Unit 7:

Arrays and Lists; Abstract Class and Interface

Object-Oriented Programming (OOP)
CCIT 4023, 2025-2026

U7: Arrays & Lists; Abstract Class & Interface

- Arrays
- Collections and Lists
  - Class **`ArrayList`**
  - Class **`LinkedList`**
- Applying Polymorphism with Arrays
- Abstract Class and Abstract Method
- Interface

Arrays

- An ***array*** is an indexed collection of data values of the *same data type* (a reference type in Java)
  - E.g. we may use array to deal with 100 integers (`int`), 500 `Account` objects, 12 real numbers (`double`), etc.

- Array Declaration: two ways in Java

```
<data type> [ ] <variable>; //variation 1,recommended in Java
<data type> <variable> [ ]; //variation 2, similar to C
```

- **Array Creation** (with Assignment)

```
<variable> = new <data type> [ <size> ];
```

Object Declaration and Creation (with Assignment)

```
double[] rainfall;
rainfall = new double[12];
```

```
double[] rainfall = new double[12];
```

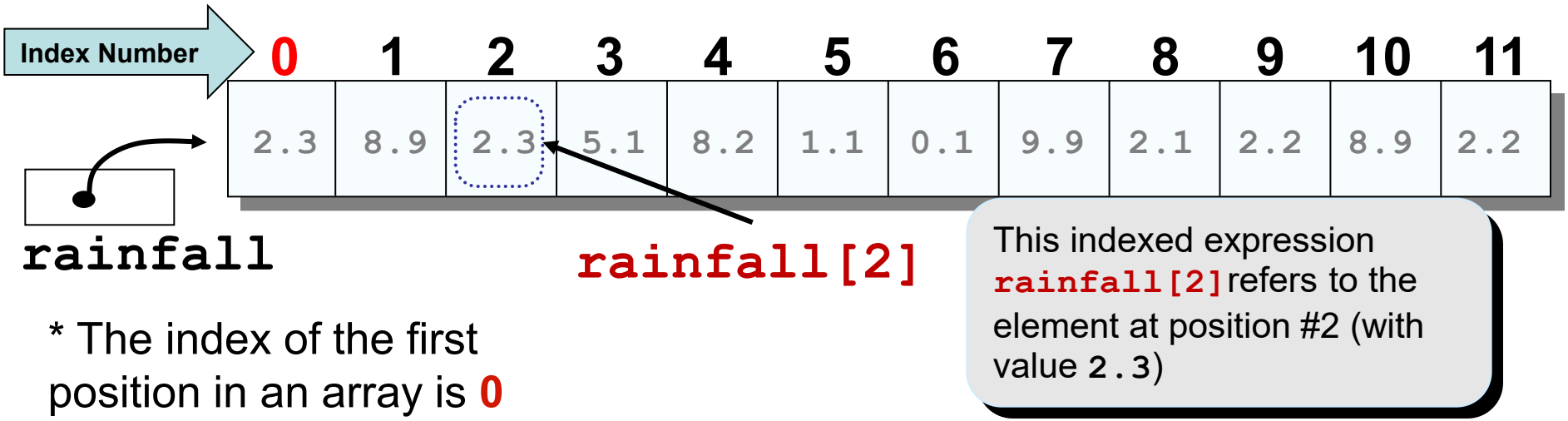
An array is created similar to an object!

Initializing and Accessing Individual Elements

- To ***declare and initialize an array in a statement***, e.g.

```
double[] rainfall = { 2.3, 8.9, 2.3, 5.1, 8.2, 1.1,
                      0.1, 9.9, 2.1, 2.2, 8.9, 2.2 };
String[] students = { "Amy", "Brenda", "Candy" };
```

- Individual elements in an array are accessed with the *indexed expression*, e.g.



This indexed expression `rainfall[2]` refers to the element at position #2 (with value `2.3`)

* The index of the first position in an array is **0**

Example of Accessing Array with `for` Statement

- The following examples compute the average rainfall or display students, with `for` loop

```
double[] rainfall = new double[12];
double   annualAverage, sum = 0.0;
//... Setting elements in array
for (int i = 0; i < rainfall.length; i++)
    sum += rainfall[i]; // Getting value from array
annualAverage = sum / rainfall.length;
```

The public constant `length` returns the size/capacity of an array

```
//... Creating and Setting elements in array of String (student names
System.out.println("Students in our class are:");
for (int i = 0; i < students.length; i++)
    System.out.println("Student " + i + students[i]);
```

** Remark:* *Array of objects is even more powerful.*
More details of Java array will be introduced in later unit.

Array Initialization

- The following examples declare and initialize arrays at the same time

```
int[] num = new int[] { 1, 3, 5 };
int[] number = { 2, 4, 6, 8 };
double[] samplingData = { 2.443, 8.99, 12.3, 45.009, 18.2,
                          9.00, 3.123, 22.084, 18.08 };
String[] monthName = { "January", "February", "March",
                       "April", "May", "June", "July",
                       "August", "September", "October",
                       "November", "December"  };

// num = { 123, 456, 789 }; // ERROR, NOT valid
// the following is required, to assign an existing variable
number = new int[]{ 12, 34, 56, 78 }; // OK, valid, correct form
```

number.length → **4**

samplingData.length → **9**

monthName.length → **12**

Array of Objects

- In Java, in addition to arrays of primitive data types such as array of integers, we can declare ***arrays of objects***
- An array of objects is even more powerful
- The use of an array of objects allows us to model the application more cleanly and logically
- Example: suppose we are having a class `Person` and a collection of objects of this class type to illustrate the use of an array of objects
  - The code below handles only **one `Person` object**

```
Person amy;
amy = new Person( ); //create a new Person object
amy.setName( "AU Amy"   ); //call methods to set values
amy.setAge(  18   );
amy.setGender( 'F' );
```

Creating an Object Array - 1

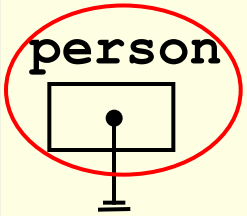
- Code below handles an **array of `Person` objects** (with size 20)

Code (A)

```
Person[ ]  person;
person = new Person[20];
person[0] = new Person( );
```

Only name `person` is declared with type of `Person` array, but no actual array is allocated yet

