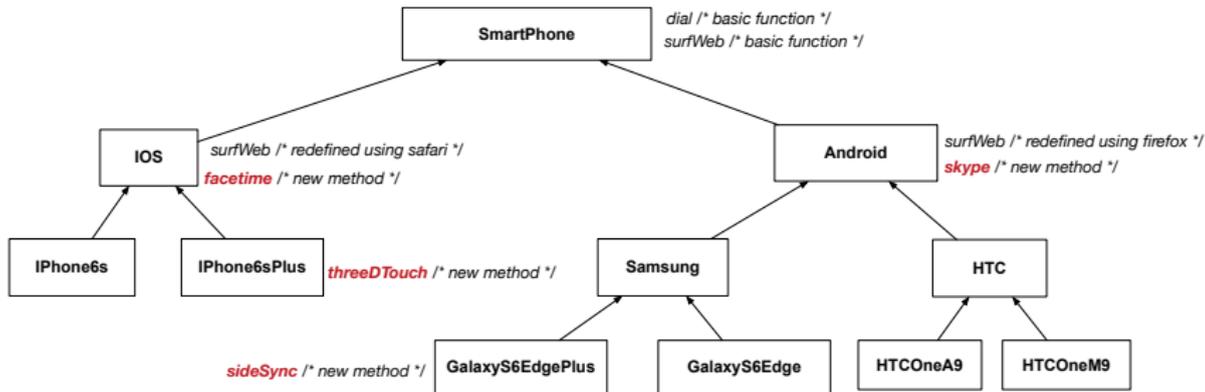


```

Student s = new Student("Stella");
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
    
```

	name	rCs	noC	reg	getT	pr	setPR	dr	setDR
s.			✓					×	
rs.			✓			✓			×
nrs.			✓				×		✓

Multi-Level Inheritance Hierarchy: Smart Phones



Polymorphism: Intuition (1)

```

1 Student s = new Student("Stella");
2 ResidentStudent rs = new ResidentStudent("Rachael");
3 rs.setPremiumRate(1.25);
4 s = rs; /* Is this valid? */
5 rs = s; /* Is this valid? */

```

- Which one of **L4** and **L5** is *valid*? Which one is *invalid*?

- **Hints:**

- **L1:** What *kind* of address can *s* store? [Student]
 ∴ The context object *s* is *expected* to be used as:
 - *s*.register(eecs2030) and *s*.getTuition()
- **L2:** What *kind* of address can *rs* store? [ResidentStudent]
 ∴ The context object *rs* is *expected* to be used as:
 - *rs*.register(eecs2030) and *rs*.getTuition()
 - *rs.setPremiumRate(1.50)* [increase premium rate]

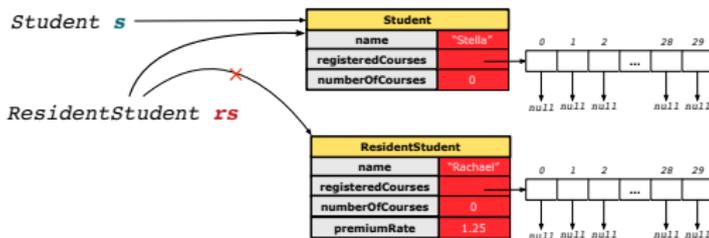
Polymorphism: Intuition (2)

```

1 Student s = new Student("Stella");
2 ResidentStudent rs = new ResidentStudent("Rachael");
3 rs.setPremiumRate(1.25);
4 s = rs; /* Is this valid? */
5 rs = s; /* Is this valid? */

```

- rs = s (L5)** should be *invalid*:



- Since **rs** is declared of type **ResidentStudent**, a subsequent call `rs.setPremiumRate(1.50)` can be expected.
- rs** is now pointing to a **Student** object.
- Then, what would happen to `rs.setPremiumRate(1.50)`?

CRASH

\therefore `rs.premiumRate` is *undefined*!!

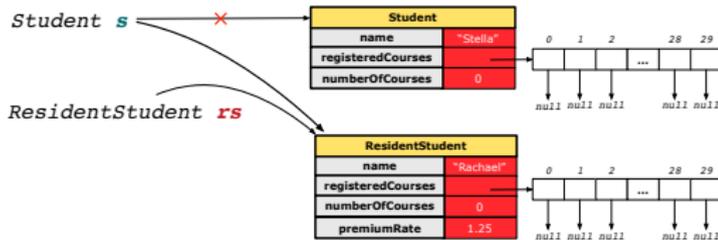
Polymorphism: Intuition (3)

```

1 Student s = new Student("Stella");
2 ResidentStudent rs = new ResidentStudent("Rachael");
3 rs.setPremiumRate(1.25);
4 s = rs; /* Is this valid? */
5 rs = s; /* Is this valid? */

```

- $s = rs$ (L4) should be *valid*:



- Since s is declared of type `Student`, a subsequent call $s.setPremiumRate(1.50)$ is *never* expected.
- s is now pointing to a `ResidentStudent` object.
- Then, what would happen to $s.getTuition()$?

OK

$\therefore s.premiumRate$ is just *never used*!!

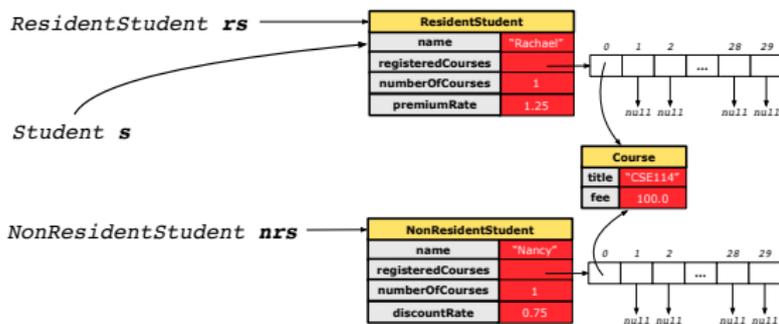
Dynamic Binding: Intuition (1)

```

1 Course eecs2030 = new Course("EECS2030", 100.0);
2 Student s;
3 ResidentStudent rs = new ResidentStudent("Rachael");
4 NonResidentStudent nrs = new NonResidentStudent("Nancy");
5 rs.setPremiumRate(1.25); rs.register(eecs2030);
6 nrs.setDiscountRate(0.75); nrs.register(eecs2030);
7 s = rs; System.out.println(s.getTuition()); /* output: 125.0 */
8 s = nrs; System.out.println(s.getTuition()); /* output: 75.0 */

```

After `s = rs` (L7), `s` points to a ResidentStudent object.
 ⇒ Calling `s.getTuition()` applies the premiumRate.



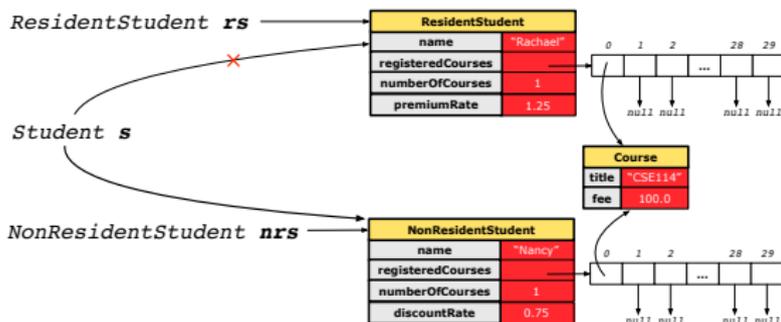
Dynamic Binding: Intuition (2)

```

1 Course eecs2030 = new Course("EECS2030", 100.0);
2 Student s;
3 ResidentStudent rs = new ResidentStudent("Rachael");
4 NonResidentStudent nrs = new NonResidentStudent("Nancy");
5 rs.setPremiumRate(1.25); rs.register(eecs2030);
6 nrs.setDiscountRate(0.75); nrs.register(eecs2030);
7 s = rs; System.out.println(s.getTuition()); /* output: 125.0 */
8 s = nrs; System.out.println(s.getTuition()); /* output: 75.0 */

```

After `s = nrs` (L8), `s` points to a `NonResidentStudent` object.
 ⇒ Calling `s.getTuition()` applies the `discountRate`.



Inheritance Forms a Type Hierarchy

- A (data) **type** denotes a set of related *runtime values*.
 - Every *class* can be used as a type: the set of runtime *objects*.
- Use of *inheritance* creates a **hierarchy** of classes:
 - (Implicit) Root of the hierarchy is `Object`.
 - Each `extends` declaration corresponds to an upward arrow.
 - The `extends` relationship is *transitive*: when A extends B and B extends C, we say A *indirectly* extends C.
e.g., Every class implicitly `extends` the `Object` class.
- **Ancestor** vs. **Descendant** classes:
 - The **ancestor classes** of a class A are: A itself and all classes that A directly, or indirectly, extends.
 - A inherits all code (attributes and methods) from its *ancestor classes*.
∴ A's instances have a **wider range of expected usages** (i.e., attributes and methods) than instances of its *ancestor* classes.
 - The **descendant classes** of a class A are: A itself and all classes that directly, or indirectly, extends A.
 - Code defined in A is inherited to all its *descendant classes*.

Inheritance Accumulates Code for Reuse

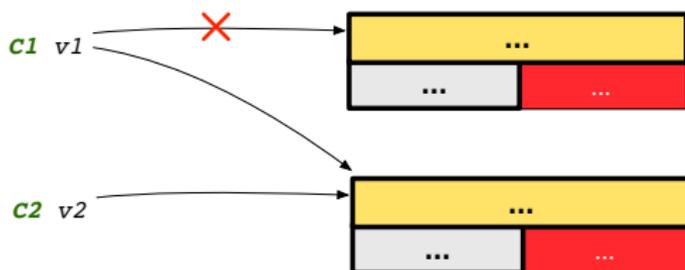
- The *lower* a class is in the type hierarchy, the *more code* it accumulates from its *ancestor classes*:
 - A *descendant class* inherits all code from its *ancestor classes*.
 - A *descendant class* may also:
 - Declare new attributes
 - Define new methods
 - **Redefine / Override** inherited methods
- Consequently:
 - When being used as **context objects**, instances of a class' *descendant classes* have a *wider range of expected usages* (i.e., attributes and methods).
 - When expecting an object of a particular class, we may **substitute** it with an object of any of its *descendant classes*.
 - e.g., When expecting a `Student` object, we may substitute it with either a `ResidentStudent` or a `NonResidentStudent` object.
 - **Justification:** A *descendant class* contains **at least as many** methods as defined in its *ancestor classes* (but not vice versa!).

