

# DATA STRUCTURE AND ALGORITHMS

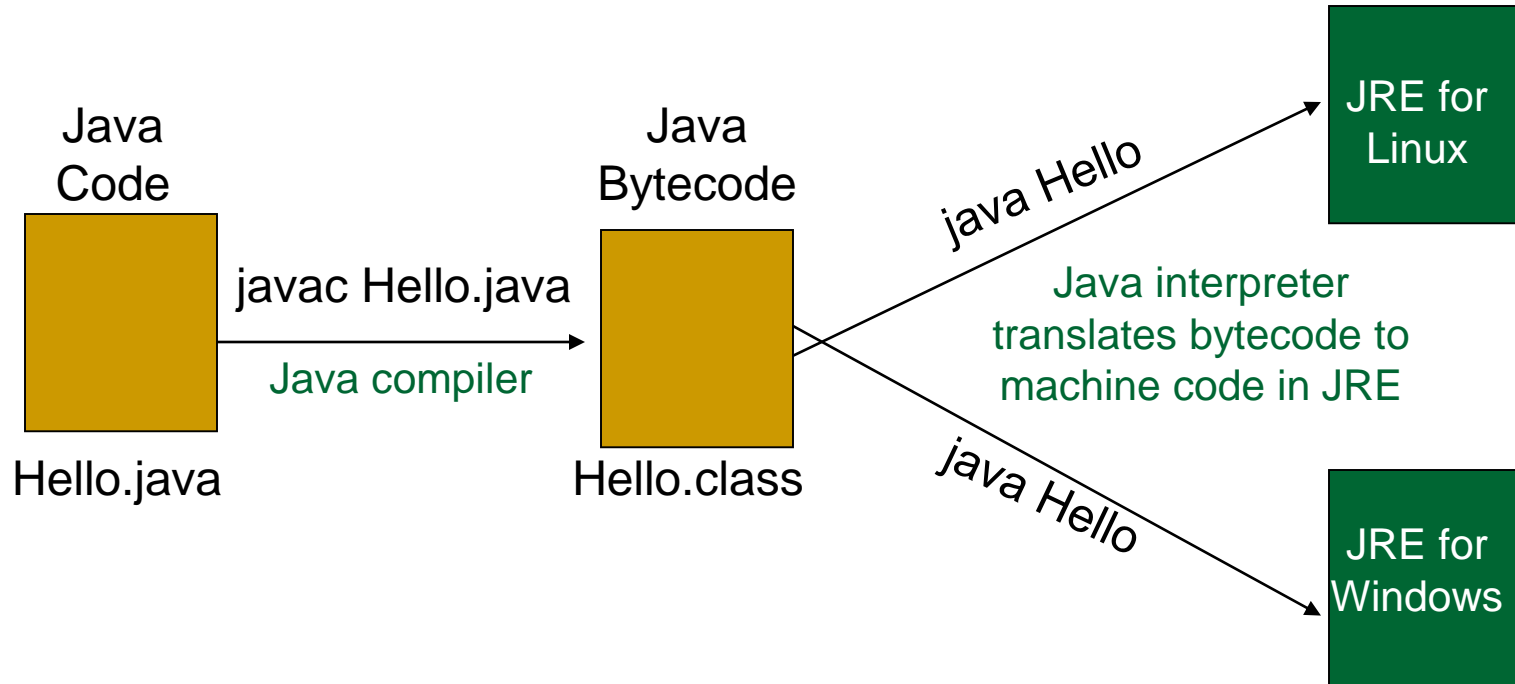
## LECTURE 1

Java review

# Important Java Concepts and Terminology

- JDK (formerly SDK) is the Java Development Kit.  
JDK = JRE + development tools
- JRE (Java Runtime Environment): creates a virtual machine; is specific to your platform and is the environment in which Java byte code is run.
- To learn more about JDK, JRE, etc., visit <http://www.oracle.com/technetwork/java/javase/tech/>

# Compiling and Running Java



JRE contains class libraries which are loaded at runtime.

---

# Important Java Concepts

- Everything in Java must be inside a class.
- Every file may only contain one public class.
- The name of the file must be the name of the class appended to the java extension.
- Thus, *Hello.java* must contain one public class named *Hello*.

# Methods in Java

The *main* method has a specific signature.