State of Memory



After (A) is executed

Creating an Object Array - 2

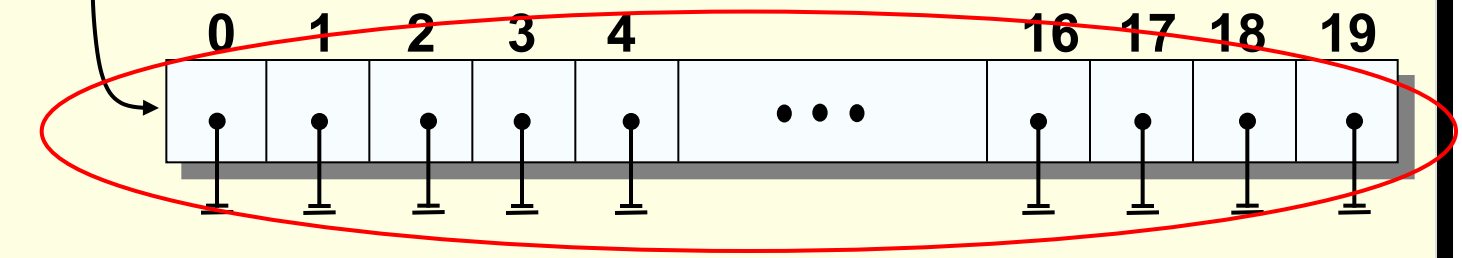
Code

B

```
Person[ ]  person;
person = new Person[20];
person[0] = new Person( );
```

Now the array for storing `20` `Person` objects is created, but the `Person` objects themselves are not yet created

State of Memory



After B is executed

Creating an Object Array - 3

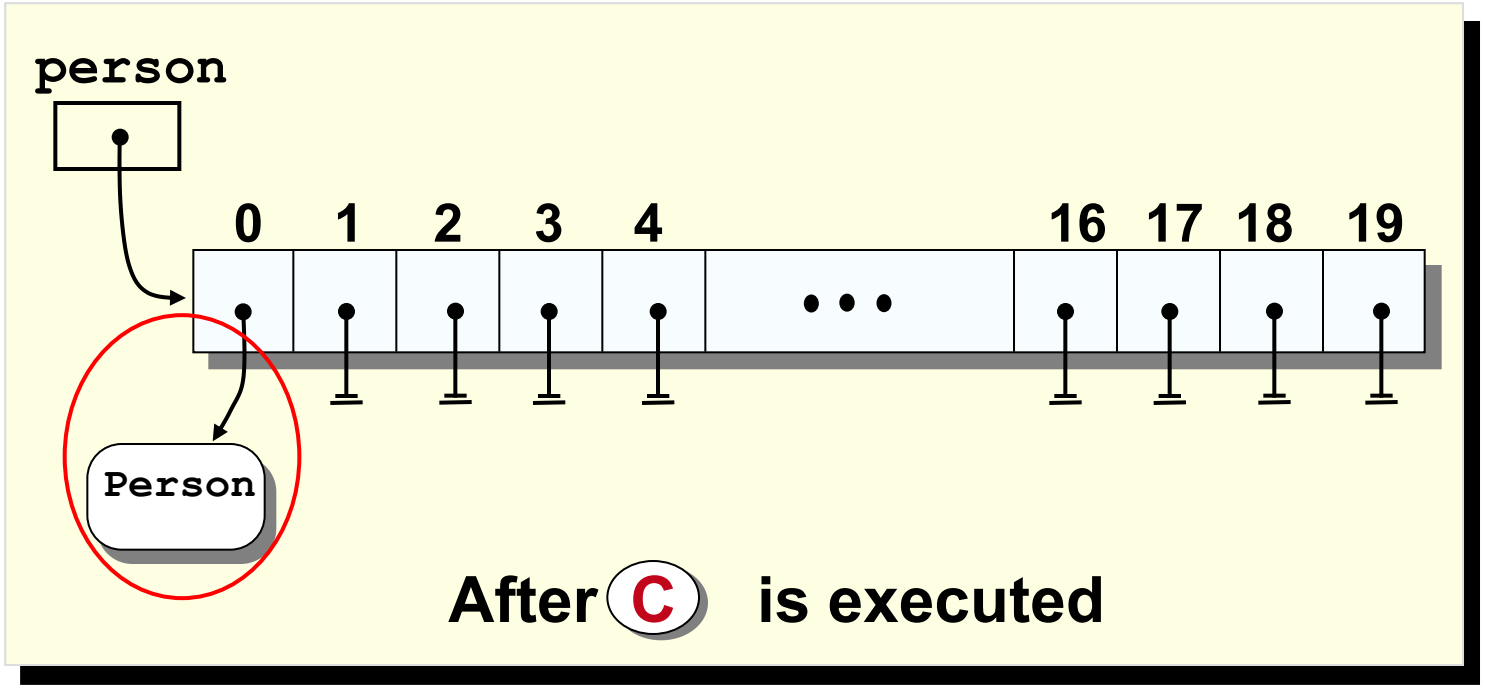
Code

C

```
Person[ ]  person;
person = new Person[20];
person[0] = new Person( );
```

One `Person` object is created and reference to this object is assigned to the position index `0`

State of Memory



Person Array Processing – Sample 1

- ***Create/new an array*** of `Person` objects
  - Declare and create `Person` array (with size 20)
  - Create `Person` objects & assign to array elements with `for` loop

```
Person[] person = new Person[20]; //declare and create Person array
String   name, inpStr;
int      age;
char     gender;

for (int i = 0; i < person.length; i++) {
    name   = inputBox.getString("Enter name:"); //read in values
    age    = inputBox.getInteger("Enter age:");
    inpStr = inputBox.getString("Enter gender:");
    gender = inpStr.charAt(0);

    person[i] = new Person( );    //create a new Person object
    person[i].setName   ( name   ); //call methods to set value
    person[i].setAge    (  age   );
    person[i].setGender( gender );
}
```

Person Array Processing – Sample 2

- ***Access elements in an array*** of `Person` objects
  - Find the youngest and oldest persons in the array of `Person` objects

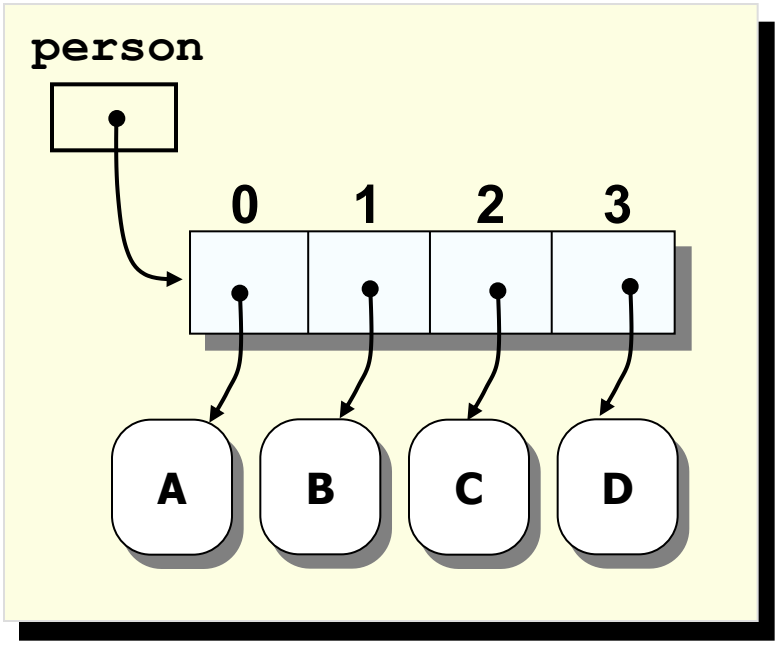
```
int minIdx = 0;  //index to the youngest person
int maxIdx = 0;    //index to the oldest person


for (int i = 1; i < person.length; i++) {
    if ( person[i].getAge() < person[minIdx].getAge() ) {
        minIdx  = i;    //found a younger person
    } else if (person[i].getAge() > person[maxIdx].getAge() ) {
        maxIdx  = i;    //found an older person
    }
}
//person[minIdx] is the youngest and person[maxIdx] is the oldest
```