Reference Variable: Static Type

- A reference variable's **static type** is what we declare it to be.
 - `Student jim` declares jim's ST as Student.
 - `SmartPhone myPhone` declares myPhone's ST as SmartPhone.
 - The **static type** of a reference variable **never changes**.
- For a reference variable v , its **static type** C defines the **expected usages of v as a context object**.
- A method call $v.m(\dots)$ is **compilable** if m is defined in C .
 - e.g., After declaring `Student jim`, we
 - **may** call `register` and `getTuition` on `jim`
 - **may not** call `setPremiumRate` (specific to a resident student) or `setDiscountRate` (specific to a non-resident student) on `jim`
 - e.g., After declaring `SmartPhone myPhone`, we
 - **may** call `dial` and `surfWeb` on `myPhone`
 - **may not** call `facetime` (specific to an IOS phone) or `skype` (specific to an Android phone) on `myPhone`

Substitutions via Assignments

- By declaring $c1$ $v1$, *reference variable* $v1$ will store the *address* of an object “of class $C1$ ” at runtime.
- By declaring $c2$ $v2$, *reference variable* $v2$ will store the *address* of an object “of class $C2$ ” at runtime.
- Assignment $v1 = v2$ *copies address* stored in $v2$ into $v1$.
 - $v1$ will instead point to wherever $v2$ is pointing to. [*object alias*]



- In such assignment $v1 = v2$, we say that we *substitute* an object of (*static*) type $C1$ by an object of (*static*) type $C2$.
- Substitutions* are subject to *rules*!

Rules of Substitution

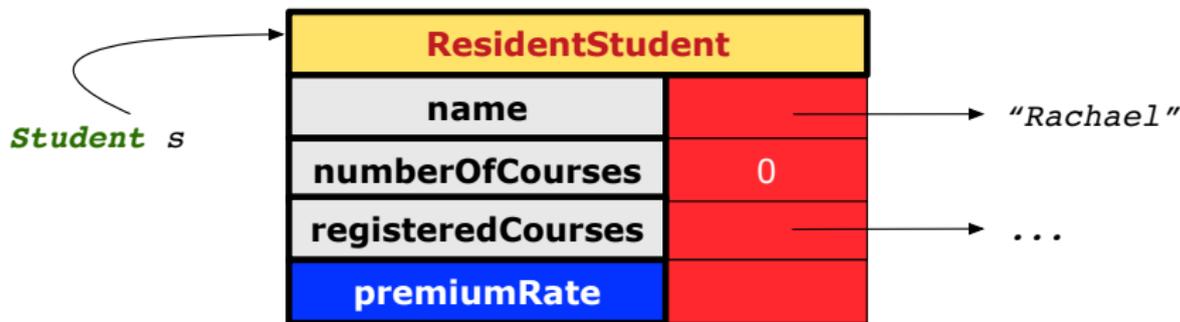
1. When expecting an object of **static type** A, it is *safe* to **substitute** it with an object whose **static type** is any of the **descendant class** of A (including A).
 - ∴ Each **descendant class** of A is guaranteed to contain code for all (non-private) attributes and methods that are defined in A.
 - ∴ All attributes and methods defined in A are *guaranteed to be available* in the new substitute.
 - e.g., When expecting an IOS phone, you *can* substitute it with either an iPhone6s or iPhone6sPlus.
2. When expecting an object of **static type** A, it is *unsafe* to **substitute** it with an object whose **static type** is any of the **ancestor classes of A's parent** (excluding A).
 - ∴ Class A may have defined new methods that do not exist in any of its **parent's ancestor classes**.
 - e.g., When expecting IOS phone, *unsafe* to substitute it with a SmartPhone ∴ facetime not supported in Android phone.

Reference Variable: Dynamic Type

A *reference variable's* **dynamic type** is the type of object that it is currently pointing to at runtime.

- The *dynamic type* of a reference variable *may change* whenever we **re-assign** that variable to a different object.
- There are two ways to re-assigning a reference variable.

Visualizing Static Type vs. Dynamic Type



- Each segmented box denotes a *runtime* object.
- Arrow denotes a variable (e.g., `s`) storing the object's address. Usually, when the context is clear, we leave the variable's *static type* implicit (`Student`).
- Title of box indicates type of runtime object, which denotes the *dynamic type* of the variable (`ResidentStudent`).

Reference Variable: Changing Dynamic Type (1)

Re-assigning a reference variable to a newly-created object:

- `Substitution Principle`: the new object's class must be a **descendant class** of the reference variable's *static type*.
- e.g., `Student jim = new ResidentStudent (...)` changes the *dynamic type* of `jim` to `ResidentStudent`.
- e.g., `Student jim = new NonResidentStudent (...)` changes the *dynamic type* of `jim` to `NonResidentStudent`.
- e.g., `ResidentStudent jim = new Student (...)` is illegal because `Student` is **not** a **descendant class** of the *static type* of `jim` (i.e., `ResidentStudent`).

Reference Variable: Changing Dynamic Type (2)

Re-assigning a reference variable `v` to an existing object that is referenced by another variable `other` (i.e., `v = other`):

- **Substitution Principle**: the static type of `other` must be a **descendant class** of `v`'s *static type*.
- e.g., Say we declare

```
Student jim = new Student(...);
ResidentStudent rs = new ResidentStudent(...);
NonResidentStudent nrs = new NonResidentStudent(...);
```

- `rs = jim` ✗
- `nrs = jim` ✗
- `jim = rs` ✓
changes the *dynamic type* of `jim` to the dynamic type of `rs`
- `jim = nrs` ✓
changes the *dynamic type* of `jim` to the dynamic type of `nrs`

Polymorphism and Dynamic Binding (1)

- **Polymorphism**: An object variable may have “*multiple possible shapes*” (i.e., allowable *dynamic types*).
 - Consequently, there are *multiple possible versions* of each method that may be called.
 - e.g., A **Student** variable may have the *dynamic type* of **Student**, **ResidentStudent**, or **NonResidentStudent**,
 - This means that there are three possible versions of the `getTuition()` that may be called.
- **Dynamic binding**: When a method `m` is called on an object variable, the version of `m` corresponding to its “*current shape*” (i.e., one defined in the *dynamic type* of `m`) will be called.

```
Student jim = new ResidentStudent(...);  
jim.getTuition(); /* version in ResidentStudent */  
jim = new NonResidentStudent(...);  
jim.getTuition(); /* version in NonResidentStudent */
```

Polymorphism and Dynamic Binding (2.1)

```
class Student {...}
class ResidentStudent extends Student {...}
class NonResidentStudent extends Student {...}
```

```
class StudentTester1 {
    public static void main(String[] args) {
        Student jim = new Student("J. Davis");
        ResidentStudent rs = new ResidentStudent("J. Davis");
        jim = rs; /* legal */
        rs = jim; /* illegal */

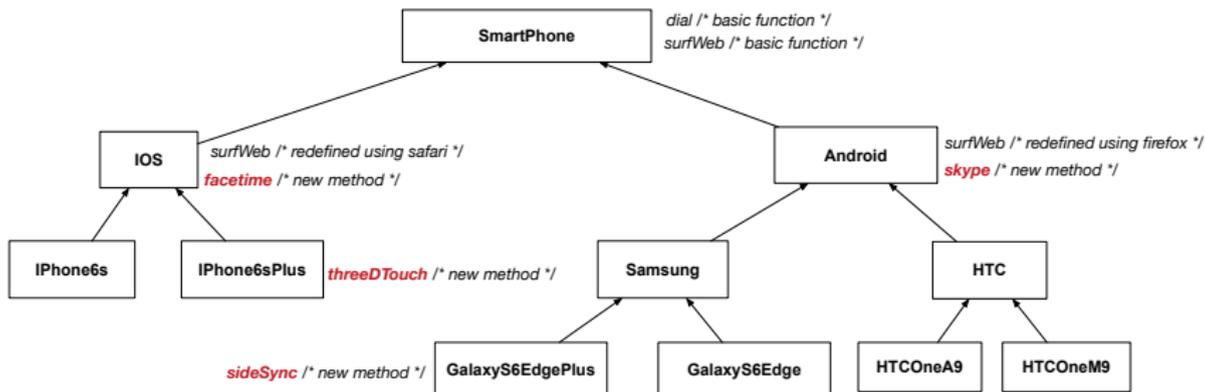
        NonResidentStudent nrs = new NonResidentStudent("J. Davis");
        jim = nrs; /* legal */
        nrs = jim; /* illegal */
    }
}
```

Polymorphism and Dynamic Binding (2.2)

```
class Student {...}
class ResidentStudent extends Student {...}
class NonResidentStudent extends Student {...}
```

```
class StudentTester2 {
    public static void main(String[] args) {
        Course eecs2030 = new Course("EECS2030", 500.0);
        Student jim = new Student("J. Davis");
        ResidentStudent rs = new ResidentStudent("J. Davis");
        rs.setPremiumRate(1.5);
        jim = rs;
        System.out.println(jim.getTuition()); /* 750.0 */
        NonResidentStudent nrs = new NonResidentStudent("J. Davis");
        nrs.setDiscountRate(0.5);
        jim = nrs;
        System.out.println(jim.getTuition()); /* 250.0 */
    }
}
```

Polymorphism and Dynamic Binding (3.1)



Polymorphism and Dynamic Binding (3.2)

```
class SmartPhoneTest1 {
    public static void main(String[] args) {
        SmartPhone myPhone;
        IOS ip = new iPhone6sPlus();
        Samsung ss = new GalaxyS6Edge();
        myPhone = ip; /* legal */
        myPhone = ss; /* legal */

        IOS presentForHeeyeon;
        presentForHeeyeon = ip; /* legal */
        presentForHeeyeon = ss; /* illegal */
    }
}
```

Polymorphism and Dynamic Binding (3.3)

```
class SmartPhoneTest2 {
    public static void main(String[] args) {
        SmartPhone myPhone;
        IOS ip = new iPhone6sPlus();
        myPhone = ip;
        myPhone.surfWeb (); /* version of surfWeb in iPhone6sPlus */

        Samsung ss = new GalaxyS6Edge ();
        myPhone = ss;
        myPhone.surfWeb (); /* version of surfWeb in GalaxyS6Edge */
    }
}
```

Reference Type Casting: Motivation (1)

```
1 Student jim = new ResidentStudent ("J. Davis");  
2 ResidentStudent rs = jim;  
3 rs.setPremiumRate(1.5);
```

- **L1** is *legal*: *ResidentStudent* is a **descendant class** of the *static type* of *jim* (i.e., *Student*).
- **L2** is *illegal*: *jim*'s **ST** (i.e., *Student*) is *not* a **descendant class** of *rs*'s **ST** (i.e., *ResidentStudent*).
- Java compiler is *unable to infer* that *jim*'s *dynamic type* in **L2** is *ResidentStudent*!
- Force the Java compiler to believe so via a cast in **L2**:

```
ResidentStudent rs = (ResidentStudent) jim;
```

- ⇒ Now it compiles ∴ *jim*'s **temporary ST** (*ResidentStudent*) is a **descendant** of *rs*' **ST** (*ResidentStudent*).
- *dynamic binding*: After the **cast**, **L3** will execute the correct version of `setPremiumRate`.

Reference Type Casting: Motivation (2)

```

1 SmartPhone aPhone = new iPhone6sPlus();
2 IOS forHeeyeon = aPhone;
3 forHeeyeon.facetime();

```

- **L1** is *legal*: iPhone6sPlus is a **descendant class** of the *static type* of aPhone (i.e., SmartPhone).
- **L2** is *illegal*: aPhone's **ST** (i.e., SmartPhone) is *not* a **descendant class** of forHeeyeon's **ST** (i.e., IOS).
- Java compiler is *unable to infer* that aPhone's *dynamic type* in **L2** is iPhone6sPlus!
- Force Java compiler to believe so via a **cast** in **L2**:

```
IOS forHeeyeon = (iPhone6sPlus) aPhone;
```

⇒ Now it compiles ∴ aPhone's **temporary ST** (iPhone6sPlus) is a **descendant** of forHeeyeon' **ST** (IOS).

