Abstract Classes and Interfaces

Colorado State University

Department of Computer Science

Slides Originally Created by Albert Lionelle (Albert.Lionelle@colostate.edu),
updated by Marcia Moraes (marcia.moraes@colostate.edu)

Announcements

TODO Reminders:

Readings are due **before** lecture

- Reading 23 (zybooks) – you should have already done that ☺
- Lab 15
- Reading 24 (zyBooks) – you should have already done that ☺
- Practical Project Lecture
- RPA 11

Keep practicing your RPAs in a spaced and mixed manner ☺

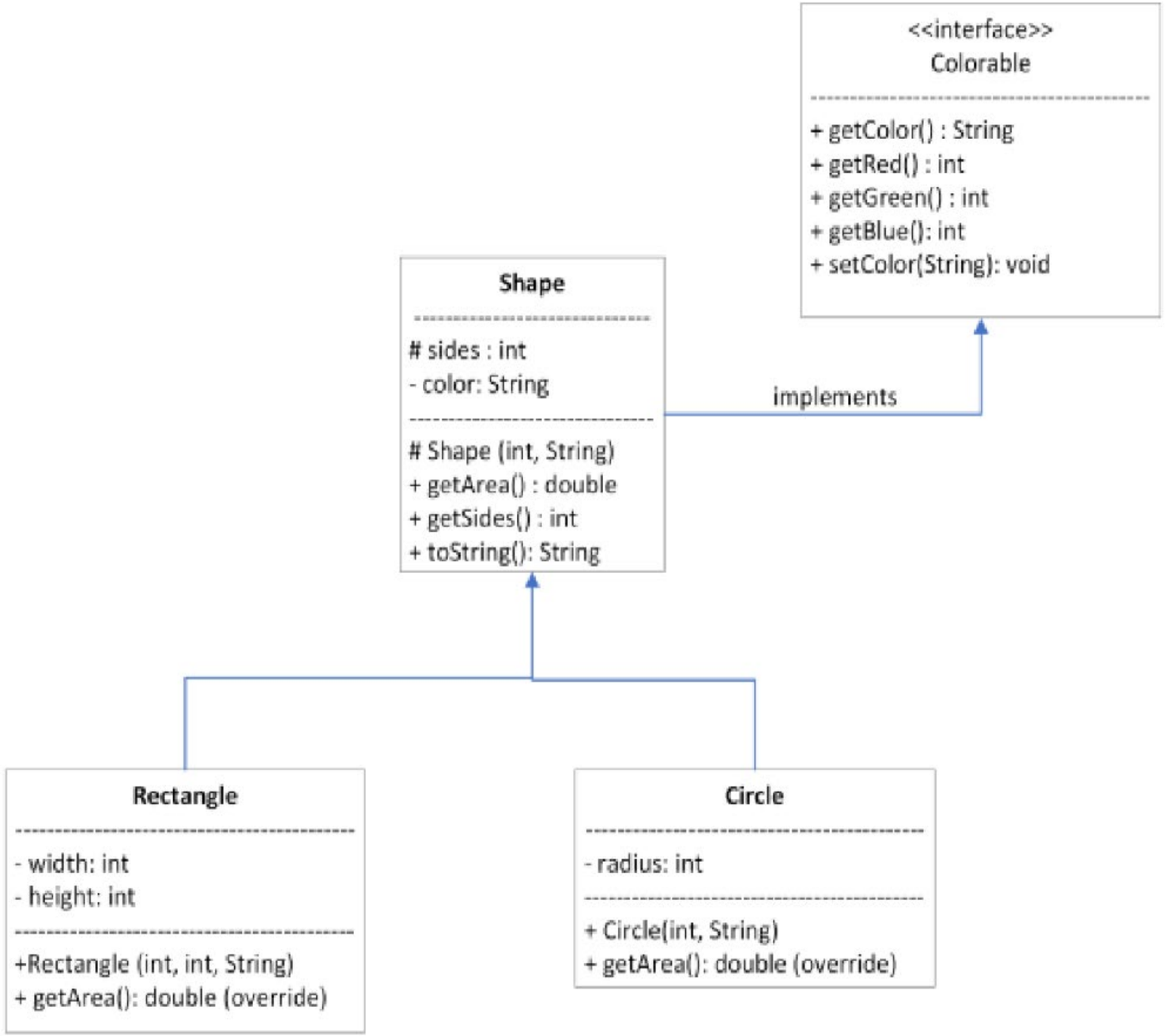
NEXT WEEK – Exam 3 Week
Catch up, if you need!