Removing Object from Array – 1

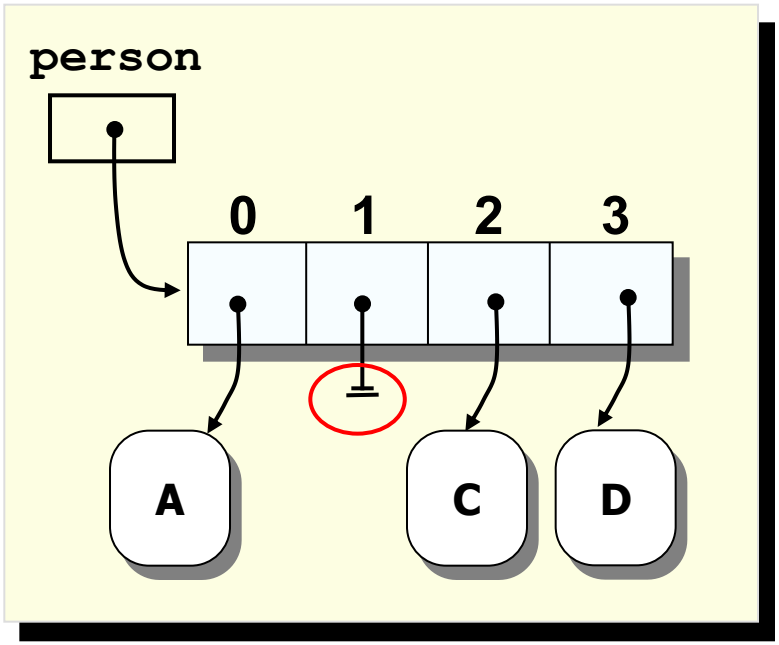
- It is common to **remove object** from an array, ***by setting it to `null`***

```
int delIdx = 1;
A person[delIdx] = null;
```

Delete `Person B` by setting its reference in position `1` to `null`



Before A is executed



After A is executed

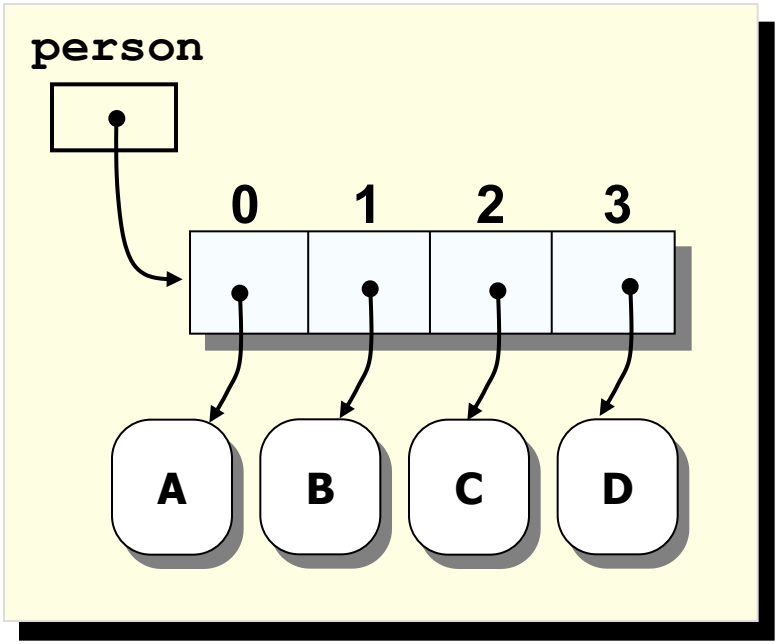
Removing Object from Array – 2

- We may also **remove object** from array, by ***re-referencing*** **it**

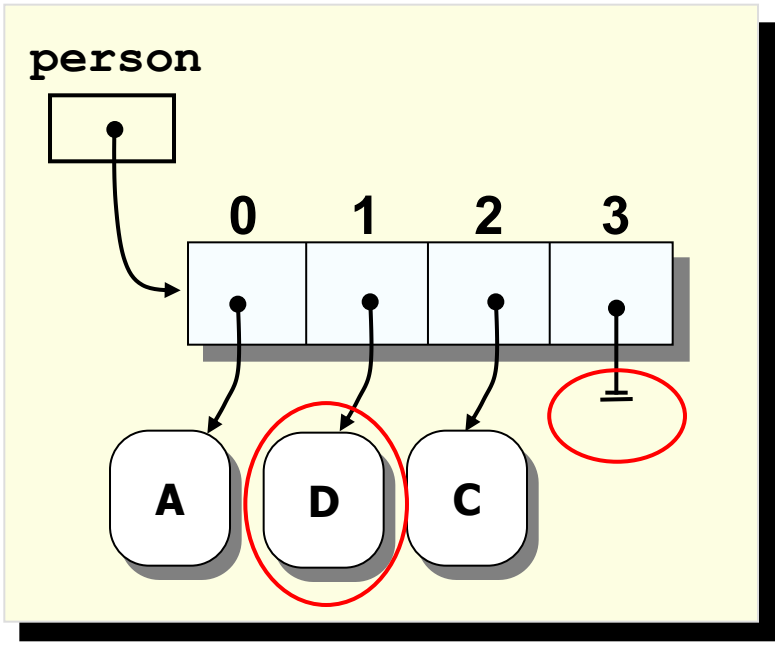
B

```
int delIdx = 1, last = 3;
person[delIndex] = person[last];
person[last]     = null;
```

Delete `Person B` by re-referencing (setting the reference in position `1` to the last person)



Before B is executed



After B is executed

Garbage Collection
(Memory Management in Java)

- When an object has no reference pointing to it, the system will automatically erase the object and make the memory space available for other users
- This process/mechanism of freeing/de-allocating memory in Java is called ***Garbage Collection***
- The previous operation of removing the array element object `Person B` will not erase the object by itself
- If this object is no more referenced by any others, it initiates a chain reaction for garbage collection, which is then done automatically by JVM

