/ JAVA DB SCHOOL
Arrays, Classes and Objects

/ Arrays

Arrays

- Stores a collection of elements of the same type; the type is specified when declaring the array

- Unidimensional arrays:

```
double[] myDoubles = new double[20];
```

- A continuous space for holding all the elements (20 in this case) of that type (double in this case) is reserved in memory

Arrays

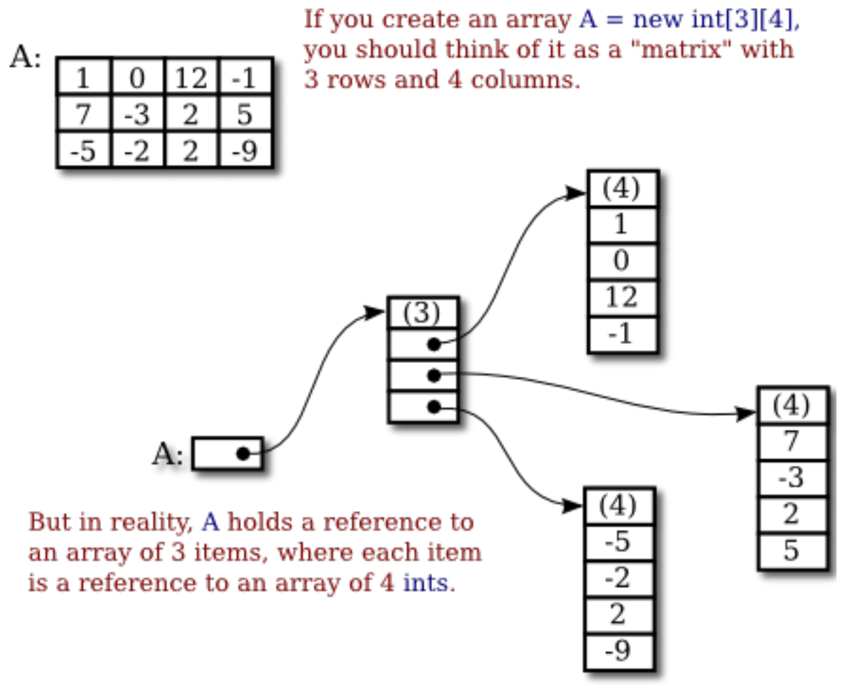
- An element can be modified by its index (0-based):

```
myDoubles[0] = 3.14;
myDoubles[4] = 6.28;
```

- The array is a “special object” in Java with a few properties https://docs.oracle.com/javase/specs/jls/se7/html/jls-10.html#jls-10.7

- `myDoubles.length` – contains the number of components of the array. length may be positive or zero

Arrays


A:
1 0 12 -1
7 -3 2 5
-5 -2 2 -9
If you create an array A = new int[3][4], you should think of it as a "matrix" with 3 rows and 4 columns.
(3)
(4)
1
0
12
-1
(4)
7
-3
2
5
(4)
-5
-2
2
-9
A:
But in reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

http://math.hws.edu/javanotes/c7/s5.html

- Bidimensional arrays:
  `int A = new int[3][4];`
- Each element of the array is a unidimensional array
- Each unidimensional array can be stored in a different memory area, while the bidimensional array stores references to those areas

/ Classes and Objects

Classes and Objects

- Computer programming model that organizes software design around data, or objects, rather than functions and logic.

- **Class** = template for an object

- **Object** = entity of a class
  - Instantiated from a class
  - Characterized by *identity*, *state* and *behaviour*
  - Identity = reference to their memory location
  - State = fields or properties
  - Behaviours = defined using methods

Classes and Objects

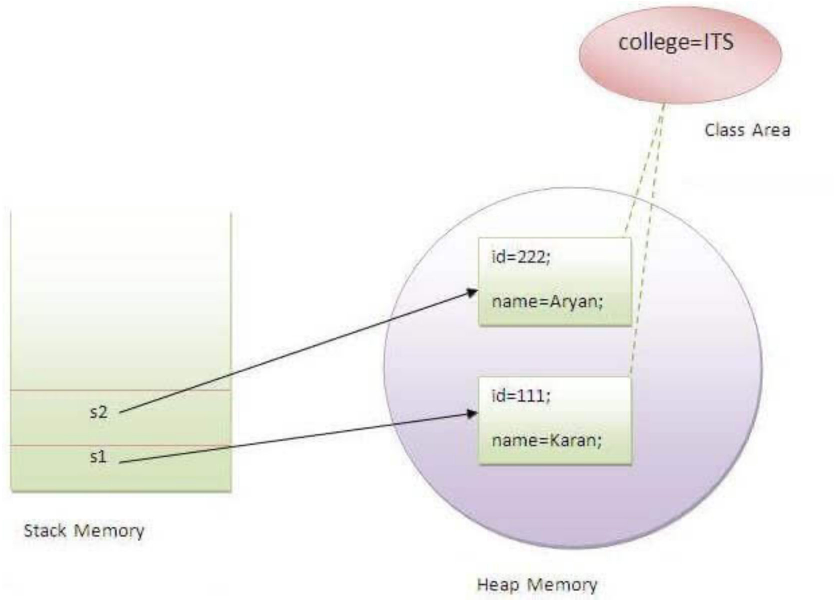
- Objects are instantiated from a class using the new operator

```
String s = new String();
```