Help Desk

Day	Time : Room
Monday	12 PM - 2 PM : CSB 120
Tuesday	6 PM - 8 PM : Teams
Wednesday	3 PM - 5 PM : CSB 120
Thursday	6 PM - 8 PM : Teams
Friday	3 PM - 5 PM : CSB 120
Saturday	12 PM - 4 PM : Teams
Sunday	12 PM - 4 PM : Teams

Recall Activity

- Define the following elements:
- Interface
- Methods that are overwritten
- Superclasses and Subclasses
- Does getArea() really make sense for Shape?



Inheritance

- Creates an is-a relationship between classes
- Allows fully implemented 'more generalized' classes as the super classes
  - specialized subclasses as the subclasses inherits instance variables and methods from the super class
- Uses the key word extends
  - can only extend / inherited from one immediate parent (but can have 'chain' of parents)

```
Circle crcl = new Circle(10, "234,255,123");

System.out.println("The color is " + crcl.getColor());
```

Polymorphism

- Allows the subclass to be declared as the super
  - actually a subclass can 'substitute' in for the super

- Extremely useful for things like Arrays and ArrayLists

```
Shape[] shapes = new Shape[3]; // fixed size
shapes[0] = crcl;
shapes[1] = new Rectangle(23, 5, "123,125,255");

System.out.println(Arrays.toString(shapes));
```

```
[Sides: 1, Area: 314.16, Sides: 4, Area: 115.00, null]
```

Abstract Class

- What if a method you write, needs specific information?
  - Unique to subtypes / children
  - BUT – the rest of the method is general
- Enter Abstract classes
  - Classes that are not *complete* by themselves
  - Contain **partial** implementations of a class
    - With other methods that are *required* to be completed by children.
- Class is abstract, some methods are abstract
- Can’t be instantiated
  - but can have constructors the children can inherit

Abstract Example

```java
public class Paladin extends AbstractJob {
    private double modifier = 1.6;
     @Override
     public double getJobModifier() {
         return modifier;
     }

     public Paladin(String name) {
           super(name);
     }
}
```

