# Last Lecture

- Data structure
  - Primitive data type
  - Simple abstract data type (ADT)
    - Array
    - List, Set, Map
- Create your own abstract data type (ADT) and put it into practice

# Take home exercise: Complex number

```java
public class Complex {

  private double real;
  private double imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = imag;
  }

  public Complex() {
    this(0.0, 0.0);
  }

  public void add(Complex complex) {
    this.real += complex.real;
    this.imag += complex.imag;
  }

  public void multiply(Complex complex) {
    this.real = this.real * complex.real - this.imag * complex.imag;
    this.imag = this.real * complex.imag + this.imag * complex.real;
  }

  // + all getters and setters

}
```

How to resolve the problem?

**What is the output of the following code segment?**

```java
Complex complex1 = new Complex(1, 2) ;
Complex complex2 = new Complex(1, 2) ;
System.out.println("complex1.real=" + complex1.real);
System.out.println("complex1.imag=" + complex1.imag);
System.out.println( (complex1 == complex2) );
```

- When creating an ADT, we have to tell the computer how to compare two objects
- Different programming languages have different approaches
- For example
  - In C++, we can overload the operators such as $+$, $-$, $==$, $<$, $>$, etc., by providing the operators with a special meaning for a data type *without* changing its original meaning

## In Java

- Every class (except primitives) is a subclass of the `Object` class
  - Allows every common methods and attributes (properties) to be available upon its creation, e.g.:
    - `clone()` – create an exact copy of the object
    - `hashCode()` – an integer representing, also known as *hash code*, of the object
    - `equals()` – check whether two objects has the same hash code
    - `toString()` – a string representation of the object
- To determine whether two objects are equal, we can override the `equals()` method, or create a new method to perform the task
- Similarly, we can override the `toString()` method to get a meaningful presentation of the object

# CPT108 Data Structures and Algorithms

Lecture 6-7

Data Structures and Abstract Data Type

Data Abstraction

# Data-directed design

- Design directed by the choice and representation of data structures
- Data requirements:
  - In addition to the getters and setters methods, what functions to be performed on the data
  - What's the proper scope
    - Ownership?
      – who owns the data
    - How is it ***shared***?

# Data-directed design: Objects

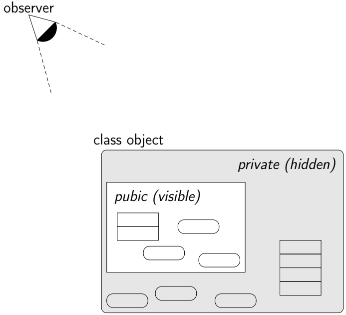
## Concept of an Object

### Members

- Attributes
  - Data members
  - Data type definitions
- Member functions

### Basic access controls

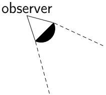
- Public
- Protected
- Private



# Data-directed design: Objects and Data abstraction

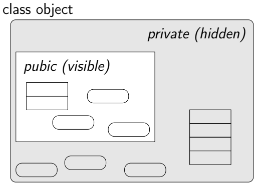
## Information hiding

- *Abstraction* for external use (logical view)
- Hiding internal detail (physical view)




## Data encapsulation

- Protect data from unintended modifications
- Access control




## Abstract data types (ADTs)

- Specification (external)
  - What information encapsulated, access control, operations provided
- Implementation (hidden)

What data encapsulation and implementation techniques have you seen in the lab exercise last week?

# Abstraction

image source: https://www.flickr.com/photos/gameoflight/26315058764/sizes/c/

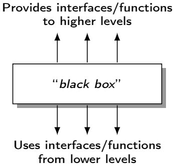

- The abstraction process decide in what level of details we need to *highlight* and what details can be *ignored* (Wing, 2008), i.e., what we can "*keep*" and what we can "*remove*" from the model

image source: http://bluebeanart.com/chinese-ink-painting-workshop

# Abstraction: Levels of Abstraction

- Each task/application may require different *levels of abstraction*
- It works like a black box, and helps us to *hide* unnecessary detail by giving things "**names**", allowing us to focus on the most *essential aspects* for the task at hand

What abstraction techniques have been used in the lab exercise last week?




- For example, when designing the mechanism and logics of an auto-driving system, the use of different types of sensors and actuators, logic and AI approaches, etc., and the way of how they interact with each others become important.
  - i.e., a lower level of abstraction is needed.

