CS 131 - Week 5

TA : Tanmay Sardesai

How to find these slides

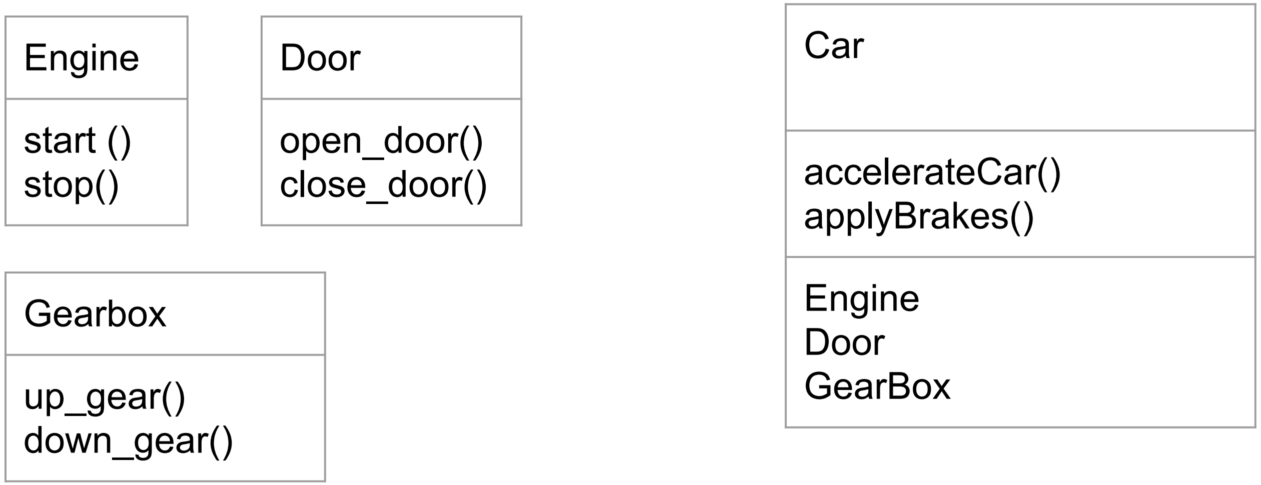
Piazza -> CS 131 -> Resources -> Discussion 1B

Announcements

- Email tanmays@cs.ucla.edu
- Office Hours Tuesday 9:00 am - 11:00 am. Bolter Hall 3256S-A
- HW3 will be due Tuesday 2/12 11:55 pm
- HW4 will be due Friday 2/22 11:55 pm

Aggregation and Decomposition

- Make new classes by reusing existing. Reuse reduces effort! Think Has-A relationship



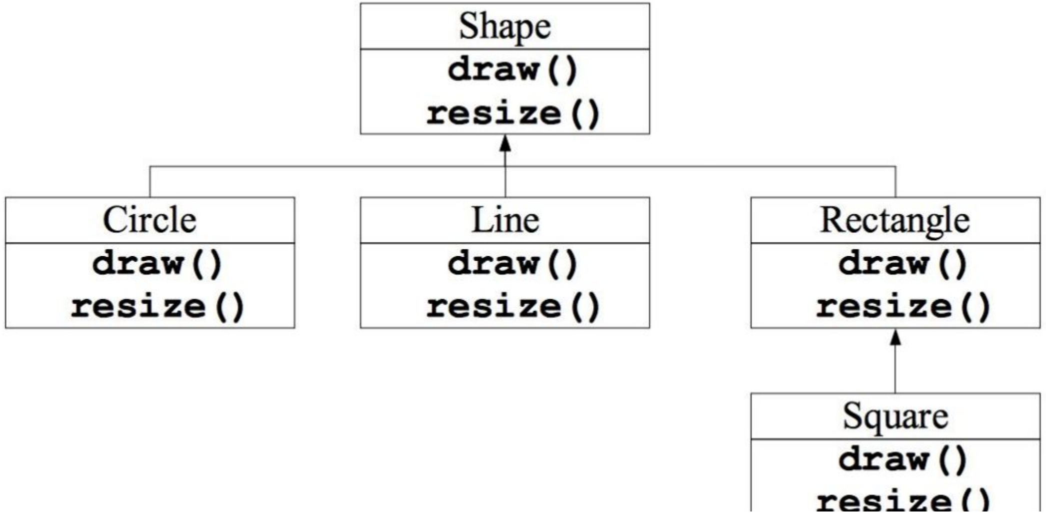
Aggregation and Decomposition

```
Class Engine{...}

Class Door{...}

Class Gearbox{...}
```

```
Class Car{

    private Engine engine;
    private Door[] doors;
    private Gearbox gearbox;

    public Car(){...}

}
```

Generalization and Specialization

Inheritance: Get the interface from the general class.