- **dynamic binding**: After the **cast**, **L3** will execute the correct version of facetime.

Type Cast: Named or Anonymous

Named Cast: Use intermediate variable to store the cast result.

```
SmartPhone aPhone = new iPhone6sPlus();  
IOS forHeeyeon = (iPhone6sPlus) aPhone;  
forHeeyeon.facetime();
```

Anonymous Cast: Use the cast result directly.

```
SmartPhone aPhone = new iPhone6sPlus();  
((iPhone6sPlus) aPhone).facetime();
```

Common Mistake:

```
1 SmartPhone aPhone = new iPhone6sPlus();  
2 ((iPhone6sPlus) aPhone).facetime();
```

L2 \equiv `(iPhone6sPlus) (aPhone.facetime())`: Call, then cast.
 \Rightarrow This does **not** compile \because `facetime()` is **not** declared in the *static type* of `aPhone` (`SmartPhone`).

Notes on Type Cast (1)

- Given variable v of **static type** ST_v , it is **compilable** to cast v to C , as long as C is an **ancestor** or **descendant** of ST_v .
- Without cast, we can **only** call methods defined in ST_v on v .
- Casting v to C **temporarily** changes the **ST** of v from ST_v to C .
 ⇒ All methods that are defined in C can be called.

```

Android myPhone = new GalaxyS6EdgePlus();
/* can call methods declared in Android on myPhone
 * dial, surfweb, skype ✓ sideSync × */
SmartPhone sp = (SmartPhone) myPhone;
/* Compiles OK ∴ SmartPhone is an ancestor class of Android
 * expectations on sp narrowed to methods in SmartPhone
 * sp.dial, sp.surfweb ✓ sp.skype, sp.sideSync × */
GalaxyS6EdgePlus ga = (GalaxyS6EdgePlus) myPhone;
/* Compiles OK ∴ GalaxyS6EdgePlus is a descendant class of Android
 * expectations on ga widened to methods in GalaxyS6EdgePlus
 * ga.dial, ga.surfweb, ga.skype, ga.sideSync ✓ */

```

Reference Type Casting: Danger (1)

```
1 Student jim = new NonResidentStudent("J. Davis");  
2 ResidentStudent rs = (ResidentStudent) jim;  
3 rs.setPremiumRate(1.5);
```

- **L1** is *legal*: NonResidentStudent is a **descendant** of the static type of jim (Student).
- **L2** is *legal* (where the cast type is ResidentStudent):
 - cast type is **descendant** of jim's ST (Student).
 - cast type is **descendant** of rs's ST (ResidentStudent).
- **L3** is *legal* ∴ setPremiumRate is in rs' **ST** ResidentStudent.
- Java compiler is *unable to infer* that jim's **dynamic type** in **L2** is actually NonResidentStudent.
- Executing **L2** will result in a **ClassCastException**.
∴ Attribute premiumRate (expected from a **ResidentStudent**) is *undefined* on the **NonResidentStudent** object being cast.

Reference Type Casting: Danger (2)

```

1 SmartPhone aPhone = new GalaxyS6EdgePlus ();
2 iPhone6sPlus forHeeyeon = (iPhone6sPlus) aPhone;
3 forHeeyeon.threeDTouch ();
  
```

- **L1** is *legal*: GalaxyS6EdgePlus is a **descendant** of the static type of aPhone (SmartPhone).
- **L2** is *legal* (where the cast type is iPhone6sPlus):
 - cast type is **descendant** of aPhone's ST (SmartPhone).
 - cast type is **descendant** of forHeeyeon's ST (iPhone6sPlus).
- **L3** is *legal* ∴ threeDTouch is in forHeeyeon's **ST** iPhone6sPlus.
- Java compiler is *unable to infer* that aPhone's **dynamic type** in **L2** is actually NonResidentStudent.
- Executing **L2** will result in a **ClassCastException**.
 - ∴ Methods facetime, threeDTouch (expected from an **iPhone6sPlus**) is *undefined* on the **GalaxyS6EdgePlus** object being cast.

Notes on Type Cast (2.1)

Given a variable v of static type ST_v and dynamic type DT_v :

- `(C) v` is **compilable** if C is ST_v 's **ancestor** or **descendant**.
- Casting v to C 's **ancestor/descendant narrows/widens** expectations.
- However, being **compilable** does not guarantee **runtime-error-free!**

```

1 SmartPhone myPhone = new Samsung();
2 /* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
3 GalaxyS6EdgePlus ga = (GalaxyS6EdgePlus) myPhone;
4 /* Compiles OK ∴ GalaxyS6EdgePlus is a descendant class of SmartPhone
5  * can now call methods declared in GalaxyS6EdgePlus on ga
6  * ga.dial, ga.surfweb, ga.skype, ga.sideSync ✓ */

```

- Type cast in **L3** is **compilable**.
- Executing **L3** will cause **ClassCastException**.

L3: myPhone's **DT** Samsung cannot meet expectations of the temporary **ST** GalaxyS6EdgePlus (e.g., sideSync).

Notes on Type Cast (2.2)

Given a variable v of static type ST_v and dynamic type DT_v :

- `(C) v` is **compilable** if C is ST_v 's **ancestor** or **descendant**.
- Casting v to C 's **ancestor/descendant** **narrows/widens** expectations.
- However, being **compilable** does not guarantee **runtime-error-free!**

```

1 SmartPhone myPhone = new Samsung();
2 /* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
3 iPhone6sPlus ip = (iPhone6sPlus) myPhone;
4 /* Compiles OK ∴ iPhone6sPlus is a descendant class of SmartPhone
5  * can now call methods declared in iPhone6sPlus on ip
6  * ip.dial, ip.surfweb, ip.facetime, ip.threeDTouch ✓ */

```

- Type cast in **L3** is **compilable**.
- Executing **L3** will cause **ClassCastException**.

L3: myPhone's **DT** Samsung cannot meet expectations of the temporary **ST** iPhone6sPlus (e.g., threeDTouch).

Notes on Type Cast (2.3)

A cast $(C) \ v$ is *compilable* and *runtime-error-free* if C is located along the **ancestor path** of DT_v .

e.g., Given `SmartPhone myPhone = new Samsung();`

- Cast `myPhone` to a class along the path between `SmartPhone` and `Samsung`.
- Casting `myPhone` to a class with more expectations than `Samsung` (e.g., `GalaxyS6EdgePlus`) will cause `ClassCastException`.
- Casting `myPhone` to a class irrelevant to `Samsung` (e.g., `IPhone6sPlus`) will cause `ClassCastException`.

Compilable Cast vs. Exception-Free Cast

```
class A { }  
class B extends A { }  
class C extends B { }  
class D extends A { }
```

```
1 B b = new C();  
2 D d = (D) b;
```

- After **L1**:
 - **ST** of b is B
 - **DT** of b is C
- Does **L2** compile? [No]
∴ cast type D is neither an ancestor nor a descendant of b's **ST** B
- Would `D d = (D) ((A) b)` fix **L2**? [YES]
∴ cast type D is an ancestor of b's cast, temporary **ST** A
- `ClassCastException` when executing this fixed **L2**? [YES]
∴ cast type D is not an ancestor of b's **DT** C

Reference Type Casting: Runtime Check (1)

```
1 Student jim = new NonResidentStudent("J. Davis");
2 if (jim instanceof ResidentStudent) {
3     ResidentStudent rs = (ResidentStudent) jim;
4     rs.setPremiumRate(1.5);
5 }
```

- **L1** is *legal*: *NonResidentStudent* is a **descendant class** of the *static type* of jim (i.e., *Student*).
- **L2** checks if jim's *dynamic type* is *ResidentStudent*.
FALSE ∴ jim's *dynamic type* is *NonResidentStudent*!
- **L3** is *legal*: jim's cast type (i.e., *ResidentStudent*) is a **descendant class** of rs's *static type* (i.e., *ResidentStudent*).
- **L3** will not be executed at runtime, hence no *ClassCastException*, thanks to the check in **L2**!

Reference Type Casting: Runtime Check (2)

```
1 SmartPhone aPhone = new GalaxyS6EdgePlus ();
2 if (aPhone instanceof iPhone6sPlus) {
3     IOS forHeeyeon = (iPhone6sPlus) aPhone;
4     forHeeyeon.facetime();
5 }
```

- **L1** is *legal*: GalaxyS6EdgePlus is a **descendant class** of the static type of aPhone (i.e., SmartPhone).
- **L2** checks if aPhone's **dynamic type** is iPhone6sPlus.
FALSE ∴ aPhone's **dynamic type** is GalaxyS6EdgePlus!
- **L3** is *legal*: aPhone's cast type (i.e., iPhone6sPlus) is a **descendant class** of forHeeyeon's **static type** (i.e., IOS).
- **L3** will not be executed at runtime, hence no `ClassCastException`, thanks to the check in **L2**!

Notes on the instanceof Operator (1)

Given a reference variable v and a class C , you write

```
v instanceof C
```

to check if the **dynamic type** of v , at the moment of being checked, is a **descendant class** of C .

```
SmartPhone myPhone = new GalaxyS6Edge();
println(myPhone instanceof Android);
/* true :: GalaxyS6Edge is a descendant of Android */
println(myPhone instanceof Samsung);
/* true :: GalaxyS6Edge is a descendant of Samsung */
println(myPhone instanceof GalaxyS6Edge);
/* true :: GalaxyS6Edge is a descendant of GalaxyS6Edge */
println(myPhone instanceof IOS);
/* false :: GalaxyS6Edge is not a descendant of IOS */
println(myPhone instanceof iPhone6sPlus);
/* false :: GalaxyS6Edge is not a descendant of iPhone6sPlus */
```

Notes on the instanceof Operator (2)

Given a reference variable v and a class C ,

`v instanceof C` checks if the *dynamic type* of v , at the moment of being checked, is a descendant class of C .

```

1 SmartPhone myPhone = new Samsung();
2 /* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
3 if(myPhone instanceof Samsung) {
4     Samsung samsung = (Samsung) myPhone;
5 }
6 if(myPhone instanceof GalaxyS6EdgePlus) {
7     GalaxyS6EdgePlus galaxy = (GalaxyS6EdgePlus) myPhone;
8 }
9 if(myPhone instanceof HTC) {
10    HTC htc = (HTC) myPhone;
11 }
    
```

- **L3** evaluates to *true*. [safe to cast]
- **L6** and **L9** evaluate to *false*. [unsafe to cast]

This prevents **L7** and **L10**, causing `ClassCastException` if executed, from being executed.