- Example: “Hello world!” Program in Java

```
public class Hello
{
    public static void main(String args[])
    {
        System.out.println("Hello world!");
    }
}
```

# Methods in Java (cont.)

- All methods must be defined inside a class.
- Format for defining a method:

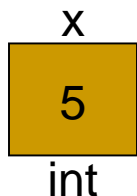
```
[modifiers] return_type method_name([param_type param] *)  
{  
    statements;  
}
```

- For *main*:
  - ❑ Modifiers: *public static*
  - ❑ Return type: *void*
  - ❑ Parameter: an array of type String, *String []* is command line arguments.

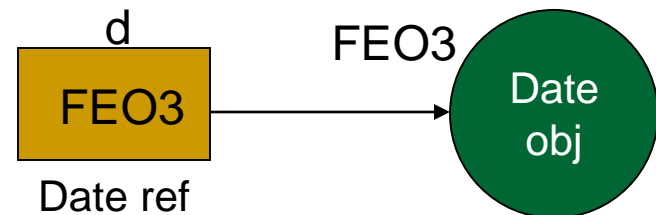
# Data Types

- Two types of data types in Java: primitives; references.
- Primitives are data types that store data.
- References store the address of an object, which is encapsulated data.

```
int x = 5;
```

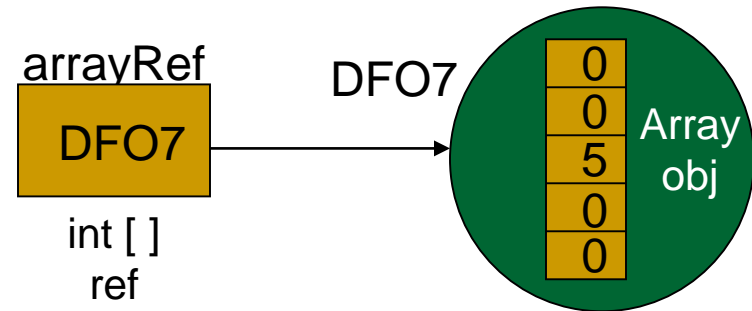


```
Date d = new Date();
```



# Arrays

```
int [] arrayRef;  
arrayRef = new int[5];  
arrayRef[2] = 5;
```



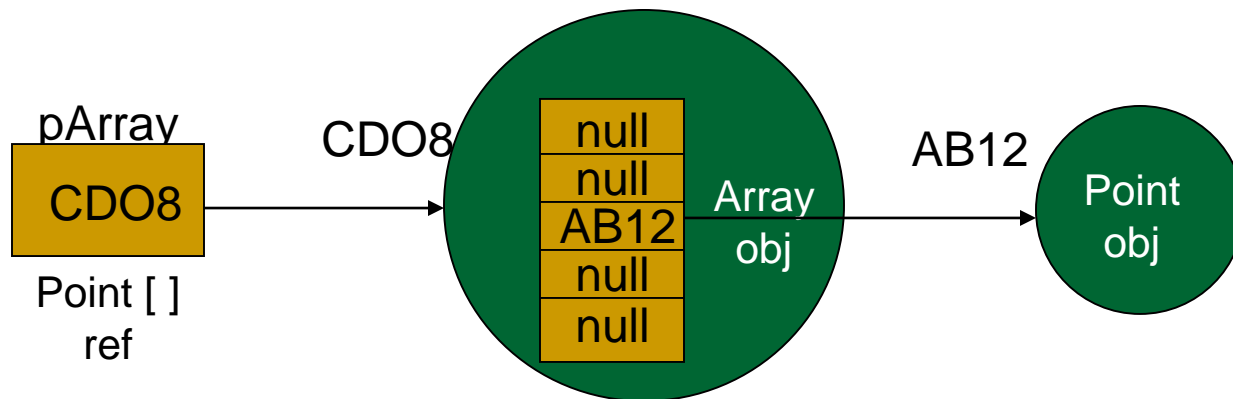
- Arrays in Java are objects. The first line of code creates a reference for an array object.
- The second line creates the array object.
- All arrays have a length property that gives you the number of elements in the array.
  - *args.length* is determined at runtime



