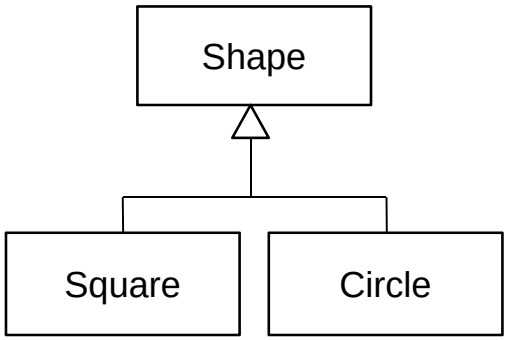
Abstract Classes

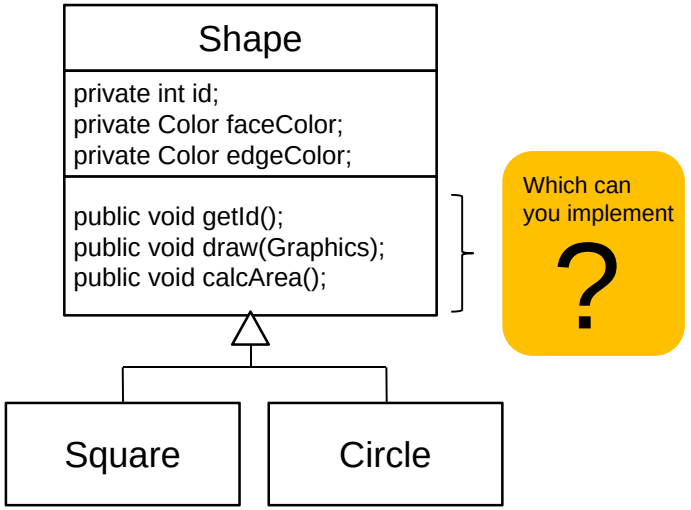
Abstract Classes

- An *abstract class* is a class with partial implementation. → Some methods are not implemented.
- Any classes that you do not want anyone to instantiate (create an object of that class) should be declared abstract.