Static Type and Polymorphism (1.1)

```
class SmartPhone {  
    void dial() { ... }  
}  
class IOS extends SmartPhone {  
    void facetime() { ... }  
}  
class iPhone6sPlus extends IOS {  
    void threeDTouch() { ... }  
}
```

```
1 SmartPhone sp = new iPhone6sPlus();    ✓  
2 sp.dial();                               ✓  
3 sp.facetime();                            ×  
4 sp.threeDTouch();                         ×
```

Static type of *sp* is SmartPhone

⇒ can only call methods defined in SmartPhone on *sp*

Static Type and Polymorphism (1.2)

```
class SmartPhone {  
    void dial() { ... }  
}  
class IOS extends SmartPhone {  
    void facetime() { ... }  
}  
class iPhone6sPlus extends IOS {  
    void threeDTouch() { ... }  
}
```

```
1 IOS ip = new iPhone6sPlus();    ✓  
2 ip.dial();                       ✓  
3 ip.facetime();                   ✓  
4 ip.threeDTouch();                ✗
```

Static type of *ip* is IOS

⇒ can only call methods defined in IOS on *ip*

Static Type and Polymorphism (1.3)

```
class SmartPhone {  
    void dial() { ... }  
}  
class IOS extends SmartPhone {  
    void facetime() { ... }  
}  
class iPhone6sPlus extends IOS {  
    void threeDTouch() { ... }  
}
```

```
1 iPhone6sPlus ip6sp = new iPhone6sPlus();    ✓  
2 ip6sp.dial();                                ✓  
3 ip6sp.facetime();                            ✓  
4 ip6sp.threeDTouch();                        ✓
```

Static type of *ip6sp* is iPhone6sPlus

⇒ can call all methods defined in iPhone6sPlus on *ip6sp*

Static Type and Polymorphism (1.4)

```
class SmartPhone {  
    void dial() { ... }  
}  
class IOS extends SmartPhone {  
    void facetime() { ... }  
}  
class iPhone6sPlus extends IOS {  
    void threeDTouch() { ... }  
}
```

```
1 SmartPhone sp = new iPhone6sPlus(); ✓  
2 ( iPhone6sPlus ) sp.dial(); ✓  
3 ( iPhone6sPlus ) sp.facetime(); ✓  
4 ( iPhone6sPlus ) sp.threeDTouch(); ✓
```

L4 is equivalent to the following two lines:

```
iPhone6sPlus ip6sp = ( iPhone6sPlus ) sp;  
ip6sp.threeDTouch();
```

Static Type and Polymorphism (2)

Given a reference variable declaration

```
C v;
```

- **Static type** of reference variable v is class C
- A method call `v.m` is valid if m is a method **defined** in class **C** .
- Despite the **dynamic type** of v , you are only allowed to call methods that are defined in the **static type** C on v .
- If you are certain that v 's **dynamic type** can be expected **more** than its **static type**, then you may use an `instanceof` check and a cast.

```
Course eecs2030 = new Course("EECS2030", 500.0);
Student s = new ResidentStudent("Jim");
s.register(eecs2030);
if(s instanceof ResidentStudent) {
    ((ResidentStudent) s).setPremiumRate(1.75);
    System.out.println(((ResidentStudent) s).getTuition());
}
```

Polymorphism: Method Call Arguments (1)

```
1 class StudentManagementSystem {
2     Student [] ss; /* ss[i] has static type Student */ int c;
3     void addRS(ResidentStudent rs) { ss[c] = rs; c++; }
4     void addNRS(NonResidentStudent nrs) { ss[c] = nrs; c++; }
5     void addStudent(Student s) { ss[c] = s; c++; } }
```

- **L3:** `ss[c] = rs` is valid. ∴ RHS's ST `ResidentStudent` is a *descendant class* of LHS's ST `Student`.
- Say we have a `StudentManagementSystem` object `sms`:
 - Method call `sms.addRS(o)` attempts the following assignment, which replaces parameter `rs` by a copy of argument `o`:

```
rs = o;
```
 - Whether this argument passing is valid depends on `o`'s *static type*.
- In the signature of a method `m`, if the type of a parameter is class `C`, then we may call method `m` by passing objects whose *static types* are `C`'s *descendants*.

Polymorphism: Method Call Arguments (2.1)

In the StudentManagementSystemTester:

```
Student s1 = new Student();  
Student s2 = new ResidentStudent();  
Student s3 = new NonResidentStudent();  
ResidentStudent rs = new ResidentStudent();  
NonResidentStudent nrs = new NonResidentStudent();  
StudentManagementSystem sms = new StudentManagementSystem();  
sms.addRS(s1);    ×  
sms.addRS(s2);    ×  
sms.addRS(s3);    ×  
sms.addRS(rs);    ✓  
sms.addRS(nrs);   ×  
sms.addStudent(s1); ✓  
sms.addStudent(s2); ✓  
sms.addStudent(s3); ✓  
sms.addStudent(rs); ✓  
sms.addStudent(nrs); ✓
```

Polymorphism: Method Call Arguments (2.2)

In the StudentManagementSystemTester:

```
1 Student s = new Student("Stella");
2 /* s' ST: Student; s' DT: Student */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(s); ✗
```

- o **L4 compiles** with a cast: `sms.addRS((ResidentStudent) s)`
 - **Valid** cast \therefore (ResidentStudent) is a descendant of s' **ST**.
 - **Valid** call \therefore s' temporary **ST** (ResidentStudent) is now a descendant class of addRS's parameter rs' **ST** (ResidentStudent).
- o But, there will be a **ClassCastException** at runtime!
 \therefore s' **DT** (Student) is **not** a descendant of ResidentStudent.
- o We should have written:

```
if(s instanceof ResidentStudent) {
    sms.addRS((ResidentStudent) s);
}
```

The **instanceof** expression will evaluate to **false**, meaning it is **unsafe** to cast, thus preventing ClassCastException.

Polymorphism: Method Call Arguments (2.3)

In the StudentManagementSystemTester:

```
1 Student s = new NonResidentStudent("Nancy");
2 /* s' ST: Student; s' DT: NonResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(s); ✗
```

- o **L4 compiles** with a cast: `sms.addRS((ResidentStudent) s)`
 - **Valid** cast \therefore (ResidentStudent) is a descendant of s' **ST**.
 - **Valid** call \therefore s' temporary **ST** (ResidentStudent) is now a descendant class of addRS's parameter rs' **ST** (ResidentStudent).
- o But, there will be a **ClassCastException** at runtime!
 \therefore s' **DT** (NonResidentStudent) **not descendant** of ResidentStudent.
- o We should have written:

```
if(s instanceof ResidentStudent) {
    sms.addRS((ResidentStudent) s);
}
```

The **instanceof** expression will evaluate to **false**, meaning it is **unsafe** to cast, thus preventing ClassCastException.

Polymorphism: Method Call Arguments (2.4)

In the StudentManagementSystemTester:

```
1 Student s = new ResidentStudent("Rachael");
2 /* s' ST: Student; s' DT: ResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(s); ×
```

- o **L4 compiles** with a cast: `sms.addRS((ResidentStudent) s)`
 - **Valid** cast \therefore (ResidentStudent) is a descendant of s' **ST**.
 - **Valid** call \therefore s' temporary **ST** (ResidentStudent) is now a descendant class of addRS's parameter rs' **ST** (ResidentStudent).
- o And, there will be **no ClassCastException** at runtime!
 \therefore s' **DT** (ResidentStudent) is descendant of ResidentStudent.
- o We should have written:

```
if(s instanceof ResidentStudent) {
    sms.addRS((ResidentStudent) s);
}
```

The **instanceof** expression will evaluate to **true**, meaning it is **safe** to cast.

Polymorphism: Method Call Arguments (2.5)

In the StudentManagementSystemTester:

```
1 NonResidentStudent nrs = new NonResidentStudent();
2 /* ST: NonResidentStudent; DT: NonResidentStudent */
3 StudentManagementSystem sms = new StudentManagementSystem();
4 sms.addRS(nrs); ×
```

Will L4 with a cast compile?

```
sms.addRS ( (ResidentStudent) nrs )
```

NO ∴ (ResidentStudent) is **not** a descendant of nrs's **ST** (NonResidentStudent).

Polymorphism: Return Values (1)

```
1  class StudentManagementSystem {
2      Student[] ss; int c;
3      void addStudent(Student s) { ss[c] = s; c++; }
4      Student getStudent(int i) {
5          Student s = null;
6          if(i < 0 || i >= c) {
7              throw new IllegalArgumentException("Invalid index.");
8          }
9          else {
10             s = ss[i];
11         }
12         return s;
13     } }
```

L4: Student is **static type** of getStudent's return value.

L10: ss[i]'s ST (Student) is **descendant** of s' ST (Student).

Question: What can be the **dynamic type** of s after L10?

Answer: All descendant classes of Student.

Polymorphism: Return Values (2)

```

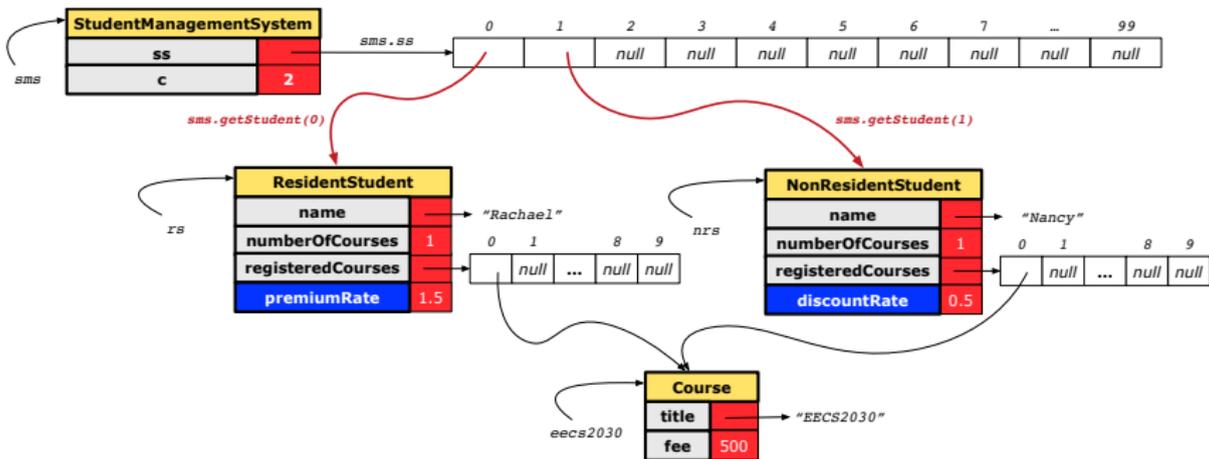
1 Course eecs2030 = new Course("EECS2030", 500);
2 ResidentStudent rs = new ResidentStudent("Rachael");
3 rs.setPremiumRate(1.5); rs.register(eecs2030);
4 NonResidentStudent nrs = new NonResidentStudent("Nancy");
5 nrs.setDiscountRate(0.5); nrs.register(eecs2030);
6 StudentManagementSystem sms = new StudentManagementSystem();
7 sms.addStudent(rs); sms.addStudent(nrs);
8 Student s = sms.getStudent(0); /* dynamic type of s? */
           static return type: Student
9 print(s instanceof Student && s instanceof ResidentStudent); /*true*/
10 print(s instanceof NonResidentStudent); /* false */
11 print(s.getTuition()); /*Version in ResidentStudent called:750*/
12 ResidentStudent rs2 = sms.getStudent(0); x
13 s = sms.getStudent(1); /* dynamic type of s? */
           static return type: Student
14 print(s instanceof Student && s instanceof NonResidentStudent); /*true*/
15 print(s instanceof ResidentStudent); /* false */
16 print(s.getTuition()); /*Version in NonResidentStudent called:250*/
17 NonResidentStudent nrs2 = sms.getStudent(1); x

```

Polymorphism: Return Values (3)

At runtime, attribute `sms.ss` is a **polymorphic** array:

- **Static type** of each item is as declared: **Student**
- **Dynamic type** of each item is a descendant of **Student**: **ResidentStudent**, **NonResidentStudent**



Why Inheritance: A Collection of Various Kinds of Students

How do you define a class `StudentManagementSystem` that contains a list of *resident* and *non-resident* students?

```
class StudentManagementSystem {
    Student[] students;
    int numOfStudents;

    void addStudent(Student s) {
        students[numOfStudents] = s;
        numOfStudents++;
    }

    void registerAll (Course c) {
        for(int i = 0; i < numberOfStudents; i++) {
            students[i].register(c)
        }
    }
}
```

Polymorphism and Dynamic Binding: A Collection of Various Kinds of Students

```
class StudentManagementSystemTester {
    static void main(String[] args) {
        ResidentStudent jim = new ResidentStudent("J. Davis");
        NonResidentStudent jeremy =
            new NonResidentStudent("J. Davis");
        StudentManagementSystem sms =
            new StudentManagementSystem();
        sms.addStudent(jim); /* polymorphism */
        sms.addStudent(jeremy); /* polymorphism */
        Course eecs2030 = new Course("EECS2030", 500.0);
        sms.registerAll(eecs2030);
        for(int i = 0; i < sms.numberOfStudents; i++) {
            /* Dynamic Binding:
             * Right version of getTuition will be called */
            System.out.println(sms.students[i].getTuition());
        }
    }
}
```

Static Type vs. Dynamic Type: When to consider which?