Subtyping

- A is a subtype of B => everywhere B can be used, A can be used
- Subtypes adds functionality i.e. more operations than super class
- Should not have any assumptions about super class

Three ways to do this in Java

- interface A **extends** interface B, interface C …
- class A **implements** interface B, interface C …
- class A **extends** class B

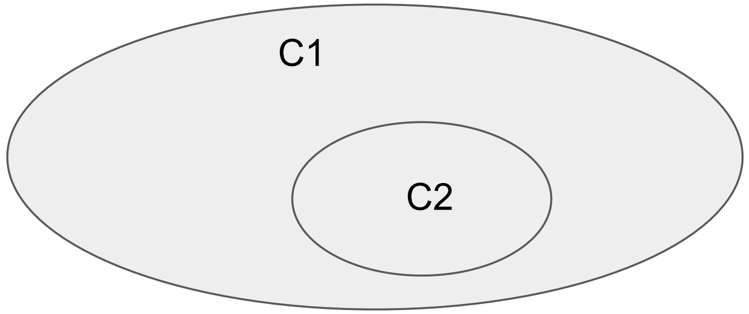
Multiple interfaces allowed but can only extend one class

Inheritance

- This is used to derive a new class from existing class.
- This can be used in:
    - Specializing an abstract data type
    - Extending an existing class

Inheritance

- Extension of the new class is a subset of the more general class



- Is-a relationship:
    - A C2 object is a C1 object (not vice-versa)
    - Example: Human is a Mammal

Inheritance

- The object class is the root of the inheritance hierarchy in Java.
- If no superclass is defined, the class implicitly inherits Object
- If a superclass is specified explicitly, the subclass will inherit indirectly from Object

Interface

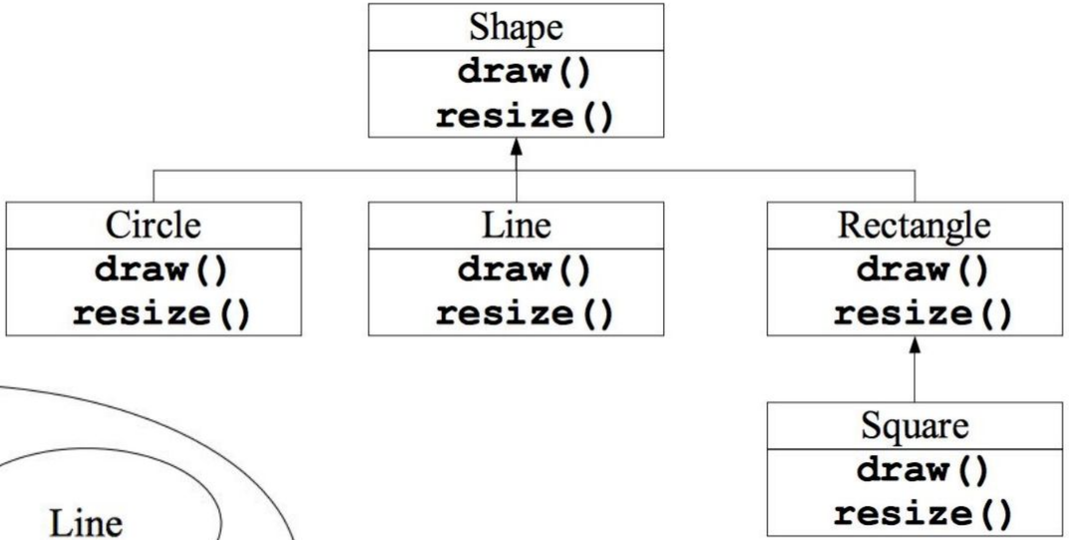
- This is a collection of method declarations
- Doesn’t have any fields. No state
- It tells what a class must do not how
- Describes a set of methods that a class can be forced to implement
- Can be used to define a set of “constants”
- Can be used as a type concept
- Can be used to implement multiple inheritance like hierarchies