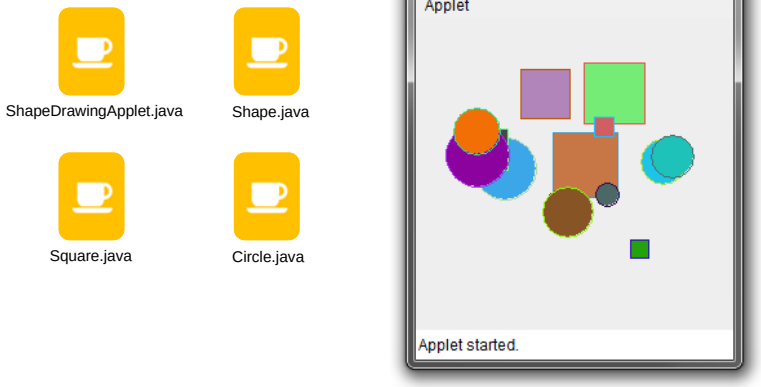
Abstract Classes



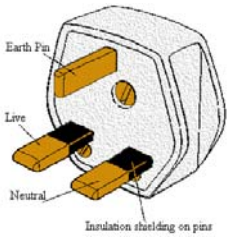
Abstract Classes



Abstract Class Example

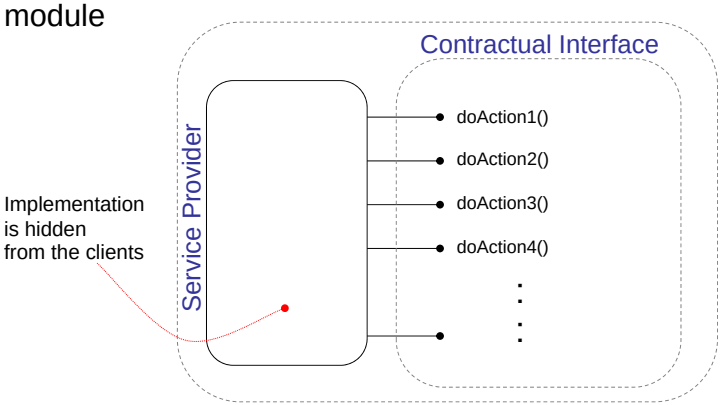


Interfaces



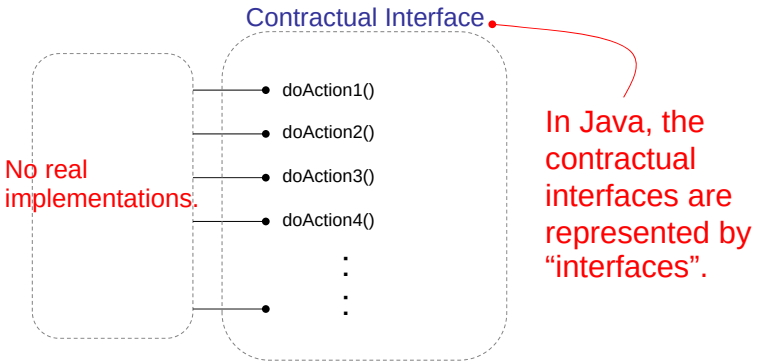
Interfaces

The implementation of a module should be separated from its *contractual interface* and hidden from the clients of the module

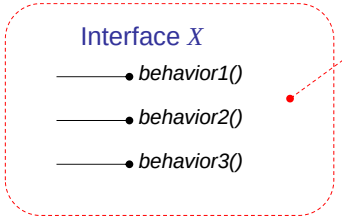


Interfaces

If the contractual interface is completely separated from the implementation, the contractual interface can exist on its own



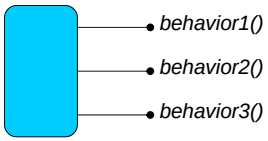
Implementing Interfaces



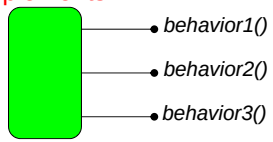
This is just a contractual interface.

Interfaces will be implemented in real classes.

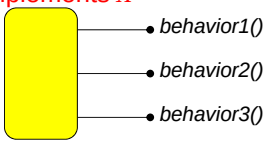
Class *C1* implements *X*



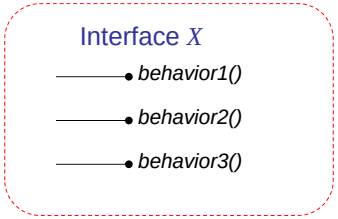
Class *C2* implements *X*



Class *C3* implements *X*



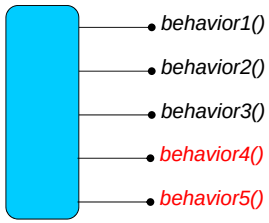
Implementing Interfaces



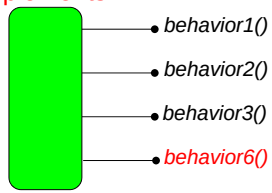
Any classes implementing interface *X*, must implement (provide details of) all of the methods declared in the interface *X*.

Those classes may have additional methods.

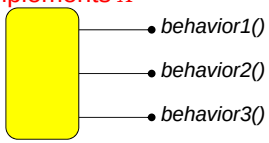
Class *C4* implements *X*



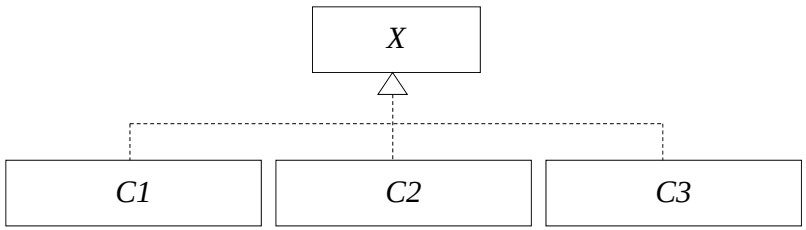
Class *C5* implements *X*



Class *C6* implements *X*



UML Notation for Interfaces



Classes *C1*, *C2*, and *C3* implement the interface *X*.