Child

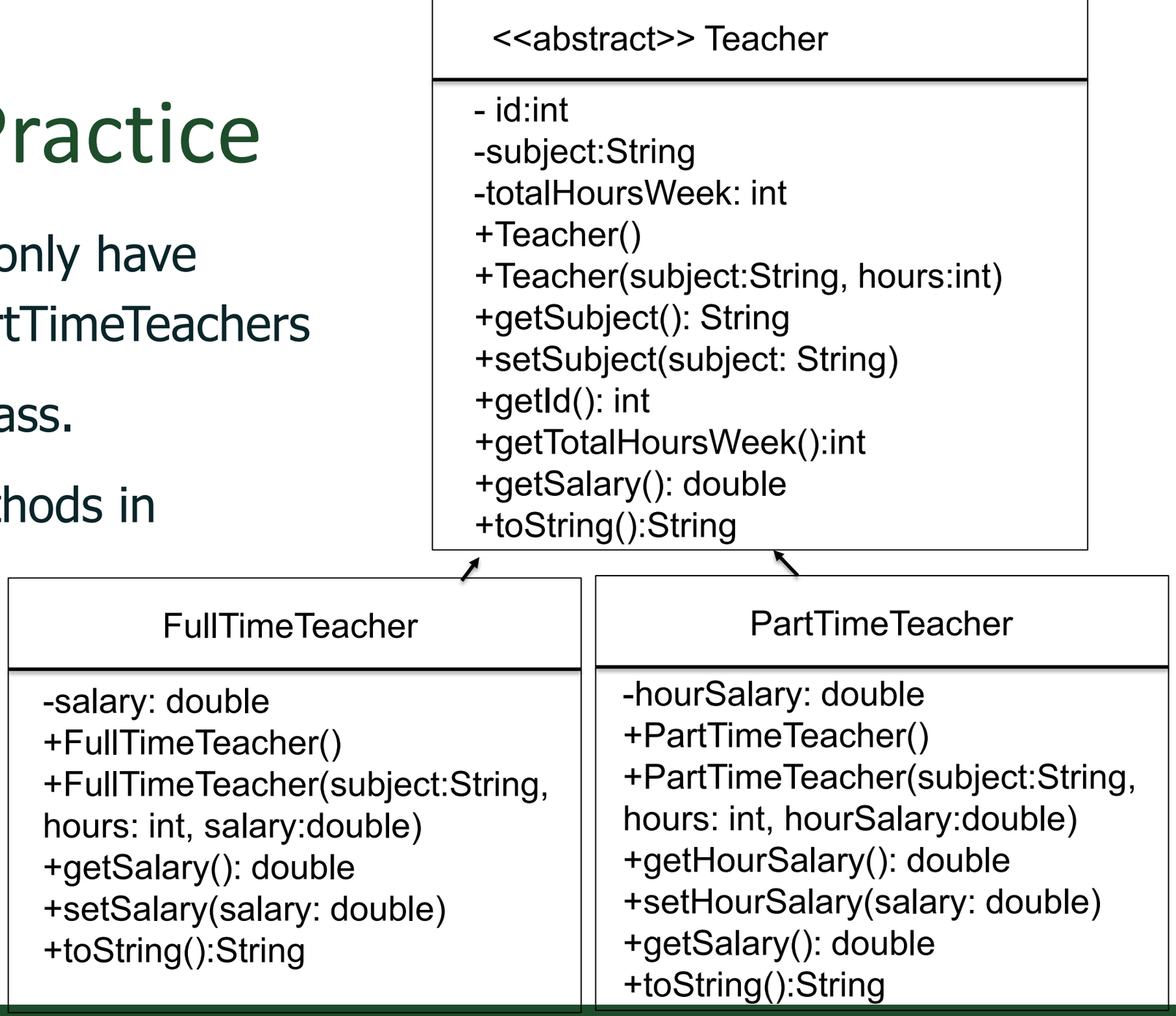
```java
public static void main(String[] args) {
     Paladin cecil = new Paladin("Cecil");
     System.out.println(cecil.getArmor());
}
```

```java
public abstract class AbstractJob {
     protected String name;
     protected ArrayList<String> inventory;

     private int armor;
     private int attack;

     public abstract double getJobModifier();
     // unique to sub classes


     public double getArmor() {
          return armor * getJobModifier();
     }

     public AbstractJob(String name) {
        this.name = name;
        this.armor = 1;
        this.attack = 1;
        inventory = new ArrayList<>();
     }
}
```

Parent

Abstract Class Practice

- Considering that we can only have FullTimeTeachers and PartTimeTeachers
- Teacher as an abstract class.
- Is there any abstract methods in Teacher?



Interfaces

- Inheritance Limitation: Can only inherit **one** class directly
  - meaning, there can be a chain of classes
- What if we wanted to 'inherited' from more than one class?
- Enter interfaces
  - **contracts** that define what methods will be implemented
  - contains no implementation – just definitions
  - uses **implements** in the class to say class is following the contract
- Common Interface
  - Comparable (specification)
    - implementing it – allows objects to be sorted in ArrayList!
    - compareTo is the method

Interface example

```
public class Paladin extends AbstractJob implements MeleeType, HealerType {
    private double modifier = 1.6;
    @Override
    double getJobModifier() {
        return modifier;
    }
    @Override
    public double getCureModifier() {
        return 1.5;
    }
    @Override
    public double getMP() {
        return 10.5;
    }
    @Override
    public boolean hasSwordAttack() { return true;  }
    @Override
    public int calcSwordDamage() {
        return (int)(100*getJobModifier());
    }
}
```

```
public interface HealerType {
    double getCureModifier();
    double getMP();
}
```

```
public interface MeleeType {
    boolean hasSwordAttack();
    int calcSwordDamage();
}
```

Interface Example in Java

- Comparable
  - requires you implement .compareTo(T obj)
  - T is your object
  - Allows you to sort based on your own ideas!