- *Whether or not Java code compiles* depends only on the **static types** of relevant variables.
 - ∴ Inferring the **dynamic type** statically is an **undecidable** problem that is inherently impossible to solve.
- *The behaviour of Java code being executed at runtime* (e.g., which version of method is called due to dynamic binding, whether or not a `ClassCastException` will occur, etc.) depends on the **dynamic types** of relevant variables.
 - ⇒ Best practice is to visualize how objects are created (by drawing boxes) and variables are re-assigned (by drawing arrows).

Summary: Type Checking Rules

CODE	CONDITION TO BE TYPE CORRECT
$x = y$	Is y 's ST a descendant of x 's ST ?
$x.m(y)$	Is method m defined in x 's ST ? Is y 's ST a descendant of m 's parameter's ST ?
$z = x.m(y)$	Is method m defined in x 's ST ? Is y 's ST a descendant of m 's parameter's ST ? Is ST of m 's return value a descendant of z 's ST ?
$(C) y$	Is C an ancestor or a descendant of y 's ST ?
$x = (C) y$	Is C an ancestor or a descendant of y 's ST ? Is C a descendant of x 's ST ?
$x.m((C) y)$	Is C an ancestor or a descendant of y 's ST ? Is method m defined in x 's ST ? Is C a descendant of m 's parameter's ST ?

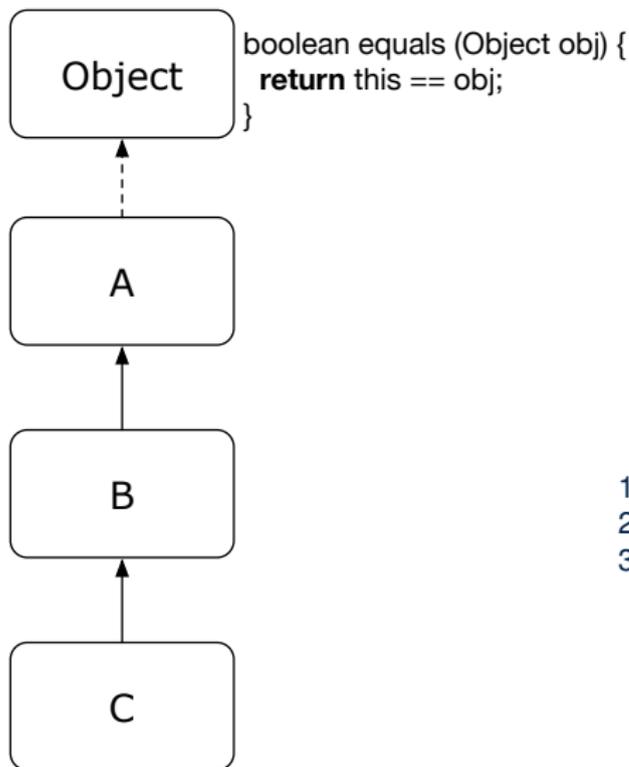
Even if $(C) y$ compiles OK, there will be a runtime `ClassCastException` if C is not an **ancestor** of y 's **DT**!

Overriding and Dynamic Binding (1)

Object is the common parent/super class of every class.

- Every class inherits the **default version** of `equals`
- Say a reference variable `v` has **dynamic type** `D`:
 - **Case 1** `D` **overrides** `equals`
⇒ `v.equals(...)` invokes the **overridden version** in `D`
 - **Case 2** `D` does **not override** `equals`
 - Case 2.1** At least one ancestor classes of `D` **override** `equals`
⇒ `v.equals(...)` invokes the **overridden version** in the **closest ancestor class**
 - Case 2.2** No ancestor classes of `D` **override** `equals`
⇒ `v.equals(...)` invokes **default version** inherited from `Object`.
- Same principle applies to the `toString` method, and all overridden methods in general.

Overriding and Dynamic Binding (2.1)

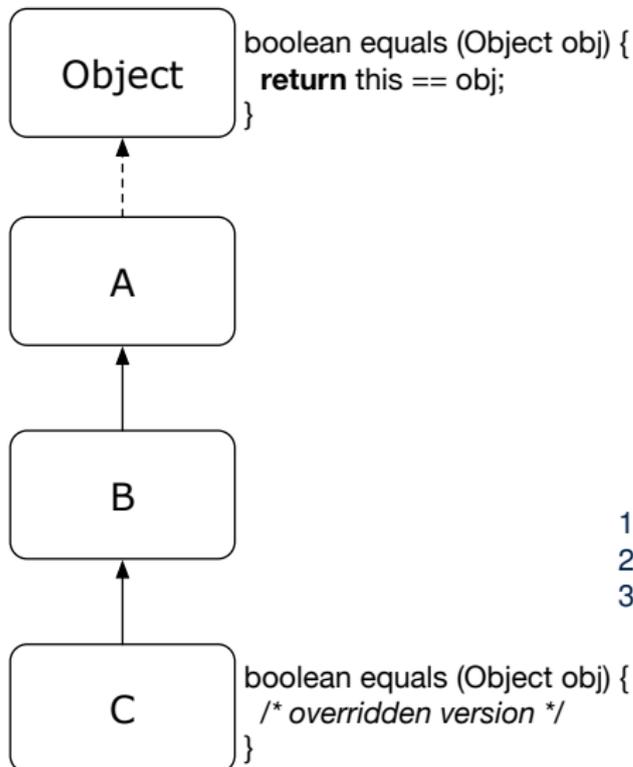


```
class A {  
    /*equals not overridden*/  
}  
class B extends A {  
    /*equals not overridden*/  
}  
class C extends B {  
    /*equals not overridden*/  
}
```

```
1 Object c1 = new C();  
2 Object c2 = new C();  
3 println(c1.equals(c2));
```

L3 calls which version of
equals? [Object]

Overriding and Dynamic Binding (2.2)

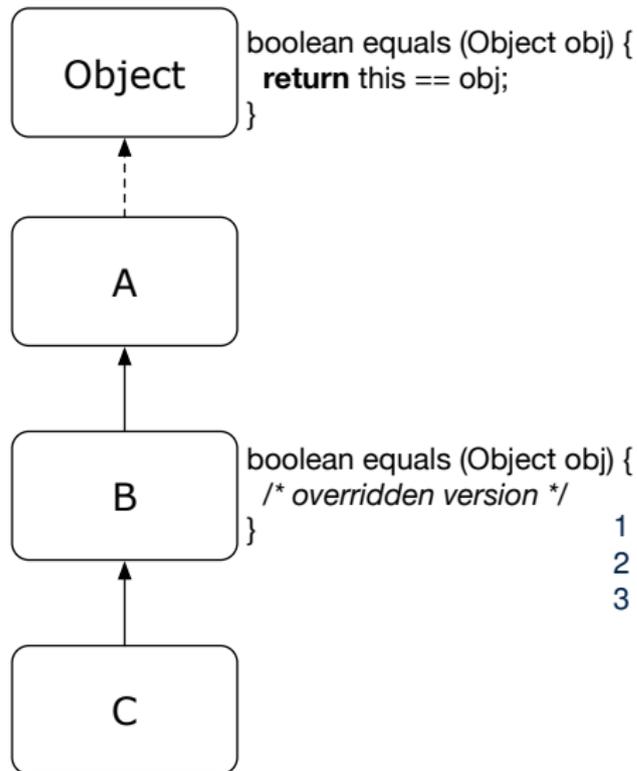


```
class A {  
    /*equals not overridden*/  
}  
class B extends A {  
    /*equals not overridden*/  
}  
class C extends B {  
    boolean equals (Object obj) {  
        /* overridden version */  
    }  
}
```

```
1 Object c1 = new C();  
2 Object c2 = new C();  
3 println(c1.equals(c2));
```

L3 calls which version of equals? [C]

Overriding and Dynamic Binding (2.3)



```
class A {  
    /*equals not overridden*/  
}  
class B extends A {  
    boolean equals (Object obj) {  
        /* overridden version */  
    }  
}  
class C extends B {  
    /*equals not overridden*/  
}
```

```
1 Object c1 = new C();  
2 Object c2 = new C();  
3 println(c1.equals(c2));
```

L3 calls which version of equals? [B]

Index (1)

Why Inheritance: A Motivating Example

No Inheritance: ResidentStudent Class

No Inheritance: NonResidentClass

No Inheritance: Testing Student Classes

No Inheritance:

Issues with the Student Classes

No Inheritance: Maintainability of Code (1)

No Inheritance: Maintainability of Code (2)

No Inheritance:

A Collection of Various Kinds of Students

Inheritance Architecture

Inheritance: The Student Parent/Super Class

Inheritance:

The ResidentStudent Child/Sub Class

Index (2)

Inheritance:

The NonResidentStudent Child/Sub Class

Inheritance Architecture Revisited

Visualizing Parent/Child Objects (1)

Visualizing Parent/Child Objects (2)

Using Inheritance for Code Reuse

Inheritance Architecture Revisited

Testing the Two Student Sub-Classes

Multi-Level Inheritance Architecture

Root of the Java Class Hierarchy

Behaviour of the Inherited equals Method (1)

Behaviour of the Inherited equals Method (2)

Behaviour of the Inherited equals Method (3)

Behaviour of the Inherited equals Method (4)

Index (3)

- Behaviour of the Inherited `equals` Method (5)
- Behaviour of Inherited `toString` Method (1)
- Behaviour of Inherited `toString` Method (2)
- Behaviour of Inherited `toString` Method (3)
- Use of the `protected` Modifier
- Visibility of Attr./Meth.: Across All Methods
- Within the Resident Package and Sub-Classes (`protected`)
- Visibility of Attr./Meth.
- Inheritance Architecture Revisited
- Multi-Level Inheritance Hierarchy:
 - Smart Phones
- Polymorphism: Intuition (1)
- Polymorphism: Intuition (2)
- Polymorphism: Intuition (3)
- Dynamic Binding: Intuition (1)