Single Vs. Multiple Inheritance

Student

float gpa
float *getGPA()*
getGPA() is implemented.

not allowed in Java

inherit

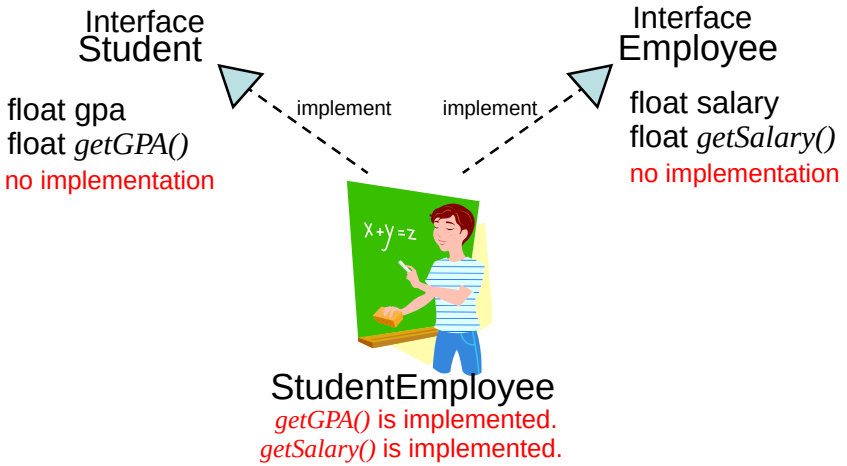
inherit

Employee

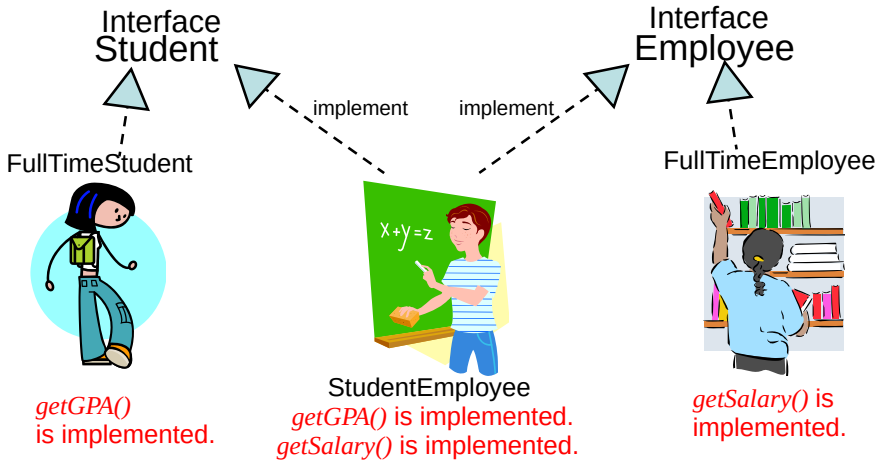
float salary
float *getSalary()*
getSalary() is implemented.