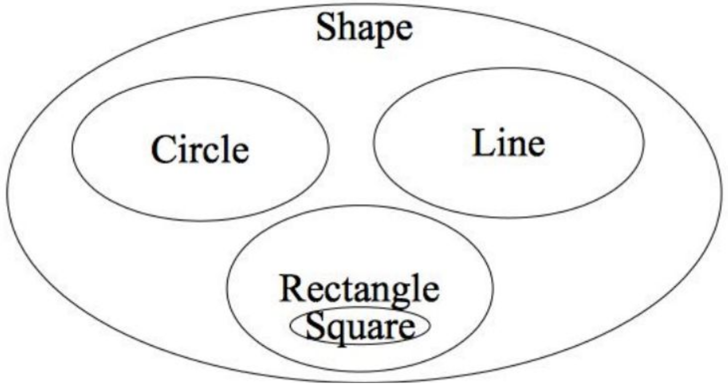
Interface vs Abstract Classes

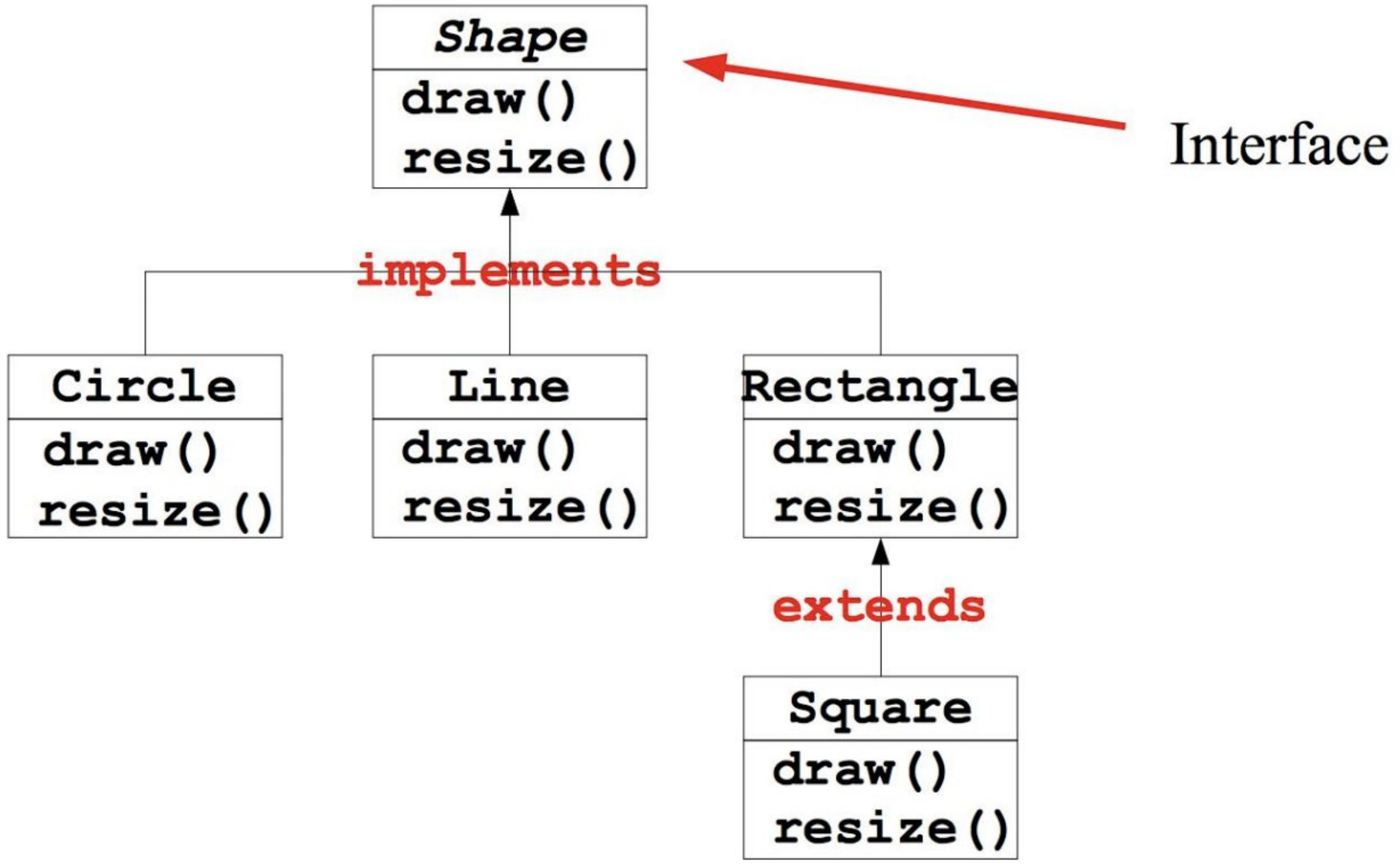
- Methods can be declared
- No method bodies
- Constants can be declared
- Has no constructors
- Multiple inheritance possible
- Has no top interface
- Multiple parent interfaces