- Objects represent a reference (address) in memory

- Unused objects are cleared by the garbage collector

Static Properties and Methods



- Belongs to the class (not to each object) and are initialized only once
- Are initialized first, before the initialization of any instance properties
- Referred using class name

Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

/ OOP Properties

Inheritance

- Used to derive new classes from existing ones
- A general class is needed, and a more specific one extends the general one
- A class can extend only one class
- Relationships and subclasses between objects are defined with the help of inheritance, allowing developers to reuse a common logic while maintaining a unique hierarchy

Encapsulation

- The implementation and the state of each object are privately held inside a defined boundary (class). Other objects do not have access to this class, but they are only able to call their public methods

- This allows data hiding, provides program security and avoids unintended data modifications

Abstraction

- Depend on abstractions, not on concretions
- Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code
- Very helpful to easily adapt the code over time when changing concrete implementations

Polymorphism

- The ability of an object to take on many forms
- The most common use occurs when a parent class reference is used to refer to a child class object

Overloading

- Reusing the same name for methods multiple times, in a same class
- The signature of the method is used to differentiate between them
- The signature refers to: the name of the method, the number of parameters, the type of parameters, the return type
- Two overloaded methods must differ by either the number of parameters, or by the type of them if two overloads have the same number of parameters

Overriding

- Allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes
- The access modifier for an overriding method can allow more, but not less, access than the overridden method
- Final methods cannot be overridden
- Static methods can not be overridden
- Private methods cannot be overridden

Casting

- Upcasting (Generalization or Widening) – casting to a parent class (casting a specific type to a more general type)
- Downcasting (specialization or narrowing) – casting to a child type (casting a more general type to a more specific one)

Casting

- Used to test whether the object is an instance of the specified type (class or subclass or interface)
- Used mostly with abstraction: an object is defined as a more general type, while a concrete implementation is bind to it

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);//true
 }
}
```

Abstract Classes

- Allows to define methods without implementing them
- Classes which extend abstract classes will implement them

```
public abstract class GraphicObject {
   // declare fields
   // declare nonabstract methods
   abstract void draw();
}
```

Interfaces

- Allow using polymorphism

- Contains methods without implementations
    - Starting with Java 8, default and static methods may have implementation in the interface definition
    - In Java 9, private and private static methods were added

- A class *implements* multiple classes (allows multiple inheritance)
    - Remember that a class *extends* only one class

Cloneable, Comparable Interfaces

- Cloneable and Comparable interfaces (defined in java.lang package)

- Classes that implement Cloneable, must define *clone* method
  - Should create a copy of the provided object
  - https://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html

- Classes that implement Comparable, must define *compareTo* method
  - Defines the “order” of the first object to the second one
  - https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

/ Practice, practice, practice

Matrix Sum and Product

Read from standard input (keyboard) an integral value n (n <= 10) and then two matrices of size n x n. Compute and print the sum matrix and the product matrix between the two.

Input sample:

```
n = 3;
a = 4 1 2
    3 4 6
    2 7 5
b = 9 1 2
    3 4 5
    7 1 2
```

Output sample

```
sum  = 13 2 4
       6 8 11
       9 8 7
prod = 53 10 17
       81 25 38
       74 35 49
```

At the Doctor’s

Define a *Doctor* class containing a *name* property. Overwrite the *toString* method in order to print a human-readable representation of an object (showing the name). Requirements:

- Doctor has an empty constructor and another constructor that sets the name
- There are two classes that extend Doctor: Surgeon, Generalist
- Surgeon class has the *expertise* property (String)
- Generalist class has the *noPatients* property (int)
- Both subclasses have empty constructors and constructors that set their specific property
- Both subclasses must rewrite *toString* method to display those fields

At the Doctor’s (cont.)

Requirements (continued):

- Define an array of Doctor and populate it with Surgeon and Generalist objects
- Iterate through the array and print all the doctors
- Add one method to each specific doctor class: intervention() for Surgeon, and writeRecipe() for Generalist
- Using instanceOf, perform the specific behaviour depending on doctor’s type
- Check if there are at least two doctors with the same expertise
- Order all generalist doctors by the number of their patients (optional)
- Find out how many generalist doctors have a larger number of patients than the medium number of all generalists

Vehicles

Create a Vehicle class [that implements Comparable interface].

Requirements:

- Vehicle has the following properties: colour (String) and functional (boolean)
- An empty constructor that sets default values, and one with parameters
- Getters and setters for the field (encapsulate the properties)
- Three abstract methods: *charge*, *profit* and *display* (optional)
- Implement the *compareTo* method that compares two vehicles by their profit (optional)

Vehicles (optional, cont.)

Requirements (continued):

- Create two classes that extend Vehicle class: Bus and Taxi, with two new fields: noPassengers, ticketPrice.
- Define constructors for the two classes
- Calculate *profit* as 25% of the price for each *charge*.
- Create a BusStation class that holds an array of Vehicles. Create three methods: *showAllVehicles*, *showTotalProfit*, *[sortVehicles]*.
- Test the methods in a main class by populating the array with various vehicles.

/ Q&A

MOBILE/ACADEMY