Index (4)

- Dynamic Binding: Intuition (2)
- Inheritance Forms a Type Hierarchy
- Inheritance Accumulates Code for Reuse
- Reference Variable: Static Type
- Substitutions via Assignments
- Rules of Substitution
- Reference Variable: Dynamic Type
- Visualizing Static Type vs. Dynamic Type
- Reference Variable:
 - Changing Dynamic Type (1)
- Reference Variable:
 - Changing Dynamic Type (2)
- Polymorphism and Dynamic Binding (1)
- Polymorphism and Dynamic Binding (2.1)

Index (5)

Polymorphism and Dynamic Binding (2.2)

Polymorphism and Dynamic Binding (3.1)

Polymorphism and Dynamic Binding (3.2)

Polymorphism and Dynamic Binding (3.3)

Reference Type Casting: Motivation (1)

Reference Type Casting: Motivation (2)

Type Cast: Named or Anonymous

Notes on Type Cast (1)

Reference Type Casting: Danger (1)

Reference Type Casting: Danger (2)

Notes on Type Cast (2.1)

Notes on Type Cast (2.2)

Notes on Type Cast (2.3)

Compilable Cast vs. Exception-Free Cast

Index (6)

Reference Type Casting: Runtime Check (1)

Reference Type Casting: Runtime Check (2)

Notes on the `instanceof` Operator (1)

Notes on the `instanceof` Operator (2)

Static Type and Polymorphism (1.1)

Static Type and Polymorphism (1.2)

Static Type and Polymorphism (1.3)

Static Type and Polymorphism (1.4)

Static Type and Polymorphism (2)

Polymorphism: Method Call Arguments (1)

Polymorphism: Method Call Arguments (2.1)

Polymorphism: Method Call Arguments (2.2)

Polymorphism: Method Call Arguments (2.3)

Polymorphism: Method Call Arguments (2.4)

Index (7)

Polymorphism: Method Call Arguments (2.5)

Polymorphism: Return Values (1)

Polymorphism: Return Values (2)

Polymorphism: Return Values (3)

Why Inheritance:

A Collection of Various Kinds of Students

Polymorphism and Dynamic Binding:

A Collection of Various Kinds of Students

Static Type vs. Dynamic Type:

When to consider which?

Summary: Type Checking Rules

Overriding and Dynamic Binding (1)

Overriding and Dynamic Binding (2.1)

Overriding and Dynamic Binding (2.2)

Overriding and Dynamic Binding (2.3)

Abstract Classes and Interfaces



EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Abstract Class (1)

Problem: A polygon may be either a triangle or a rectangle. Given a polygon, we may either

- Grow its shape by incrementing the size of each of its sides;
 - Compute and return its perimeter; or
 - Compute and return its area.
- For a rectangle with *length* and *width*, its area is $length \times width$.
 - For a triangle with sides *a*, *b*, and *c*, its area, according to Heron's formula, is

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where

$$s = \frac{a + b + c}{2}$$

- How would you solve this problem in Java, while minimizing code duplicates?

Abstract Class (2)

```
public abstract class Polygon {
    double[] sides;
    Polygon(double[] sides) { this.sides = sides; }
    void grow() {
        for(int i = 0; i < sides.length; i++) {
            sides[i]++;
        }
    }
    double getPerimeter() {
        double perimeter = 0;
        for(int i = 0; i < sides.length; i++) { perimeter += sides[i]; }
        return perimeter;
    }
    abstract double getArea();
}
```

- Method `getArea` not implemented and shown **signature** only.
- \therefore Polygon cannot be used as a **dynamic type**
- Writing `new Polygon(...)` is forbidden!

Abstract Class (3)

```
public class Rectangle extends Polygon {  
    Rectangle(double length, double width) {  
        super(new double[4]);  
        sides[0] = length; sides[1] = width;  
        sides[2] = length; sides[3] = width;  
    }  
    double getArea() { return sides[0] * sides[1]; }  
}
```

- Method `getPerimeter` is inherited from the super-class `Polygon`.
- Method `getArea` is implemented in the sub-class `Rectangle`.
- \therefore `Rectangle` can be used as a **dynamic type**
- Writing `Polygon p = new Rectangle(3, 4)` allowed!

Abstract Class (4)

```
public class Triangle extends Polygon {
    Triangle(double side1, double side2, double side3) {
        super(new double[3]);
        sides[0] = side1; sides[1] = side2; sides[2] = side3;
    }
    double getArea() {
        /* Heron's formula */
        double s = getPerimeter() * 0.5;
        double area = Math.sqrt(
            s * (s - sides[0]) * (s - sides[1]) * (s - sides[2]));
        return area;
    }
}
```

- Method `getPerimeter` is inherited from `Polygon`.
- Method `getArea` is implemented in the sub-class `Triangle`.
- ∴ `Triangle` can be used as a **dynamic type**
- Writing `Polygon p = new Triangle(3, 4, 5)` allowed!

Abstract Class (5)

```
1 public class PolygonCollector {
2     Polygon[] polygons;
3     int numberOfPolygons;
4     PolygonCollector() { polygons = new Polygon[10]; }
5     void addPolygon(Polygon p) {
6         polygons[numberOfPolygons] = p; numberOfPolygons ++;
7     }
8     void growAll() {
9         for(int i = 0; i < numberOfPolygons; i ++ ) {
10            polygons[i].grow();
11        }
12    }
13 }
```

- **Polymorphism:** Line 5 may accept as argument any object whose *static type* is Polygon or any of its sub-classes.
- **Dynamic Binding:** Line 10 calls the version of `grow` inherited to the *dynamic type* of `polygons[i]`.

Abstract Class (6)

```
1 public class PolygonConstructor {
2     Polygon getPolygon(double[] sides) {
3         Polygon p = null;
4         if(sides.length == 3) {
5             p = new Triangle(sides[0], sides[1], sides[2]);
6         }
7         else if(sides.length == 4) {
8             p = new Rectangle(sides[0], sides[1]);
9         }
10        return p;
11    }
12    void grow(Polygon p) { p.grow(); }
13 }
```

- **Polymorphism:**

- **Line 2** may accept as return value any object whose **static type** is Polygon or any of its sub-classes.
- **Line 5** returns an object whose **dynamic type** is Triangle; **Line 8** returns an object whose **dynamic type** is Rectangle.

Abstract Class (7.1)

```
1 public class PolygonTester {
2     public static void main(String[] args) {
3         Polygon p;
4         p = new Rectangle(3, 4); /* polymorphism */
5         System.out.println(p.getPerimeter()); /* 14.0 */
6         System.out.println(p.getArea()); /* 12.0 */
7         p = new Triangle(3, 4, 5); /* polymorphism */
8         System.out.println(p.getPerimeter()); /* 12.0 */
9         System.out.println(p.getArea()); /* 6.0 */
10
11        PolygonCollector col = new PolygonCollector();
12        col.addPolygon(new Rectangle(3, 4)); /* polymorphism */
13        col.addPolygon(new Triangle(3, 4, 5)); /* polymorphism */
14        System.out.println(col.polygons[0].getPerimeter()); /* 14.0 */
15        System.out.println(col.polygons[1].getPerimeter()); /* 12.0 */
16        col.growAll();
17        System.out.println(col.polygons[0].getPerimeter()); /* 18.0 */
18        System.out.println(col.polygons[1].getPerimeter()); /* 15.0 */
```

Abstract Class (7.2)

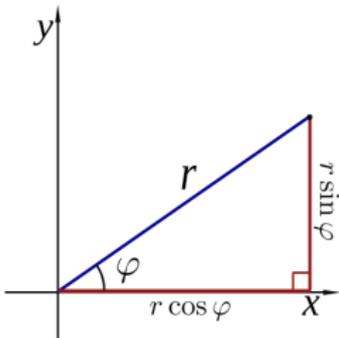
```
1 PolygonConstructor con = new PolygonConstructor();
2 double[] recSides = {3, 4, 3, 4}; p = con.getPolygon(recSides);
3 System.out.println(p instanceof Polygon); ✓
4 System.out.println(p instanceof Rectangle); ✓
5 System.out.println(p instanceof Triangle); ✗
6 System.out.println(p.getPerimeter()); /* 14.0 */
7 System.out.println(p.getArea()); /* 12.0 */
8 con.grow(p);
9 System.out.println(p.getPerimeter()); /* 18.0 */
10 System.out.println(p.getArea()); /* 20.0 */
11 double[] triSides = {3, 4, 5}; p = con.getPolygon(triSides);
12 System.out.println(p instanceof Polygon); ✓
13 System.out.println(p instanceof Rectangle); ✗
14 System.out.println(p instanceof Triangle); ✓
15 System.out.println(p.getPerimeter()); /* 12.0 */
16 System.out.println(p.getArea()); /* 6.0 */
17 con.grow(p);
18 System.out.println(p.getPerimeter()); /* 15.0 */
19 System.out.println(p.getArea()); /* 9.921 */
20 } }
```

Abstract Class (8)

- An **abstract class**:
 - Typically has **at least one** method with no implementation body
 - May define common implementations inherited to **sub-classes**.
- Recommended to use an **abstract class** as the **static type** of:
 - A **variable**
e.g., `Polygon p`
 - A **method parameter**
e.g., `void grow(Polygon p)`
 - A **method return value**
e.g., `Polygon getPolygon(double[] sides)`
- It is forbidden to use an **abstract class** as a **dynamic type**
e.g., `Polygon p = new Polygon(...)` is not allowed!
- Instead, create objects whose **dynamic types** are descendant classes of the **abstract class** ⇒ Exploit **dynamic binding** !
e.g., `Polygon p = con.getPolygon(recSides)`
This is as if we did `Polygon p = new Rectangle(...)`

Interface (1.1)

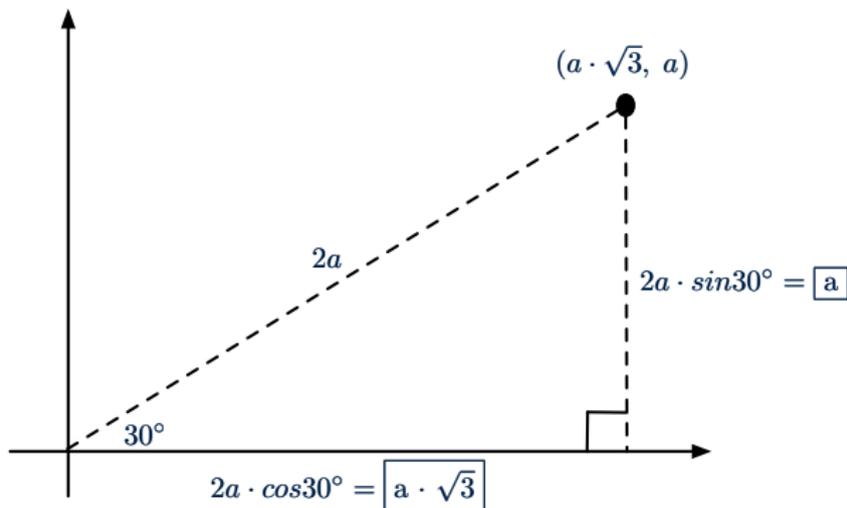
- We may implement `Point` using two representation systems:



- The *Cartesian system* stores the *absolute* positions of x and y .
 - The *Polar system* stores the *relative* position: the angle (in radian) ϕ and distance r from the origin $(0,0)$.
- As far as users of a `Point` object p is concerned, being able to call `p.getX()` and `getY()` is what matters.
- How `p.getX()` and `p.getY()` are internally computed, depending on the *dynamic type* of p , do not matter to users.