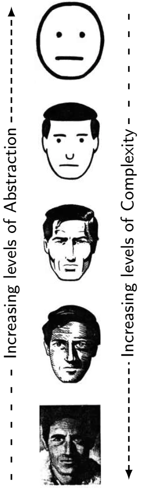



image adopted from: (Ramsden, 2018)

# Data Abstraction Example: Indexed Set

- A *set* of items of same type (e.g., `double`, `Complex`, etc.)
- Data Required:
  - data array (`double`)
- Functions provide:
  - Add an item to the set, the item is indexed
  - Retrieve an indexed element from the set
  - Get the size of the set
- Constraint
  - Limited capacity

# Data Abstraction Example: Indexed Set (cont.)

```
public class IndexSet {

  private static final int
        DEFAULT_MAX_SIZE = 20;
  private static final int
        INVALID_INDEX = -1;

  private int size;
  private int maxSize;

  private double[] items;

  public IndexSet() {}
  public IndexSet(int maxSize) {}

  /**
   * add an item to the set
   *
   * @param item to be added
   * @return index of item in the set
   */
  public int addItem(double item) {}

  /**
   * retrieve an indexed element in the
         set
   *
   * @param index of the item
   * @return value of the item
   */
  public double retrieve(int index) {}

  /**
   * get the size of the set
   *
   * @return size of the set
   */
  public int getSize() {}

}
```

# Data Abstraction Example: Indexed Set

## Implementation

### Constants

```
private static final int
      DEFAULT_MAX_SIZE = 20;
private static final int
      INVALID_INDEX = -1;
```

### Variables

```
private int maxSize;
private int size;

private double[] items;
```

### Constructors

```
public IndexSet(int maxSize) {
  this.maxSize = maxSize;
  this.items = new double[maxSize];
}

public IndexSet() {
  this(DEFAULT_MAX_SIZE);
}
```

### Getters/Setters

```
/**
 * get the size of the set
 *
 * @return size of the set
 */
public int getSize() {
  return size;
}
```

# Data Abstraction Example: Indexed Set

## Implementation

### Constants

```
private static final int
      DEFAULT_MAX_SIZE = 20;
private static final int
      INVALID_INDEX = -1;
```

### Variables

```
private int maxSize;
private int size;

private double[] items;
```

### Methods

```
/**
 * add an item to the set
 *
 * @param item to be added
 * @return index of item in the set
 */
public int addItem(double item) {
  if (size < maxSize) {
    items[size] = item;
    return size++;
  } else {
    System.err.println("Set already full!  Can't add more item!");
    return INVALID_INDEX;
  }
}
```

# Data Abstraction Example: Indexed Set

## Implementation

### Constants

```
private static final int
      DEFAULT_MAX_SIZE = 20;
private static final int
      INVALID_INDEX = -1;
```

### Variables

```
private int maxSize;
private int size;

private double[] items;
```

### Methods

```
/**
 * retrieve an indexed element in the set
 *
 * @param index of the item
 * @return value of the item
 */
public double retrieve(int index) {
  if (index < 0 || index >= size) throw new IllegalArgumentException("
       Index out of range: " + index);
  return items[index];
}
```

This complete the implementation of `IndexSet`!

# Data Abstraction Example: Complex Set

Now, suppose that we are going to update the `IndexSet` class to a new class `ComplexSet` to:

- Data Required:
  - data array (`Complex`)
- Functions provide:
  - Add a complex number to the set
  - Find whether a copy of a complex number is in the set
  - Get the size of the set.
- Constraints
  - *No* duplicated items
  - Variable size ⇐ how to handle this?

# Data Abstraction Example: Complex Set (cont.)

## Implementation

Replaced `DEFAULT_MAX_SIZE` with new constants

```java
public class ComplexSet {

  private static final int
        DEFAULT_INITIAL_CAPACITY = 2;
  private static final double
        GROWTH_FACTOR = 1.5;

  private static final int
        INVALID_INDEX = -1;

  int size;
  int maxSize;

  Complex[] items;

  public ComplexSet() {}
  public ComplexSet(int initialSize) {}

  /**
   * get the size of the set
   */
  public int getSize() {}

  /**
   * add an item to the set
   */
  public int addItem(Complex item) {}

  /**
   * Allocate more space for the set
   */
  private void growthSet() {}

  /**
   * check whether an identical copy of
        item is already in the set
   */
  public int isInSet(Complex item) {}

  /**
   * retrieve an indexed element in the
        set
   */
  public Complex retrieve(int index) {}

}
```

Changed data type to `Complex`

new methods added

# Data Abstraction Example: Complex Set (cont.)

## Implementation

### Constants

```
private static final int
      DEFAULT_INITIAL_CAPACITY = 2;
private static final double
      GROWTH_FACTOR = 1.5;

private static final int
      INVALID_INDEX = -1;
```

### Variables

```
private int maxSize;
private int size;

private Complex[] items;
```

### Constructors

```
public ComplexSet(int initialSize) {
  this.items = new Complex[initialSize];
  for (int i = 0; i < initialSize; i++) this.items[i] = null;
  size = 0;
  maxSize = initialSize;
}

public ComplexSet() {
  this(DEFAULT_INITIAL_CAPACITY);
}
```