## Arrays (cont.)

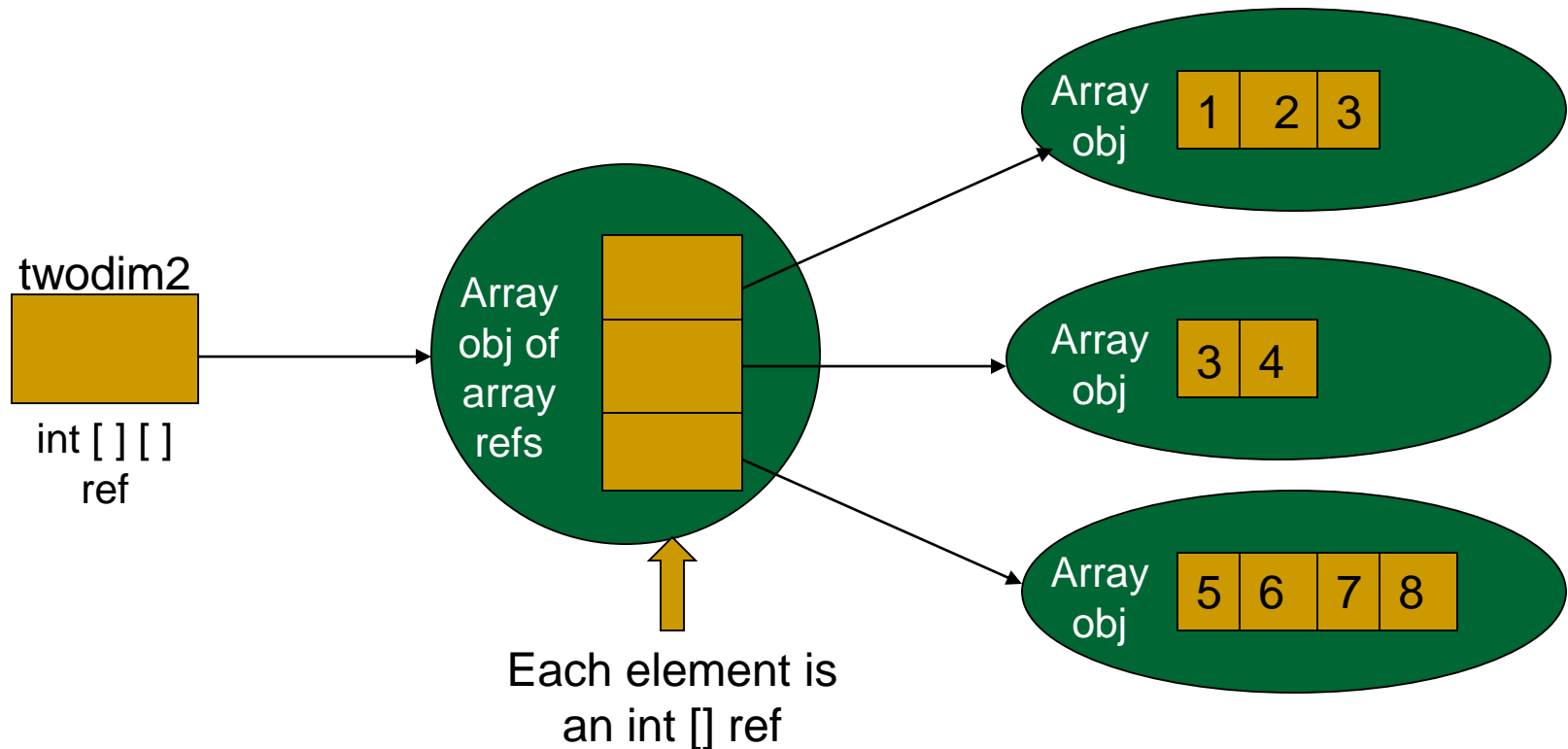
- An array of objects is an array of object references until the objects are initialized.

```
Point pArray [] = new Point[5];  
pArray[2] = new Point();
```



# Multidimensional Arrays

- A pictorial rendition of twodim2.



# Java Naming Conventions

- **Classes and Interfaces**

`StringBuffer, Integer, MyDate`

- **Identifiers for methods, fields, and variables**

`_name, getName, setName, isName, birthDate`

- **Packages**

`java.lang, java.util, proj1`

- **Constants**

`PI, MAX_NUMBER`

- **Coding Style for Java**

<https://google.github.io/styleguide/javaguide.html>

# Comments

- Java supports three types of comments.

- C style            `/* multi-liner comments */`

- C++ style    `// one liner comments`











- Javadoc

- `/**`

- This is an example of a javadoc comment. These comments can be converted to part of the pages you see in the API.