- Methods can be declared
- Methods can be implemented
- All types of variables can be declared
- Can have constructors
- Multiple inheritance is not possible
- Always inherits from Object
- Only one parent

Example


Shape
draw()
resize()
Circle
draw()
resize()
Line
draw()
resize()
Rectangle
draw()
resize()
Square
draw()
resize()


Shape
Circle
Line
Rectangle
Square


Shape
draw()
resize()
Interface
implements
Circle
draw()
resize()
Line
draw()
resize()
Rectangle
draw()
resize()
extends
Square
draw()
resize()

```
public interface Shape {
      double PI = 3.14;  // static and final (constant) and uppercase (convention)
      void draw();
      void area();
}

public class Rectangle implements Shape {
      public void draw() { /* do stuff */; System.out.println(“Drew Rectangle”) };
      public void area() { /* calculate area */ };
}

public class Square extends Rectangle {
      public void draw() { /* do stuff */; System.out.println(“Drew Square”) };
}
```

Memory Model and Object Oriented Style

```
Set s = new ListSet();
// s is on the stack. The new object, let us call it o, is on the heap. s is a pointer to this new object.
```

```
s.add(“hi”);
// the above command dereferences s and updates the object o to contain the value “hi”
```

```
Set s2 = s;
// This command creates an alias of s. s2 also now points to o
```

```
s2.remove(“hi”);
// This removes “hi” from object o
```

Parameter Passing

In java, parameter is passed by value. The value of the actual parameter is copied into the formal parameter. Values are always pointers except for primitives

This is basically saying that value of actual parameter cannot be changed by a function call.

```
int add(int x, int y) {
	x +=y;
	return x;
}

void foo() {
	int x = 4;
	int y = 2;
	int z = add(x,y);
}
```

```
class Integer {
	int i;
	Integer(int val) { this.i = val; }
}

Integer new_plus(Integer a, Integer b) {
	a = new Integer(a.i + b.i);
	return a;
}

Integer plus(Integer a, Integer b) {
	a.i += b.i;
	return a;
}
```

```
void foo() {
	Integer x = new Integer(4);
	Integer y = new Integer(3);
	Integer z = new_plus(x,y);
	Integer s = plus(x,y)
}
```

After calling new_plus x still points to original and is unchanged. z points to a new object with value 7 in it.

After calling plus both x and s point to the same object that pointed to x. Now this object holds value 7 instead of 4

JMM

R1 and R2 are thread local variables. A & B are shared variables. Initially A = B = 0.

Impossible to get (R2=2 && R1=1)?

Thread 1	Thread 2
1: R2 = A	3: R1 = B
2: B = 1	4: A = 2

JMM - Data race

R1 and R2 are thread local variables. A & B are shared variables. Initially A = B = 0.

Impossible to get (R2=2 && R1=1)?

Thread 1	Thread 2
1: R2 = A	3: R1 = B
2: B = 1	4: A = 2

Compiler reorders and leads to race condition.

Thread 1	Thread 2
B = 1	3: R1 = B
R2 = A	4: A = 2

JMM - Synchronized

Synchronized statement -> locks the objects -> performs the task -> unlock

Synchronized methods -> locks the method and the object associated with the method -> perform the task -> unlock

JMM.jar code review

Questions

References

https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html

https://docs.oracle.com/javase/tutorial/getStarted/index.html

https://docs.oracle.com/javase/tutorial/java/index.html

https://docs.oracle.com/javase/tutorial/essential/index.html