Interface (1.2)

Recall: $\sin 30^\circ = \frac{1}{2}$ and $\cos 30^\circ = \frac{1}{2} \cdot \sqrt{3}$



We consider the same point represented differently as:

- $r = 2a, \psi = 30^\circ$ [polar system]
- $x = 2a \cdot \cos 30^\circ = a \cdot \sqrt{3}, y = 2a \cdot \sin 30^\circ = a$ [cartesian system]

Interface (2)

```
interface Point {  
    double getX();  
    double getY();  
}
```

- An interface `Point` defines how users may access a point: either get its `x` coordinate or its `y` coordinate.
- Methods `getX` and `getY` similar to `getArea` in `Polygon`, have no implementations, but *signatures* only.
- \therefore `Point` cannot be used as a *dynamic type*
- Writing `new Point(...)` is forbidden!

Interface (3)

```
public class CartesianPoint implements Point {
    double x;
    double y;
    CartesianPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double getX() { return x; }
    public double getY() { return y; }
}
```

- CartesianPoint is a possible implementation of Point.
- Attributes `x` and `y` declared according to the *Cartesian system*
- All method from the interface `Point` are implemented in the sub-class `CartesianPoint`.
- ∴ CartesianPoint can be used as a **dynamic type**
- `Point p = new CartesianPoint(3, 4)` allowed!

Interface (4)

```
public class PolarPoint implements Point {
    double phi;
    double r;
    public PolarPoint(double r, double phi) {
        this.r = r;
        this.phi = phi;
    }
    public double getX() { return Math.cos(phi) * r; }
    public double getY() { return Math.sin(phi) * r; }
}
```

- PolarPoint is a possible implementation of Point.
- Attributes phi and r declared according to the *Polar system*
- All method from the interface Point are implemented in the sub-class PolarPoint.
- ∴ PolarPoint can be used as a *dynamic type*
- Point p = new PolarPoint(3, $\frac{\pi}{6}$) allowed! [360° = 2π]

Interface (5)

```
1 public class PointTester {
2     public static void main(String[] args) {
3         double A = 5;
4         double X = A * Math.sqrt(3);
5         double Y = A;
6         Point p;
7         p = new CartesianPoint(X, Y); /* polymorphism */
8         print("(" + p.getX() + ", " + p.getY() + ")"); /* dyn. bin. */
9         p = new PolarPoint(2 * A, Math.toRadians(30)); /* polymorphism */
10        print("(" + p.getX() + ", " + p.getY() + ")"); /* dyn. bin. */
11    }
12 }
```

- Lines 7 and 9 illustrate *polymorphism*, how?
- Lines 8 and 10 illustrate *dynamic binding*, how?

Interface (6)

- An **interface**:
 - Has **all** its methods with no implementation bodies.
 - Leaves complete freedom to its **implementors**.
- Recommended to use an **interface** as the **static type** of:
 - A **variable**
e.g., `Point p`
 - A **method parameter**
e.g., `void moveUp(Point p)`
 - A **method return value**
e.g., `Point getPoint(double v1, double v2, boolean isCartesian)`
- It is forbidden to use an **interface** as a **dynamic type**
e.g., `Point p = new Point(...)` is not allowed!
- Instead, create objects whose **dynamic types** are descendant classes of the **interface** ⇒ Exploit **dynamic binding** !

Abstract Classes vs. Interfaces: When to Use Which?

- Use **interfaces** when:
 - There is a *common set of functionalities* that can be implemented via *a variety of strategies*.
e.g., Interface `Point` declares signatures of `getX()` and `getY()`.
 - Each descendant class represents a different implementation strategy for the same set of functionalities.
 - `CartesianPoint` and `PolarPoint` represent different strategies for supporting `getX()` and `getY()`.
- Use **abstract classes** when:
 - *Some (not all) implementations can be shared* by descendants, and *some (not all) implementations cannot be shared*.
e.g., Abstract class `Polygon`:
 - Defines implementation of `getPerimeter`, to be shared by `Rectangle` and `Triangle`.
 - Declares signature of `getArea`, to be implemented by `Rectangle` and `Triangle`.

Index (1)

Abstract Class (1)

Abstract Class (2)

Abstract Class (3)

Abstract Class (4)

Abstract Class (5)

Abstract Class (6)

Abstract Class (7.1)

Abstract Class (7.2)

Abstract Class (8)

Interface (1.1)

Interface (1.2)

Interface (2)

Interface (3)

Interface (4)

Index (2)

Interface (5)

Interface (6)

Abstract Classes vs. Interfaces: When to Use Which?

Generics in Java



EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Motivating Example: A Book of Objects

```
1 class Book {
2     String[] names;
3     Object[] records;
4     /* add a name-record pair to the book */
5     void add (String name, Object record) { ... }
6     /* return the record associated with a given name */
7     Object get (String name) { ... } }
```

Question: Which line has a type error?

```
1 Date birthday; String phoneNumber;
2 Book b; boolean isWednesday;
3 b = new Book();
4 phoneNumber = "416-67-1010";
5 b.add ("Suyeon", phoneNumber);
6 birthday = new Date(1975, 4, 10);
7 b.add ("Yuna", birthday);
8 isWednesday = b.get("Yuna").getDay() == 4;
```

Motivating Example: Observations (1)

- In the `Book` class:
 - By declaring the attribute

```
Object[] records
```

We meant that each book instance may store any object whose *static type* is a **descendant class** of `Object`.

- Accordingly, from the return type of the `get` method, we only know that the returned record is an `Object`, but not certain about its *dynamic type* (e.g., `Date`, `String`, *etc.*).
∴ a record retrieved from the book, e.g., `b.get("Yuna")`, may only be called upon methods in its *static type* (i.e., `Object`).
- In the tester code of the `Book` class:
 - In **Line 1**, the *static types* of variables `birthday` (i.e., `Date`) and `phoneNumber` (i.e., `String`) are **descendant classes** of `Object`.
 - So, **Line 5** and **Line 7** compile.

Motivating Example: Observations (2)

Due to **polymorphism**, the *dynamic types* of stored objects (e.g., `phoneNumber` and `birthday`) need not be the same.

- Methods supported in the *dynamic types* (e.g., method `getDay` of class `Date`) may be new methods not inherited from `Object`.
- This is why **Line 8** would fail to compile, and may be fixed using an explicit **cast**:

```
isWednesday = ((Date) b.get("Yuna")).getDay() == 4;
```

- But what if the *dynamic type* of the returned object is not a `Date`?

```
isWednesday = ((Date) b.get("Suyeon")).getDay() == 4;
```

- To avoid such a `ClassCastException` at runtime, we need to check its *dynamic type* before performing a cast:

```
if (b.get("Suyeon") instanceof Date) {  
    isWednesday = ((Date) b.get("Suyeon")).getDay() == 4;  
}
```

Motivating Example: Observations (2.1)

- It seems: combining *instanceof check* and *type cast* works.
- Can you see any potential problem(s)?
- **Hints:** What happens when you have a large number of records of distinct *dynamic types* stored in the book (e.g., Date, String, Person, Account, *etc.*)?

Motivating Example: Observations (2.2)

Imagine that the tester code (or an application) stores 100 different record objects into the book.

- All of these records are of *static type* `Object`, but of distinct *dynamic types*.

```
Object rec1 = new C1(); b.add(..., rec1);  
Object rec2 = new C2(); b.add(..., rec2);  
...  
Object rec100 = new C100(); b.add(..., rec100);
```

where classes `C1` to `C100` are **descendant classes** of `Object`.

- **Every time** you retrieve a record from the book, you need to check “exhaustively” on its *dynamic type* before calling some method(s).

```
Object rec = b.get("Jim");  
if (rec instanceof C1) { ((C1) rec).m1; }  
...  
else if (rec instanceof C100) { ((C100) rec).m100; }
```

- Writing out this list multiple times is tedious and error-prone!

Motivating Example: Observations (3)

We need a solution that:

- Saves us from explicit `instanceof` checks and type casts
- Eliminates the occurrences of `ClassCastException`

As a sketch, this is how the solution looks like:

- When the user declares a `Book` object `b`, they must *commit to the kind of record that `b` stores at runtime*.
e.g., `b` stores either `Date` objects only or `String` objects only, but *not a mix*.
- When attempting to store a new record object `rec` into `b`, what if `rec`'s *static type* is not a **descendant class** of the type of book that the user previously commits to?
⇒ A *compilation error*
- When attempting to retrieve a record object from `b`, there is *no longer a need to check and cast*.
∴ *Static types* of all records in `b` are guaranteed to be the same.

Parameters

- In mathematics:
 - The same *function* is applied with different *argument values*.
e.g., $2 + 3$, $1 + 1$, $10 + 101$, *etc.*
 - We **generalize** these instance applications into a definition.
e.g., $+: (\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z}$ is a function that takes two integer **parameters** and returns an integer.
- In Java programming:
 - We want to call a *method*, with different *argument values*, to achieve a similar goal.
e.g., `acc.deposit(100)`, `acc.deposit(23)`, *etc.*
 - We generalize these possible method calls into a definition.
e.g., In class `Account`, a method `void deposit(int amount)` takes one integer **parameter**.
- When you design a mathematical function or a Java method, always consider the list of **parameters**, each of which representing a set of possible *argument values*.

Java Generics: Design of a Generic Book

```
class Book <E> {  
    String[] names;  
    E [] records;  
    /* add a name-record pair to the book */  
    void add (String name, E record) { ... }  
    /* return the record associated with a given name */  
    E get (String name) { ... } }
```

Question: Which line has a type error?

```
1 Date birthday; String phoneNumber;  
2 Book<Date> b; boolean isWednesday;  
3 b = new Book<Date> ();  
4 phoneNumber = "416-67-1010";  
5 b.add ("Suyeon", phoneNumber);  
6 birthday = new Date(1975, 4, 10);  
7 b.add ("Yuna", birthday);  
8 isWednesday = b.get("Yuna").getDay() == 4;
```

Java Generics: Observations

- In class `Book`:
 - At the class level, we *parameterize the type of records* that an instance of `Book` may store: `class Book<E>` where `E` is the name of a type parameter, which should be *instantiated* when the user declares an instance of `Book`.
 - Every occurrence of `Object` (the most general type of records) is replaced by `E`.
 - As soon as `E` at the class level is committed to some known type (e.g., `Date`, `String`, *etc.*), every occurrence of `E` will be replaced by that type.
- In the tester code of `Book`:
 - In **Line 2**, we commit that the book `b` will store `Date` objects only.
 - **Line 5** now fails to compile. [String is not a Date]
 - **Line 7** still compiles.
 - **Line 8** does *not need* any instance check and type cast, and does *not cause* any `ClassCastException`.

Bad Example of using Generics

Has the following client made an appropriate choice?

```
Book<Object> book
```

NO!!!!!!!!!!!!!!!!!!!!!!!!!!!!