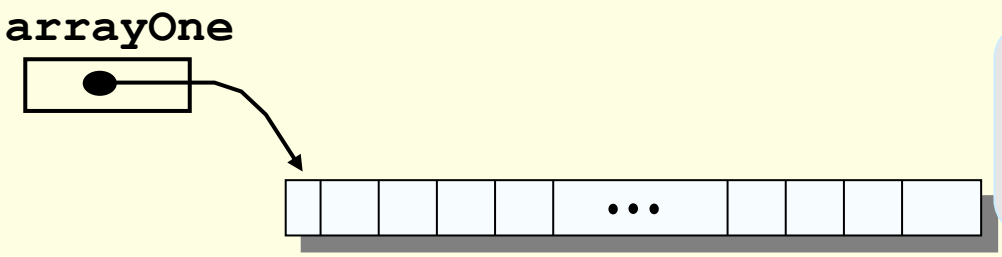
Passing Arrays to Methods - 1

- Pass array as an argument / parameter into a method is similar to passing object into a method, we pass the *reference* of the array.

```
minOne =
searchMinimum(arrayOne);
```
A

```
public int
searchMinimum(float[] number))
{
    //...
}
```

At A before `searchMinimum`



State of Memory

A. Local variable number does not exist before the method execution

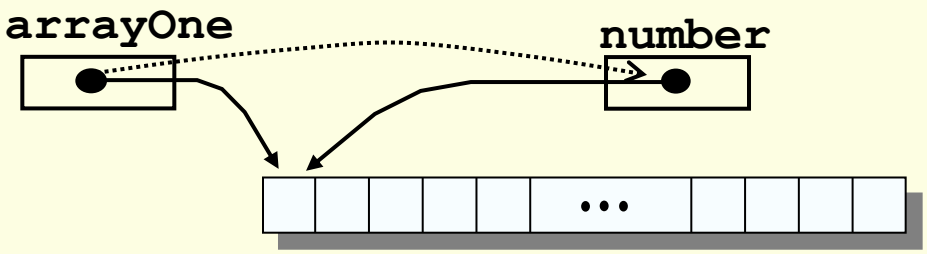
Passing Arrays to Methods - 2

```
minOne =
searchMinimum(arrayOne);
```

```
public int
searchMinimum(float[] number))
{
    //...
}
```

B

The address is copied at B



State of Memory

B. The value of the argument, which is an address, is copied to the parameter

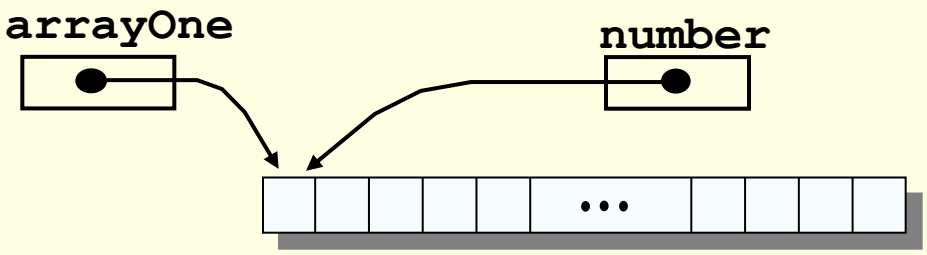
Passing Arrays to Methods - 3

```
minOne =
searchMinimum(arrayOne);
```

```
public int
searchMinimum(float[] number))
{
    //...
}
```

C

While at C inside the method



State of Memory

C. The array is accessed via `number` inside the method

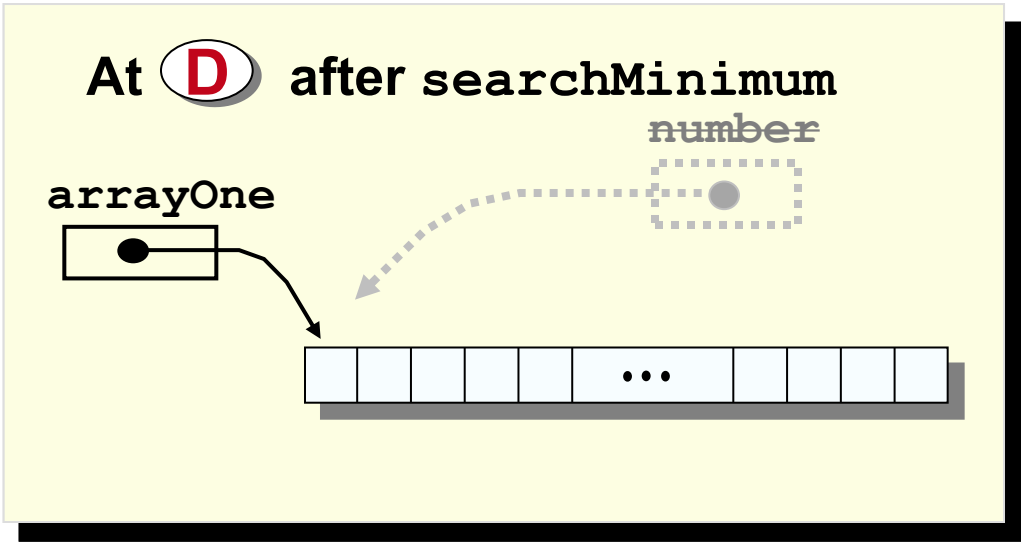
Passing Arrays to Methods - 4

```
minOne =
searchMinimum(arrayOne);
```

D

```
public int
searchMinimum(float[] number))
{
    //...
}
```

State of Memory



D. The parameter is erased, but the argument still points to the same object

Two-Dimensional Arrays

- Two-dimensional arrays are useful in representing tabular information.
  - E.g. Weather data in 12 days

Days	Pressure	Temperature	Humidity
1	1022.4	16	72
2	1022	17.1	77
...	...	...	...
12	1017.9	20.6	89

- ***Declaration***

```
<data type> [][] <variable>     //variation 1
<data type> <variable>[][]      //variation 2
```