- `*/`

# Access Control

<b><i>Modifier</i></b>	<b><i>Same class</i></b>	<b><i>Same package</i></b>	<b><i>Subclass</i></b>	<b><i>Universe</i></b>
private				
default				
protected				
public				

# Classes

- In Java, all classes at some point in their inheritance hierarchy are subclasses of `java.lang.Object`, therefore all objects have some inherited, default implementation before you begin to code them.
  - `String toString()`
  - `boolean equals(Object o)`

# Inheritance in Java

- Inheritance is implemented using the keyword `extends`.

```
public class Employee extends Person
{
    //Class definition goes here - only the
    //implementation for the specialized behavior
}
```

- A class may only inherit from only one superclass.
- If a class is not derived from a super class then it is derived from *java.lang.Object*. The following two class declarations are equivalent:

```
public class Person {...}
public class Person extends Object {...}
```

# Polymorphism

- If Employee is a class that extends Person, an Employee “is-a” Person and polymorphism can occur.



Creates an array of  
Person references

```
Person [] p = new Person[2];  
p[0] = new Employee();  
p[1] = new Person();
```



## Polymorphism (cont.)

- However, a Person is not necessarily an Employee. The following will generate a compile-time error.

```
Employee e = new Person();
```

- Polymorphism requires general class on left of assignment operator, and specialized class on right.
- Casting allows you to make such an assignment provided you are confident that it is ok.

```
public void convertToPerson(Object obj)
{
    Person p = (Person) obj;
}
```

# Abstract Classes and Methods

- Java also has abstract classes and methods. If a class has an abstract method, then it must be declared abstract.

```
public abstract class Node{
```

```
    String name;
```

```
    public abstract void type();
```

```
    public String toString(){ return name;}
```

```
    public Node(String name){
```

```
        this.name = name;
```

```
    }
```

```
}
```



Abstract methods have no implementation.

# More about Abstract Classes

- **Abstract classes can not be instantiated.**

```
// OK because n is only a reference.
```

```
Node n;
```

```
// OK because NumberNode is concrete.
```

```
Node n = new NumberNode("Penta", 5);
```

```
// Not OK. Gives compile error.
```

```
Node n = new Node("Name");
```

# Interfaces

- An *interface* is like class without the implementation. It contains only
  - public, static and final fields, and
  - public and abstract method headers (no body).
- A public interface, like a public class, must be in a file of the same name.

# Interface Example

- The methods and fields are implicitly public and abstract by virtue of being declared in an interface.

```
public interface Employable
{
    void raiseSalary(double d);
    double getSalary();
}
```

## Interfaces (cont.)

- Many classes may implement the same interface. The classes may be in completely different inheritance hierarchies.
- A class may implement several interfaces.

```
public class TA extends Student
implements Employable
{
    /* Now TA class must implement the getSalary
       and the raiseSalary methods here */
}
```

---

# The Collections Framework

- Is a collection of interfaces, abstract and concrete classes that provide generic implementation for many of the data structures.
- It has:
  - Interfaces and its implementations, i.e., classes
  - Algorithm

# Generics

- Since JDK 1.5 (Java 5), the Collections framework has been parameterized.
- A class that is defined with a parameter for a type is called a generic or a parameterized class.



# Collection <E> Interface

- The E represents a type and allows the user to create a homogeneous collection of objects.
- Using the parameterized collection or type, allows the user to retrieve objects from the collection without having to cast them.

## Before:

```
List c = new ArrayList();  
c.add(new Integer(10));  
Integer i = (Integer) c.get(0);
```

## After:

```
List<Integer> c = new ArrayList<Integer>();  
c.add(new Integer(10));  
Integer i = c.get(0);
```

---

# Implementing Generic Classes

- In the projects for this course, you will be implementing your own parameterized generic classes.
- The Cell class that follows is a small example of such a class.

# Generic Cell Example

```
*CellGenericExample.java ✕  
  
package generics;  
  
public class CellGenericExample{  
    public class Cell< T >{  
        private T prisoner;  
        public Cell( T p)  
            { prisoner = p; }  
        public T getPrisoner(){return prisoner; }  
    }  
  
    public static void main (String[ ] args){  
        // define a cell for Integers  
        Cell<Integer> intCell = new Cell<Integer>( new Integer(5) );  
  
        // define a cell for Floats  
        Cell<Float> floatCell = new Cell<Float>( new Float(6.7) );  
  
        // compiler error if we remove a Float from Integer Cell  
        Float t = (Float)intCell.getPrisoner( );  
        System.out.println(t);  
    }  
}
```

Cannot cast from Integer to Float

# Dont's of Generic Programming

- You CANNOT use a type parameter in a constructor.