### Getters/Setters

```
public int getSize() {
  return size;
}
```

# Data Abstraction Example: Complex Set (cont.)

## Implementation

### Constants

```
private static final int
     DEFAULT_INITIAL_CAPACITY = 2;
private static final double
     GROWTH_FACTOR = 1.5;

private static final int
     INVALID_INDEX = -1;
```

### Variables

```
private int maxSize;
private int size;

private Complex[] items;
```

### Methods

```
/**
 * retrieve an indexed element in the set
 *
 * @param index of the element
 * @return value of the element
 */
public Complex retrieve(int index) {
  if (index < 0 || index >= size) throw new IllegalArgumentException("
       Index out of range: " + index);
  return items[index];
}
```

# Data Abstraction Example: Complex Set (cont.)

## Implementation

### Constants

```
private static final int
      DEFAULT_INITIAL_CAPACITY = 2;
private static final double
      GROWTH_FACTOR = 1.5;

private static final int
      INVALID_INDEX = -1;
```

### Variables

```
private int maxSize;
private int size;

private Complex[] items;
```

### Methods

```
/**
 * check whether an identical copy of item is already in the set
 *
 * @param item to be checked
 * @return index of the item if it is already in the set;
 *              ItemSet.INVALID_INDEX otherwise
 */
public int isInSet(Complex item) {
  if (null == item) return INVALID_INDEX;
  for (int i = 0; i < size; i++) {
    if (items[i].equals(item)) return i;
  }
  return INVALID_INDEX;
}
```

# Data Abstraction Example: Complex Set (cont.)

## Implementation

### Constants

```
private static final int
      DEFAULT_INITIAL_CAPACITY = 2;
private static final double
      GROWTH_FACTOR = 1.5;

private static final int
      INVALID_INDEX = -1;
```

### Variables

```
private int maxSize;
private int size;

private Complex[] items;
```

### Methods

```
private void growthSet() {
  int newMaxSize = (int) (GROWTH_FACTOR * maxSize);
  System.out.println(getClass().getSimpleName() + ": new set size=" +
        newMaxSize);
  // create an array with new max size
  Complex[] newItems = new Complex[newMaxSize];
  // copy the items to the new array
  for (int i = 0; i < size; i++) {
    newItems[i] = items[i];
  }
  // initialize the rest of the array to null
  for (int i = size; i < newMaxSize; i++) {
    newItems[i] = null;
  }
  // replace the old array with the new one
  // and update the new max size
  items = newItems;
  maxSize = newMaxSize;
}
```

We can:

(i) Create an array with larger size

(ii) Copy the contents in the old array to the new one

(iii) Set the old array to the new array

# Data Abstraction Example: Complex Set (cont.)

## Implementation

### Constants

```
private static final int
      DEFAULT_INITIAL_CAPACITY = 2;
private static final double
      GROWTH_FACTOR = 1.5;

private static final int
      INVALID_INDEX = -1;
```

### Variables

```
private int maxSize;
private int size;

private Complex[] items;
```

### Methods

```
/**
 * add an item to the set
 *
 * @param item to be added
 * @return index of item in the set
 */
public int addItem(Complex item) {
  if (null == item) return INVALID_INDEX;
  // check whether the item is in the set
  int index = isInSet(item);
  if (index >= 0) return index;

  // check whether there is any free space in the set
  // and increase the size of the set if not
  if (size == maxSize) growthSet();

  items[size] = item;
  return size++;
}
```

This complete the implementation of `ComplexSet`!

Question: what is the cost of adding an item to the `ComplexSet`?

In the examples above, we showed:

- How ADT can be used to store the information and abstract away the implementation details from users
- In both `IndexSet` and `ComplexSet` classes
  - items are stored inside an array (internal), and
  - the same set of methods, a.k.a. *interface*, are provided (external)
    - `getSize()`, `retrieve()`, `addItem()`
- However
  - In `IndexSet` class
    - the size of the set is fixed, and
    - duplication of data is allowed
  - In `ComplexSet` class
    - the size of the set is dynamic, and
    - duplication of data is *not* allowed

## Reading

- Chapter 3, Cormen (2022)

# References I


Ramsden, Dan (Mar. 2018). *Shapes and ladders — the art of abstraction and meaning making*. Online: https://danramsden.medium.com/shapes-and-ladders-the-art-of-abstraction-and-meaning-making-36208eec2098. last accessed: 13 Mar 2024.

Wing, Jeannette M. (2008). “Computational thinking and thinking about computing”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Science* 366, pp. 3717–3725.