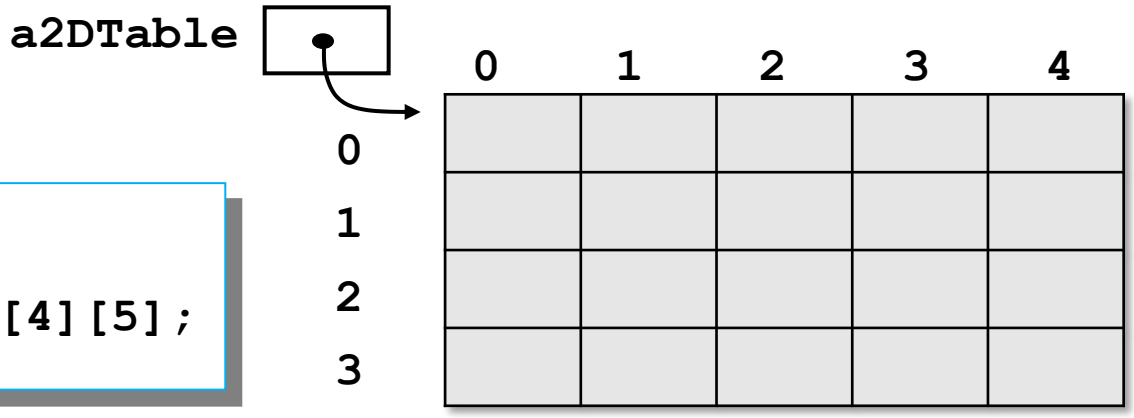
- ***Creation***

```
<variable>  = new <data type>[<# of row>][<# of col>]
```

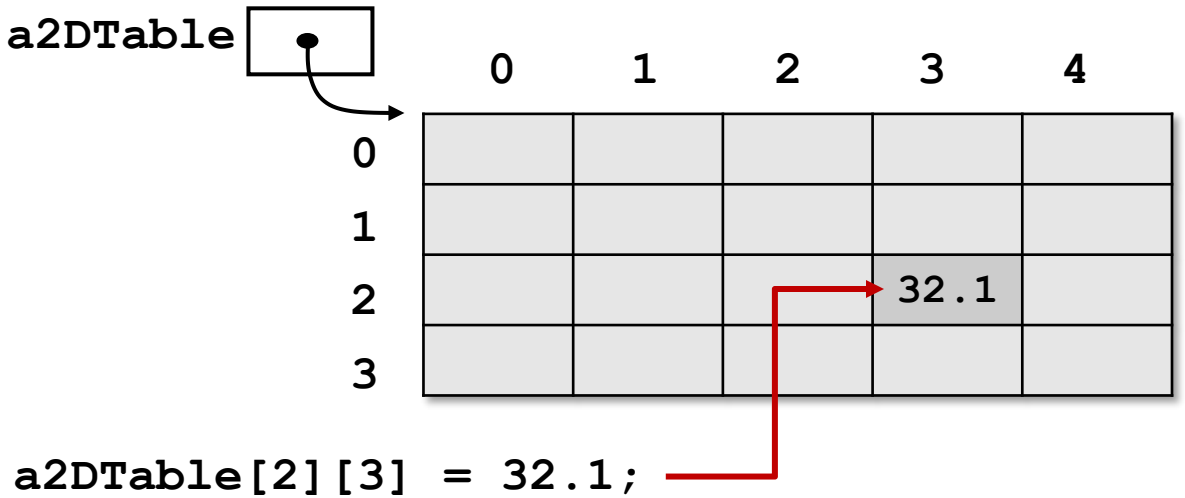
Declaring and Creating a 2-D Array

- *Example :*

```
double[][] a2DTable;
a2DTable = new double[4][5];
```



- An element in a two-dimensional array is accessed by its row and column index:



Sample 2-D Array Processing

- It is common to use nested `for`-loop to access elements in 2D array
  - E.g. Find the average of each row of the 2D array

```
double[][] a2DTable = new double [4][5];
// ... Suppose set elements in 2D array HERE
double[] average = { 0.0, 0.0, 0.0, 0.0 };
for (int i = 0; i < a2DTable.length; i++) { // each row
    for (int j = 0; j < a2DTable[i].length; j++) {// each col
        average[i] += a2DTable[i][j];
    }
    average[i] = average[i] / a2DTable[i].length; // row average
}
```

Multidimensional Arrays

- Similar to 2D array (which consists of an array of 1D arrays), a 3-D array consists of an array of 2D arrays
  - E.g. use a 3D array as the example below to handle marks of many *exercises* of many *students* in many *classes*
  - we may further handle higher dimensional arrays in the similar way

```
// An Example of a 3D array
double[][][] sExMarks = new double[CLASSMAX][STUDENTMAX][EXMAX];
double[][][] sAvgMarks = new double[CLASSMAX][STUDENTMAX];
// ...
for (int i = 0; i < sExMarks.length; i++) { // loop for CLASS
    for (int j = 0; j < sExMarks[i].length; j++) { // for STUDENT
        double total = 0;
        for (int k = 0; k < sExMarks[i][j].length; k++) //for Exercise
           total += sExMarks[i][j][k];
        sAvgMarks[i][j] = total / k; // find average mark of student
    }
}
```

Collections and Lists

- Array is a *fixed-size* data structure.
  - Once an array is created, its capacity *cannot* be changed
- When we add an element to an array that does not have any unused position, we will have array overflow
- *Solution 1 (Bad):* Declare a larger array to avoid overflow, but it will end up underutilizing the space
- *Solution 2:* Using more flexible types, such as Collection and List (including `ArrayList` and `LinkedList`)

Collections and Lists

- Collection represents a group of objects known as its elements
  - The Java **`Collection`** *interface* is used to pass around collections of objects where generality is desired
  - Some sub-interfaces include `List`, `Queue`, and `Set` with specific features
- **`List`** is an ordered `Collection` (sometimes called a sequence)
  - Lists may contain duplicate elements
- Implementing classes of the interface `List` includes **`ArrayList`** and **`LinkedList`**
  - In the package **`java.util`**

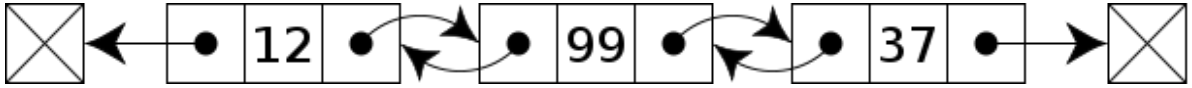
Class `ArrayList`
(In the package `java.util`)

- **`ArrayList`** is a ***"resizable-array"*** implementation of the List interface
  - An array that automatically reallocates its capacity
- As elements are added to the `ArrayList`, its capacity grows automatically
- There are many implemented methods supporting list operations, including method `add()` which appends the specified element to the end of the list.

```
ArrayList strArrList;
strArrList = new ArrayList();
// Objects (String) are added/appended to list below
strArrList.add( new String("This String") );
strArrList.add( new String("That String") );
```

Class `LinkedList`
(In the package `java.util`)

- **`LinkedList`** is the implementation of the List interface, based on ***doubly-linked structure***. Each element is contained in a linked node.



- Similar to `ArrayList`, many implemented methods support list operations, including method `add()` which appends the specified element to the end of the list.