~~T obj = new T();~~

- You CANNOT create an array of a generic type.

~~T [] array = new T[5];~~

# Do's of Generic Programming

- The type parameter must always represent a reference data type.
- Class name in a parameterized class definition has a type parameter attached.

```
class Cell<T>
```

- The type parameter is not used in the header of the constructor.

```
public Cell( )
```

- Angular brackets are not used if the type parameter is the type for a parameter of the constructor.

```
public Cell3(T prisoner );
```

- However, when a generic class is instantiated, the angular brackets are used

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

# The Arrays class

- The `java.util.Arrays` class is a utility class that contains several static methods to process arrays of primitive and reference data.
  - ❑ *binarySearch* – searches sorted array for a specific value
  - ❑ *equals* – compares two arrays to see if they contain the same elements in the same order
  - ❑ *fill* – fills an array with a specific value
  - ❑ *sort* – sorts an array or specific range in array in ascending order according to the natural ordering of elements

# Natural Order

- The natural order of primitive data types is known. However, if you create an `ArrayList` or `Array` of some object type, how does the *sort* method know how to sort the array?
- To be sorted, the objects in an array must be comparable to each other.

# The Comparable<T> Interface

- The Comparable<T> interface defines just one method to define the natural order of objects of type T

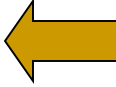
```
public interface java.lang.Comparable<T>
{
    int compareTo(T obj);
}
```

- *compareTo* returns
  - ❑ a negative number if the calling object precedes *obj*
  - ❑ a zero if they are equal, and
  - ❑ a positive number if *obj* precedes the calling object



# Comparable Example

```
import java.util.*;
public class Fraction implements Comparable<Fraction>
{
    private int n;
    private int d;
    public Fraction(int n, int d){ this.n = n; this.d = d;}
    public int compareTo(Fraction f)
    {
        double d1 = (double) n/d;
        double d2 = (double) f.n/f.d;
        if (d1 == d2)
            return 0;
        else if (d1 < d2)
            return -1;
        return 1;
    }
    public String toString() { return n + "/" + d; }
}
```

 Casting required for floating point division

# Sort Example

```
public class FractionTest
{
    public static void main(String []args)
    {
        Fraction [] array = {new Fraction(2,3),
                               new Fraction (4,5), new Fraction(1,6);
        Arrays.sort(array);
        for(Fraction f :array)
            System.out.println(f);
    }
}
```

# Generic Sorting

```
public class Sort
{
    public static <T extends Comparable<T>>
    void bubbleSort(T[] a)
    {
        for (int i = 0; i < a.length - 1; i++)
            for (int j = 0; j < a.length - 1 - i; j++)
                if (a[j+1].compareTo(a[j]) < 0)
                {
                    T tmp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = tmp;
                }
    }
}
```

# Generic Sorting (cont.)

- Given the following:

```
class Animal implements Comparable<Animal> { ...}  
class Dog extends Animal { ... }  
class Cat extends Animal { ... }
```

- Now we should be able to sort dogs if contains the *compareTo* method which compares animals by weight.
- BUT... bubblesort only sorts objects of type T which implements Comparable<T>. Here the super class implements Comparable.... HENCE, we can't use bubblesort for Cats or Dogs
- New and improved sort on next page can handle sorting Dogs and Cats.

# Generic Sorting (cont.)

```
public class Sort
{
    public static <T extends Comparable<? super T>>
    void bubbleSort(T[] a)
    {
        for (int i = 0; i < a.length - 1; i++)
            for (int j = 0; j < a.length - 1 - i; j++)
                if (a[j+1].compareTo(a[j]) < 0)
                {
                    T tmp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = tmp;
                }
    }
}
```

# Exercises

- Nhập và in các giá trị kiểu dữ liệu cơ bản
- Thao tác với mảng 1 chiều
- Thao tác với phân số
- Thao tác với hình cầu (sphere)
- (\*) Project sách M.Goodrich trang 57



Homework 1: Java Review