StudentEmployee

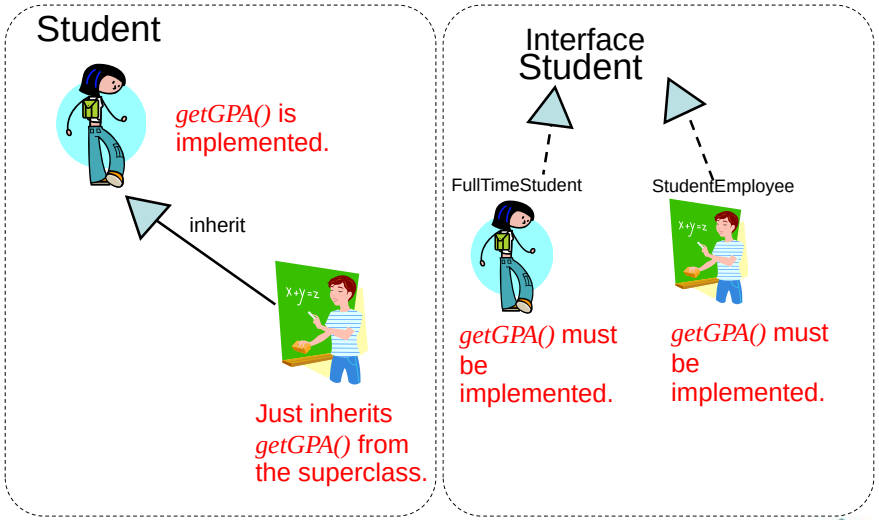
Single Vs. Multiple Inheritance



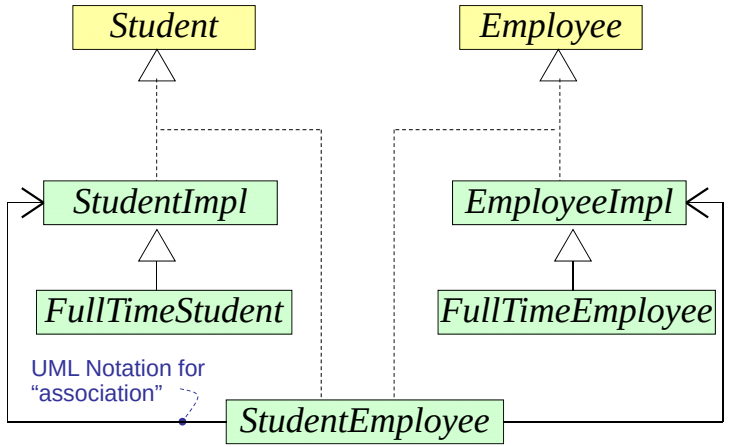
Single Vs. Multiple Inheritance



Re-use Advantage?



Delegation



```
interface Student{
	public float getGPA();
}
```

```
interface Employee{
	public float getSalary();
}
```

```
public class StudentImpl implements Student{
	private float gpa;
	public float getGPA(){
		return gpa;
	}
}
```

```
public class EmployeeImpl implements Employee{
	private float salary;
	public float getSalary(){
		return salary;
	}
}
```


```
public class FullTimeStudent extends StudentImpl{
}
```

```
public class FullTimeEmployee extends EmployeeImpl{
}
```

```
public class StudentEmployee implements Student, Employee{
	private StudentImpl studentImpl = new StudentImpl();
	private EmployeeImpl employeeImpl = new EmployeeImpl();
	public float getGPA(){
		return studentImpl.getGPA();
	}
	public float getSalary(){
		return employeeImpl.getSalary();
	}
}
```

The methods delegate the tasks to other objects.

Threads

A Brief Overview

Threads

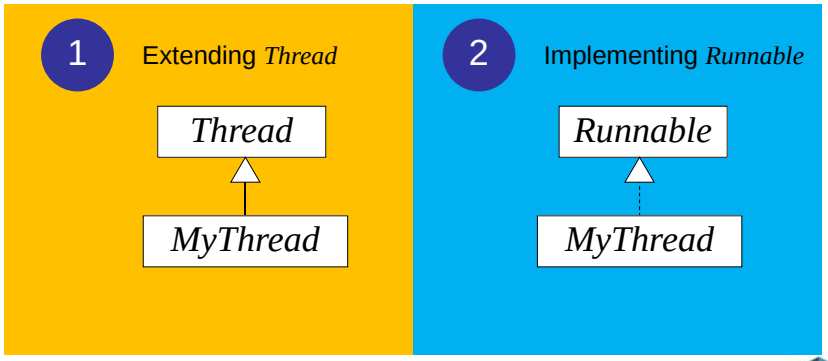
- A *thread* is a *single sequential flow of control* within a program.
- A *multi-threaded* program has multiple threads (flows of control) running simultaneously.
- Most modern OS allows multiple threads to run on a single processor on a time-sharing basis.

Advantages of Multi-threaded Programs

- Applications are more responsive to the users.
  - Monitoring
  - User Interface
- allows a servers to handle multiple clients.
- takes advantage of multiple processors → executing threads on different processors in parallel.

Creation of Threads

- A thread is an instance of the *Java.lang.Thread* class.
- We make use of *Java.lang.Thread* in two ways:



Extending *Thread*

- The subclass MUST override *run()*.
- The *run()* method of a thread is invoked when execution starts.
- The execution of that thread ends when its *run()* method returns.

creating an instance of *MyThread*

```
Thread t = new MyThread();
t.start();
```

starting the thread.
run() of that instance of *MyThread* will be invoked automatically.

Do not invoke *run()* directly.