- It allows **all** kinds of objects to be stored.
 - ∴ All classes are descendants of **Object**.
- We can expect **very little** from an object retrieved from this book.
 - ∴ The **static type** of `book`'s items are **Object**, root of the class hierarchy, has the **minimum** amount of features available for use.
 - ∴ Exhaustive list of casts are unavoidable.

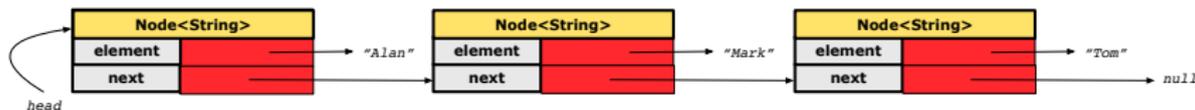
[**bad** for extensibility and maintainability]

Generic Classes: Singly-Linked List (1)

```
public class Node<E> {  
    private E element;  
    private Node<E> next;  
    public Node(E e, Node<E> n) { element = e; next = n; }  
    public E getElement() { return element; }  
    public Node<E> getNext() { return next; }  
    public void setNext(Node<E> n) { next = n; }  
    public void setElement(E e) { element = e; }  
}
```

```
public class SinglyLinkedList<E> {  
    private Node<E> head;  
    private Node<E> tail;  
    private int size = null;  
    public void addFirst(E e) { ... }  
    Node<E> getNodeAt(int i) { ... }  
    ...  
}
```

Generic Classes: Singly-Linked List (2)



Approach 1

```
Node<String> tom = new Node<>("Tom", null);  
Node<String> mark = new Node<>("Mark", tom);  
Node<String> alan = new Node<>("Alan", mark);
```

Approach 2

```
Node<String> alan = new Node<>("Alan", null);  
Node<String> mark = new Node<>("Mark", null);  
Node<String> tom = new Node<>("Tom", null);  
alan.setNext(mark);  
mark.setNext(tom);
```

Generic Classes: Singly-Linked List (3)

Assume we are in the context of class `SinglyLinkedList`.

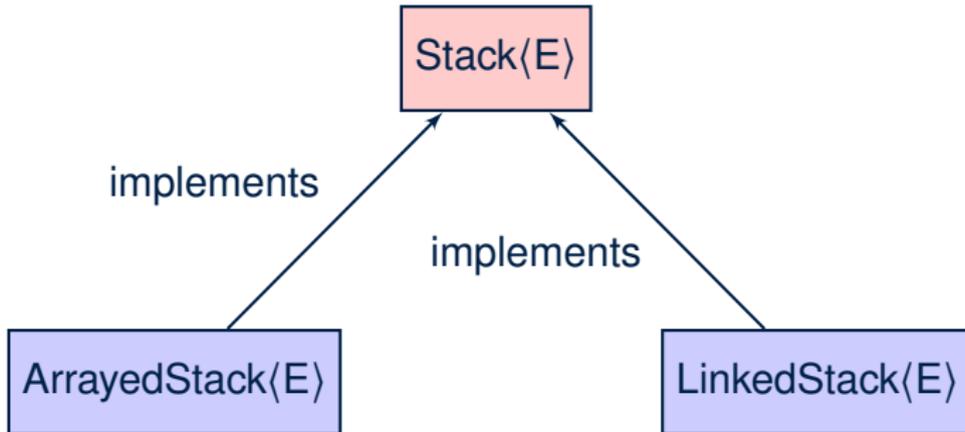
```
void addFirst (E e) {  
    head = new Node<E>(e, head);  
    if (size == 0) { tail = head; }  
    size++;  
}
```

```
Node<E> getNodeAt (int i) {  
    if (i < 0 || i >= size) {  
        throw new IllegalArgumentException("Invalid Index"); }  
    else {  
        int index = 0;  
        Node<E> current = head;  
        while (index < i) {  
            index++; current = current.getNext();  
        }  
        return current;  
    }  
}
```

Generic Stack: Interface

```
public interface Stack< E > {  
    public int size();  
    public boolean isEmpty();  
    public E top();  
    public void push( E e);  
    public E pop();  
}
```

Generic Stack: Architecture



Generic Stack: Array Implementation

```
public class ArrayedStack<E> implements Stack<E> {
    private static final int MAX_CAPACITY = 1000;
    private E[] data;
    private int t; /* top index */
    public ArrayedStack() {
        data = (E[]) new Object[MAX_CAPACITY];
        t = -1; }
    public int size() { return (t + 1); }
    public boolean isEmpty() { return (t == -1); }
    public E top() {
        if (isEmpty()) { /* Error: Empty Stack. */ }
        else { return data[t]; } }
    public void push(E e) {
        if (size() == MAX_CAPACITY) { /* Error: Stack Full. */ }
        else { t++; data[t] = e; } }
    public E pop() {
        E result;
        if (isEmpty()) { /* Error: Empty Stack */ }
        else { result = data[t]; data[t] = null; t--; }
        return result; }
}
```

Generic Stack: SLL Implementation

```
public class LinkedStack<E> implements Stack<E> {
    private SinglyLinkedList<E> data;
    public LinkedStack() {
        data = new SinglyLinkedList<E>();
    }
    public int size() { return data.size(); }
    public boolean isEmpty() { return size() == 0; }
    public E top() {
        if (isEmpty()) { /* Error: Empty Stack. */ }
        else { return data.getFirst(); } }
    public void push(E e) {
        data.addFirst(e); }
    public E pop() {
        E result;
        if (isEmpty()) { /* Error: Empty Stack */ }
        else { result = top(); data.removeFirst(); }
        return result; }
}
```

Generic Stack: Testing Both Implementations

```
@Test
public void testPolymorphicStacks() {
    Stack<String> s = new ArrayedStack<>();
    s.push("Alan"); /* dynamic binding */
    s.push("Mark"); /* dynamic binding */
    s.push("Tom"); /* dynamic binding */
    assertTrue(s.size() == 3 && !s.isEmpty());
    assertEquals("Tom", s.top());

    s = new LinkedStack<>();
    s.push("Alan"); /* dynamic binding */
    s.push("Mark"); /* dynamic binding */
    s.push("Tom"); /* dynamic binding */
    assertTrue(s.size() == 3 && !s.isEmpty());
    assertEquals("Tom", s.top());
}
```

Beyond this lecture ...

- Study <https://docs.oracle.com/javase/tutorial/java/generics/index.html> for further details on Java generics.

Index (1)

Motivating Example: A Book of Objects

Motivating Example: Observations (1)

Motivating Example: Observations (2)

Motivating Example: Observations (2.1)

Motivating Example: Observations (2.2)

Motivating Example: Observations (3)

Parameters

Java Generics: Design of a Generic Book

Java Generics: Observations

Bad Example of using Generics

Generic Classes: Singly-Linked List (1)

Generic Classes: Singly-Linked List (2)

Generic Classes: Singly-Linked List (3)

Generic Stack: Interface

Index (2)

Generic Stack: Architecture

Generic Stack: Array Implementation

Generic Stack: SLL Implementation

Generic Stack: Testing Both Implementations

Beyond this lecture ...

Wrap-Up



EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

What You Learned (1)

- *Procedural Programming in Java*
 - Utilities classes
 - Recursion (implementation, running time, correctness)
- *Data Structures*
 - Arrays
 - Maps and Hash Tables
 - Singly-Linked Lists
 - Stacks and Queues
 - Binary Trees

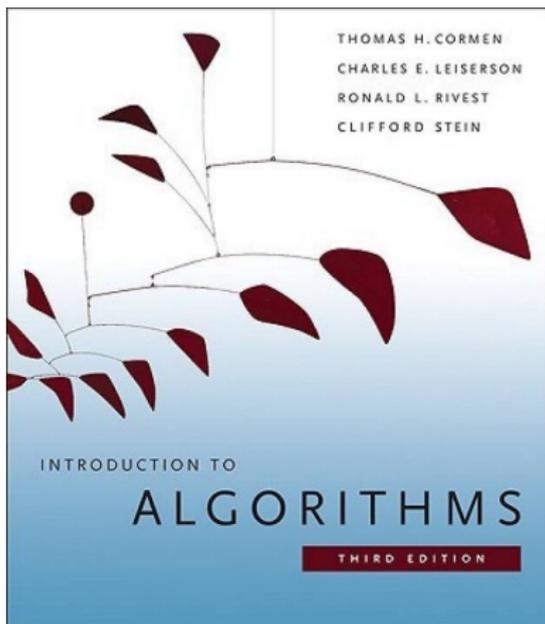
What You Learned (2)

- *Object-Oriented Programming in Java*
 - classes, attributes, encapsulation, objects, reference data types
 - methods: constructors, accessors, mutators, helper
 - dot notation, context objects
 - aliasing
 - inheritance:
 - code reuse
 - expectations
 - static vs. dynamic types
 - rules of substitutions
 - casts and `instanceof` checks
 - polymorphism and method arguments/return values
 - method overriding and dynamic binding: e.g., `equals`
 - abstract classes vs. interfaces
 - generics (vs. collection of `Object`)
 - **keywords:** `private`, `this`, `protected`, `static`, `extends`, `super`, `abstract`, `implements`

What You Learned (3)

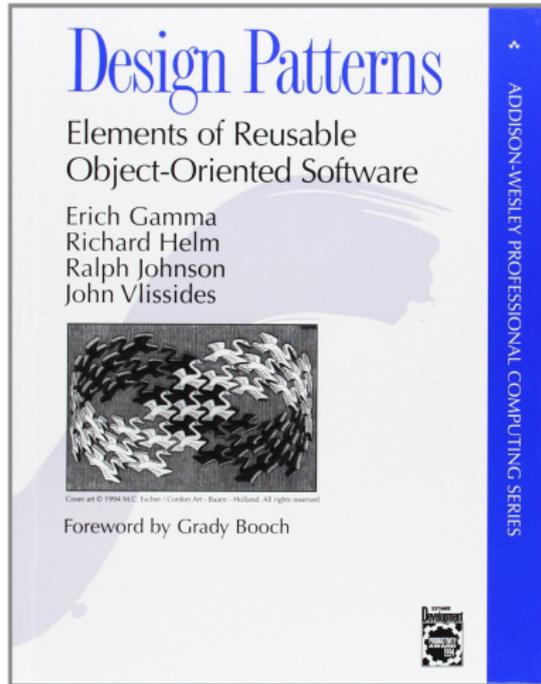
- *Integrated Development Environment (IDE) for Java: Eclipse*
 - Break Point and Debugger
 - Unit Testing using JUnit

Beyond this course... (1)



- *Introduction to Algorithms (3rd Ed.)* by Cormen, *etc.*
- DS by DS, Algo. by Algo.:
 - **Understand** math analysis
 - **Read** pseudo code
 - **Translate** into Java code
 - **Write and pass** JUnit tests

Beyond this course... (2)



- *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, etc.
- Patter by Pattern:
 - **Understand** the problem
 - **Read** the solution (not in Java)
 - **Translate** into Java code
 - **Write and pass** JUnit tests

Beyond this course... (3)

Visit my lectures on *EECS3311 Software Design*:

`http://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS3311_F17`

- Design by Contracts
- Design Patterns
- Program Verification

Wish You All the Best

- What you have learned will be **assumed** in EECS2011.
- Logic is your friend: Learn/Review EECS1019/EECS1090.
- Do **not** abandon Java during the break!!
- As ever, feel free to get in touch and let me know how you're doing :D