- `ArrayList` and `LinkedList` could be used almost in the same way, with different performances. In general, `ArrayList` is comparatively faster in random access (get/set items), but slower in manipulation (insert/remove items) than `LinkedList`.

```
LinkedList strLList = new LinkedList();
// Objects (String) are added/appended to list below
strLList.add( new String("This String") );
strLList.add( new String("That String") );
```

Some Common Methods of `ArrayList` & `LinkedList`

- **Add** to the end

  `myList.add(E elt)`

- **Remove** at specified position

  `myList.remove(int index)`

- **Check** if list contains an object

  `myList.contains(Object obj)`

- **Size** (number of elements)

  `myList.size()`

- **Convert** and return an array

  `myList.toArray()`

- **Add** at specific position

  `myList.add(int index, E elt)`

- **Clear** / Remove All elements

  `myList.clear()`

- **Get**

  `myList.get(int index)`

- **Set /** Replace an element

  `myList.set(int index, E elt)`

** Check more methods with API Documents.*

`ArrayList` with **Generics**

- There are no restrictions on the type of objects added into a typical list, e.g. with `ArrayList` (example below)
  - This is often undesired in most applications

```
ArrayList myAList = new ArrayList();
myAList.add( new String("248") ); // String Object can be added
myAList.add( new Integer(248) );  // Integer Object can also be added
// * Warning may be given in compilation, e.g. unchecked or unsafe operations
```

- To restrict a specific type (e.g. `ArrayList` of `String`) and improve program reliability, we may use it with Generics. See example codes below:

```
ArrayList<String> myGAList = new ArrayList<String>();
myGAList.add(new String("248")); // String Object can be added
myGAList.add(new Integer(248));//ERR: Non-String Object cannot be added
```

** Refer external resources for more details.*

Applying Polymorphism with Arrays

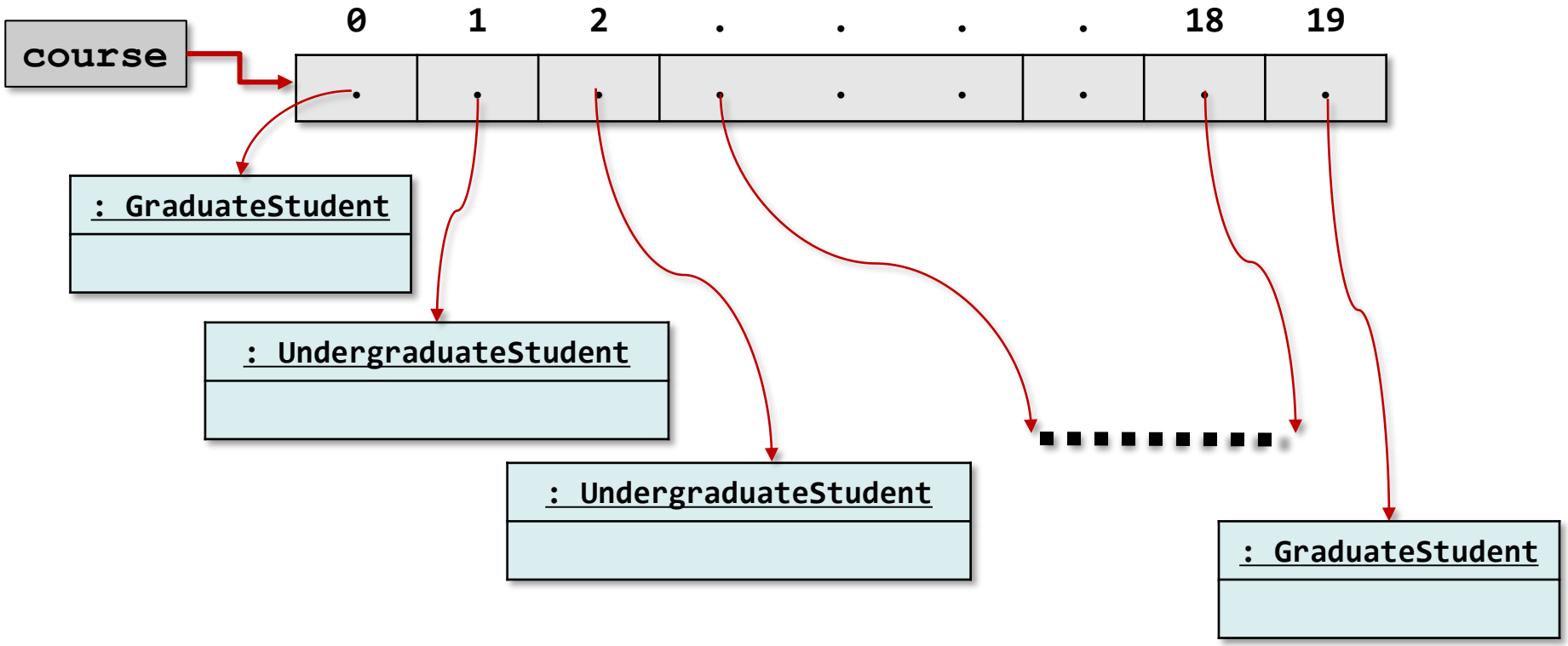
- We can maintain a `course` (consisting of different students) using an array of `Student`, combining objects of its subclasses `UndergraduateStudent`, and `GraduateStudent` instead
  - Based on subtype polymorphism, object of a superclass type (`Student`) can refer to that of its subclasses (`UndergraduateStudent` and `GraduateStudent`)
  - This can be applied to elements of array in the same manner.

```
Student[] course = new Student[20];
//...
course[0] = new GraduateStudent();
course[1] = new UndergraduateStudent();
course[2] = new UndergraduateStudent();
//...
course[19] = new GraduateStudent();
```

Mixed Objects in an Array

- The `course` array contains elements referring to instances of `GraduateStudent` or `UndergraduateStudent` classes

```
Student[] course = new Student[20];
course[0] = new GraduateStudent();
// ...
```



Mixed Objects in an Array

- To compute the course grade using the `course` array, we execute

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

- If `course[i]` refers to a **`GraduateStudent`**, then the `computeCourseGrade()` method of the `GraduateStudent` class is executed
- If `course[i]` refers to an **`UndergraduateStudent`**, then the `computeCourseGrade()` method of the `UndergraduateStudent` class is executed

Type Checking: `getClass()` Method

- The `getClass()` method returns the runtime class of the object
  - As such, we may use this method to compare an object class type
- The following code counts the number of undergraduate students

```
int undergradCount = 0;
for (int i = 0; i < course.length; i++) {
    if ( course[i].getClass() == UndergraduateStudent.class ){
        undergradCount++;
    }
}
```