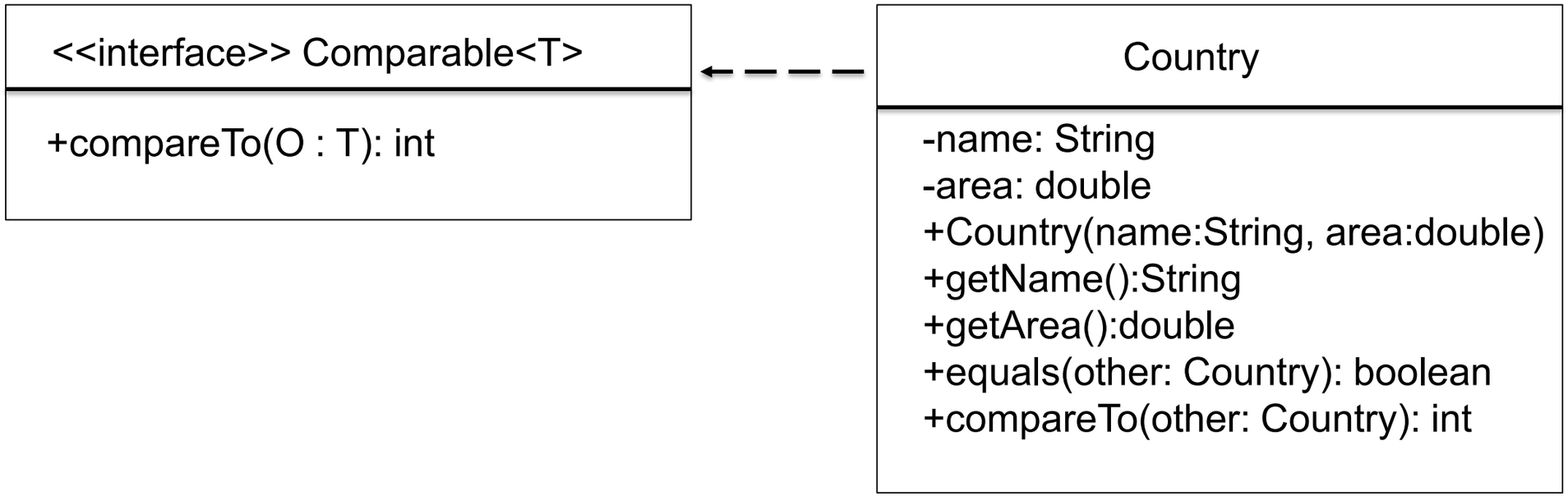
```
public static void main(String[] args) {
    ArrayList<Paladin> list = new ArrayList<>();
    list.add(new Paladin("Cecil"));
    list.add(new Paladin("Caliban"));
    Collections.sort(list); // puts Caliban before Cecil
    System.out.println(list); // Caliban, Cecil
}
```

```
public class Paladin extends AbstractJob implements Comparable<Paladin> {
   private double modifier = 1.6;
    @Override
    public double getJobModifier() {
        return modifier;
    }
    @Override
    public int compareTo(Paladin obj2) {
      return name.compareTo(obj2.name);
    }

    public Paladin(String name) {   super(name);    }
}
```

-1 if less than
0 if equal
1 if greater

Interface Practice

- Implement a list of Countries that can be ordered by their area.
- Need to implement interface Comparable and the method compareTo the areas.



Download Abstract class and Interface – classes for this lecture