Example: InfiniteCounter

```
>java InfiniteCounter
0       (By Thread ID = 1)
1       (By Thread ID = 1)
2       (By Thread ID = 1)
3       (By Thread ID = 1)
4       (By Thread ID = 1)
5       (By Thread ID = 1)
6       (By Thread ID = 1)
7       (By Thread ID = 1)
8       (By Thread ID = 1)
9       (By Thread ID = 1)
10      (By Thread ID = 1)
11      (By Thread ID = 1)
12      (By Thread ID = 1)
13      (By Thread ID = 1)
14      (By Thread ID = 1)
15      (By Thread ID = 1)
16      (By Thread ID = 1)
17      (By Thread ID = 1)
18      (By Thread ID = 1)
19      (By Thread ID = 1)
20      (By Thread ID = 1)
21      (By Thread ID = 1)
22      (By Thread ID = 1)
23      (By Thread ID = 1)
24      (By Thread ID = 1)
25      (By Thread ID = 1)
26      (By Thread ID = 1)
27      (By Thread ID = 1)
28      (By Thread ID = 1)
29      (By Thread ID = 1)
30      (By Thread ID = 1)
```

Each output is 1000 milliseconds apart.

InfiniteCounter.java

Example: InfiniteCounter

```
public class InfiniteCounter extends Thread{
       int delay =1000; int id; int i =0;
       public InfiniteCounter(int id,int delay){
              this.id =id; this.delay =delay;
       }
       public void run(){
         try{
           while(true){
              System.out.println(i+"\t(By Thread ID ="+id+")");
              i++;
              this.sleep(delay);
           }
         }catch(InterruptedException e){}
       }
       public static void main(String[] args){
              new InfiniteCounter(1,1000).start();
              new InfiniteCounter(2,2000).start();
       }
}
```

more than a single thread

Example: InfiniteCounter

```
>java InfiniteCounter
0       (By Thread ID = 1)
0       (By Thread ID = 2)
1       (By Thread ID = 1)
1       (By Thread ID = 2)
2       (By Thread ID = 1)
3       (By Thread ID = 1)
2       (By Thread ID = 2)
4       (By Thread ID = 1)
5       (By Thread ID = 1)
3       (By Thread ID = 2)
6       (By Thread ID = 1)
7       (By Thread ID = 1)
4       (By Thread ID = 2)
8       (By Thread ID = 1)
9       (By Thread ID = 1)
5       (By Thread ID = 2)
10      (By Thread ID = 1)
11      (By Thread ID = 1)
6       (By Thread ID = 2)
12      (By Thread ID = 1)
13      (By Thread ID = 1)
7       (By Thread ID = 2)
14      (By Thread ID = 1)
15      (By Thread ID = 1)
8       (By Thread ID = 2)
16      (By Thread ID = 1)
17      (By Thread ID = 1)
9       (By Thread ID = 2)
18      (By Thread ID = 1)
19      (By Thread ID = 1)
10      (By Thread ID = 2)
20      (By Thread ID = 1)
21      (By Thread ID = 1)
```

Output from the first *InfiniteCounter* instance

Output from the second *InfiniteCounter* instance

Thread and *Applet*

What if we want to create a Java applet that also extends *Thread*

```
extends Thread
```

```
extends java.applet.Applet
```

Recall that Java does not allow multiple inheritance.

Implementing *Runnable*

- Instead of creating a class that extends *Thread*, we create a class that implement the *Runnable* interface (which mean the class can still extend another class such as *Applet*).
- Give the implementation of the *run()* method in the new class.
- When creating an instance of *Thread*, use the following constructor.

```
public Thread(Runnable target)
```

This creates an instance of *Thread* whose *run()* method is overridden by the one in **`target`**.

Animation Applets

extending *Applet* while implementing *Runnable*

Animation Applet Example


DigitalClockApplet.java


DigitalClockApplet2.java

Concurrent Programming

- More to come later in the course:
  - Controlling Threads
  - Communication among threads
  - Thread Synchronization
  - Thread Priority and Scheduling
  - …

Self-Study for Topic 2

A: **Trail: Learning the Java Language:**
http://docs.oracle.com/javase/tutorial/java/TOC.html

Read all pages under “Interfaces and Inheritance”.

Self-Study for Topic 2

B: **Trail: 2D Graphics:**

http://docs.oracle.com/javase/tutorial/2d/TOC.html

Read all pages under “Overview of the Java 2D API Concepts”.

Read all pages under “Working with Geometry”

Read all pages under “Working with Text APIs” except “Advanced Text Display”

Read “Working with Images” only in “Reading/Loading an Image” and “Drawing an Image”

Self-Study Test: Topic 1

The test must be done during:

Saturday 23 August
to Monday 25 August

in the “Assessment” section of

my CourseVille