Type Comparison Operator: `instanceof`

- The `instanceof` operator compares an object to a specified type
  - It will return `true`, if an object is 1) an instance of a ***class***, 2) an instance of a ***subclass***, or 3) an instance of a class that implements a particular ***interface***.
  - For example, given:

```
class SuperC {}
class SubC extends SuperC implements InterF {}
interface InterF {}
```

```
SuperC superObj;
SubC subObj;
System.out.println( superObj instanceof SuperC ); // true
System.out.println( superObj instanceof SubC ); // false
System.out.println( subObj instanceof SuperC ); // true
System.out.println( subObj instanceof SubC ); // true
System.out.println( subObj instanceof InterF ); // true
```

Using `instanceof` Operator

- The `instanceof` operator can help us learn the class of an object
- The following code also counts the number of undergraduate students

```
int undergradCount = 0;
for (int i = 0; i < course.length; i++) {
    if (course[i] instanceof UndergraduateStudent ) {
        undergradCount++;
    }
}
```

Casting Objects

- Given `Student` is superclass of `GraduateStudent` class, ***implicit casting*** can be applied as follows:

```
Student sRep = new GraduateStudent();//ok, implicit cast
```

  - so that an instance of `GraduateStudent` is automatically casted (*implicit casting*) as an instance of `Student`

- It is NOT valid if we later write (although we know `sRep` is referring to `GraduateStudent`):

```
GraduateStudent gradS = sRep; // ERR: PROBLEM
```

  - because `sRep` (original type with class `Student`) is not known to the compiler to be a `GraduateStudent`
  - Instead, we can tell the computer (explicitly) that we need to convert an `Student` to `GraduateStudent`, by ***explicit casting***:

```
GraduateStudent gradS = (GraduateStudent) sRep;
```

Abstract Class and Abstract Method

- There are situations we may want to define a class, which we would not create their objects
  - This often happens when we design certain classes containing abstract method, method with name and signature but without method body implementation.
  - Abstract class is often used for defining certain superclasses
- An ***abstract class*** is such a class which is defined with the keyword modifier `abstract` in Java
- **No instances can be created** from an abstract class, but it can be "subclassed"

Abstract Class and Abstract Method

- An **abstract method** is a method with the keyword **`abstract`**, and it ends with a semicolon instead of a method body
  - `private`, `static`, and `final` methods cannot be declared as abstract (Compile-time error)
- If a class has abstract methods, the class *must* be declared abstract:

```
public abstract class GraphicObject { //abstract class
        // fields
        // non-abstract methods
        abstract void draw(); //abstract method
}
```

- The subclass of an abstract class usually provides implementation for all `abstract` methods in its abstract superclass
  - If it does not, the subclass must also be declared abstract
- Depending on whether we need to create instances of the superclass, we would define the class differently

Abstract Superclass & its Subclass

- With the example case based on `Student` superclass defined earlier
  - `Student` is the superclass and `Undergraduate` or `Graduate` are the subclasses. Should the `Student` be abstract?
  - Choosing approach depends on different situations
- When considering design options, we may ask ourselves the following:
  - which one better reflects the real scenarios,
  - which one better avoids potential mistakes,
  - which one allows easier modification, future extension, etc.
- ***Possible Approach 1:*** We can define the `Student` class so that is can be instantiated (non-abstract) for possible future extension, even though all `Student` must be either `Undergraduate` or `Graduate` at the moment
- ***Possible Approach 2:*** We may instead define the `Student` class *abstract* at the moment, and introduce a new subclass (e.g. say `OtherStudent`) in the further extension if necessary.

Interface

- A Java **interface** is a reference type that can *only* contain constants, and method signatures
  - Method signatures have no method bodies. (Methods in an *interface* are *implicitly* public abstract, and `public abstract` modifiers may be omitted)

    *** Newer Java version is a bit different.*
  - An *abstract method* is a method with the keyword **`abstract`**, and it ends with a semicolon instead of a method body, e.g.:

    ```
    public abstract double computeCourseGrade();
    ```

    - `private`, `static`, and `final` methods cannot be declared as abstract

- Typically an interface contains a group of related methods without method bodies, and these methods define certain behaviors for the related classes (and their objects) that they promises to provide for interfacing and interacting with outside world.

- To use an interface, the related classes **`implements`** the interface.

Syntax for Interface

- **Interface Declaration** (similar to Class declaration):

```
<modifier> interface <interface-name>{
    // public fields(constants)
    // method signatures (public abstract methods)
}
```

- A **class that implements interface(s)**:

```
<modifier> class <classname> implements <interface-name(s)>
{
    // implement (abstract) methods declared in interface
    // other fields and methods
}
```

Java Interface Example

```
// declare an interface
public interface WorkHard {
   int MIN_WORKING_HR = 12;    // implicitly a final (constant)
   public void spendTime(int sT);// implicitly public abstract method
   public void concentrate();
}

// declare a class that implements an interface
public class HardWorker implements WorkHard {
// HERE implements the interface method(s)
   public void spendTime(int sT){
        //... implements the method (define the method body here)
   }
   public void concentrate(){
        //... implements the method (define the method body here)
   }
//...
}
```

Inheritance vs. Interface

- A Java class can ***inherit/extend from only one class*** but it can ***implement more than one interface***
- When designing a Java program, we often need to choose if we model certain modules with inheritance or interface
  - It is common that if the entity A and B have a *strong is-a relationship* that clearly describes a parent-child relationship, we should consider to model them with inheritance
  - Otherwise interface better represents a *weak is-a relationship*, and is more flexible than class

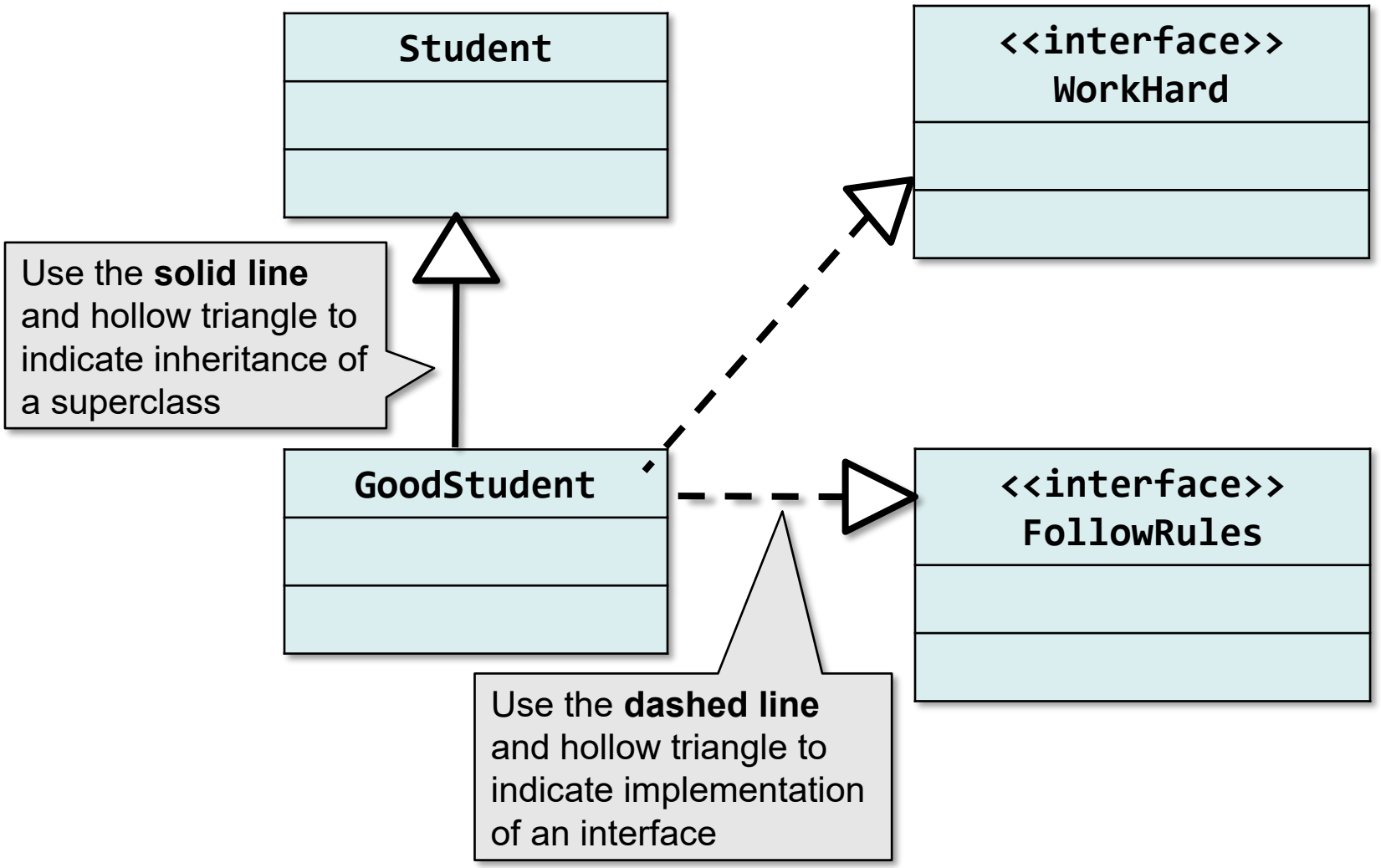
Abstract Class vs. Interface

	Fields	**Constructors**	**Methods**
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator	No restrictions
Interface	All variables are implicitly `public static final`	No constructors. An interface cannot be instantiated using the new operator	All methods must be `public abstract` instance methods

Implement Multiple Interfaces

```
public interface WorkHard { // declare an interface
    int MIN_WORKING_HR = 12;        // implicitly a final (constant)
    public void spendTime(int sT); // implicitly abstract method
    public void concentrate();
}

public interface FollowRules {// declare another interface
    public void followRegulation (String s);
}

// below declare a class that implements 2 interfaces
public class GoodStudent extends Student
                          implements WorkHard, FollowRules {
    public void spendTime(int sT){ //... implements the method
    }
    public void concentrate(){ //... implements the method
    }
    public void followRegulation (String s){ //... implements method
    }
}
```

UML Notations



Abstract Class Implements Interface

- You can define a class that does not implement all of the interface methods, provided that the class is declared to be *abstract*, e.g.

```
public abstract class ABitHardWorker implements WorkHard {
        public void spendTime(int sT){ //... implements method
        }
}
```

```
public class MoreHardWorker extends ABitHardWorker {
        public void concentrate(){ //... implements the left method
        }
}
```

- In this case, class `ABitHardWorker` must be abstract because it does not fully implement `WorkHard`, but class `MoreHardWorker` does implement the rest method
  - Class `MoreHardWorker` can be a “non-abstract” class

Using Interface in GUI

- Using Java interface is important when producing applications with GUI (Graphical User Interface)
- ***Java GUI interaction*** is done with ***event handling using interfaces***
  - E.g. the interface `ActionListener` for handling a button click event:

```
public class ButtonHandler implements ActionListener {
// implementing method of ActionListener interface below
        public void actionPerformed(ActionEvent event) {
//... do something, if the registered button got clicked
        }
}
```

* Ref: https://docs.oracle.com/en/java/javase/20/docs/api/java.desktop/java/awt/event/ActionListener.html

* *Details of GUI will be discussed in later unit*

Syntaxes for Class Declaration (RECAP)

- Class in a Basic Form:

```
<modifier(s)> class <class name> {  <class body>  }
```

- Class inherits from a specified (direct) superclass:

```
<modifier(s)> class <class name> extends <superclass name> {
     <class body>
}
```

  - Modifiers `<modifier(s)>` (e.g. `public`, `private` which determine what other classes can access)
  - Class body `<class body>` is surrounded by a brace pair `{}`
  - Class name `<class name>`, with initial letter capitalized by convention, preceded by the keyword `class`
  - Name of the class's parent ("direct" superclass) `<superclass name>`, if any, preceded by the keyword `extends`
    - If not explicitly designate the "direct" superclass with the `extends` clause, the class's superclass is class `Object` (the root) in Java as in the first Basic Form

Syntaxes for Class Declaration (RECAP)

- Class implements interface(s):

```
<modifier(s)> class <class name> implements <interface(s)> {
      <class body>     }
```

- Class inherits from its specified (direct) superclass and implements interface(s):

```
<modifier(s)> class <class name> extends <superclass name>
      implements <interface(s)> {
      <class body>
}
```

  - A comma-separated list of interfaces `<interface(s)>` implemented by the class, if any, preceded by the keyword `implements`, a class can *implement* more than one interface

E.g.: Below declares a class `SubC`, which is the subclass of class `SuperC` (its direct superclass) and also implements two interfaces (`InA`, `InB`)

```
public class SubC extends SuperC implements InA, InB { //...
```

** Details of the topic will be further discussed in later unit*

References

- This set of slides is only for educational purpose.

- Part of this slide set is referenced, extracted, and/or modified from the followings:
  - Deitel, P. and Deitel H. (2017) “Java How To Program, Early Objects”, 11ed, Pearson.
  - Liang, Y.D. (2017) “Introduction to Java Programming and Data Structures”, Comprehensive Version, 11ed, Prentice Hall.
  - Wu, C.T. (2010) “An Introduction to Object-Oriented Programming with Java”, 5ed, McGraw Hill.
  - Oracle Corporation, “Java Language and Virtual Machine Specifications” https://docs.oracle.com/javase/specs/
  - Oracle Corporation, “The Java Tutorials” https://docs.oracle.com/javase/tutorial/
  - Wikipedia, Website: https://en.wikipedia.org/