Abstract Classes

- Unlike classes, these cannot be instantiated.
- Like classes, they introduce types
  - but no objects can have as actual type the type of an abstract class.
- Why use them?
  - Because there is a set of common features and implementation for all derived classes but...
  - We want to prevent users from handling objects that are too generic (Example 1)
  - We cannot give a full implementation for the class (Example 2)

Example 1

- The problem:
  - Students are either undergraduate, PhD or MSc.
  - We want to guarantee that nobody creates a Student object. The application always creates a specific kind of Student.
- The solution:
  - Declare Student as abstract.
- Why have the Student class in the first place?
  - A common implementation of common aspects of all students. (e.g. setLogin() and getLogin())
  - A placeholder in my hierarchy that corresponds to a significant concept in my problem domain.
  - To handle all students independently of their subclass using type Student and polymorphism.

Abstract Classes in Java

```java
public abstract class Student {
    protected String login, department, name;
    public Student() {
        login = ""; department = ""; name = "";
    }
    public void setLogin(String login) {
        this.login = new String(login);
    }
    public String getLogin() {
        return new String(login);
    }
}
```

```
public class PhdStudent extends Student {
    private String supervisor;
    public PhdStudent(String login) {
        //...
    }
    public void setSupervisor(String login) {
        //...
    }
}
```
Example 2

The Problem
- How do we calculate the area of an arbitrary shape?
- We cannot allow Shape objects, because we cannot provide a reasonable implementation of getArea();

The Solution
- So we declare the Shape to be an abstract class.
- Furthermore, we declare getArea() as an abstract method because it has no implementation.

Why have the Shape class in the first place?
- Same reasons as for Student: a common implementation, a placeholder in the hierarchy and polymorphism.
- Plus that we want to force all shapes to provide an implementation for getArea();

Abstract Classes

What are the differences between both examples?
- In Example 1, I choose to declare Student abstract because I think it is convenient to prevent the existence of plain Students.
- In Example 2, I must declare Shape abstract because it lacks an implementation for getArea();

Abstract Methods in Java

public abstract class Shape {
    private int colour;
    public Shape() {colour = BLACK;}
    public void setColour(int c) {colour = c;}
    public abstract double getArea();
}

public class Circle extends Shape {
    private int radius;
    public Circle(int r) {radius = r;}
    public double getArea() {return (radius^2)*PI;}
}

Using abstract classes

// Shape s = new Shape(); // ERROR
Shape s = new Circle(4); // Ok
double area = s.getArea(); // Ok - Remember polymorphism?
Circle c = new Circle(3); // Ok
c.setColour(GREEN); // Ok
area = c.getArea(); // Ok

- Class Shape cannot be instantiated (it provides a partial implementation)
- Abstract methods can be called on an object of apparent type Shape (they are provided by Circle) (Polymorphism)
Interfaces

- An interface is a set of methods and constants that is identified with a name.
- They are similar to abstract classes
  - You cannot instantiate interfaces
  - An interface introduces types
  - But, they are completely abstract (no implementation)
- Classes and abstract classes realize or implement interfaces
  - They must have (at least) all the methods and constants of the interface with public visibility

Example: Clock Interface

```java
interface Clock {
    Time MIDNIGHT = new Time(0, 0, 0);
    void setTime(Time t);
}

class DigitalClock implements Clock {
    private Time currentTime;
    public DigitalClock() {reset();}
    public void setTime(Time t) {currentTime = new Time(t);}
    public void reset() {setTime(MIDNIGHT);}
}
```

Interface Hierarchies

- Interfaces can extend each other

```java
interface Clock {
    Time MIDNIGHT = new Time(0, 0, 0);
    void setTime(Time t);
}

interface AlarmClock extends Clock {
    void setAlarm(Time t, Wakeable w);
}
```

Why use Interfaces?

- To separate (decouple) the specification available to the user from implementation
  - I can use any class that implements the interface through the interface type (i.e. polymorphism)
- As a partial solution to Java's lack of multiple inheritance
Decoupling: Alarm Clock Example (1/3)

```
interface Clock
  MIDNIGHT:Time
  setMIDNIGHT(Time): void

interface Wakeable
  +wake()

interface AlarmClock
  +setAlarm(Time, Wakeable): void
```

```
AnalogueClock
  reset(): void
```

```
DigitalClock
```

Decoupling: Alarm Clock Example (2/3)

```
class Sleeper implements Wakeable
  private boolean sleeping;
  public Sleeper() { sleeping = false; }
  public void sleep() { if (!sleeping) { sleeping = true; System.out.println("Yawn… Time for a nap!… Zzzz..."); } }
  public void wake() { if (sleeping) { sleeping = false; System.out.println("What? Is it time to wake up?"); } }
  public boolean sleeping() { return sleeping; }
```

Decoupling: Alarm Clock Example (3/3)

```
public static void main(String[] args) {
  AlarmClock a = new DigitalClock();
  AlarmClock b = new AnalogueClock();
  Sleeper me = new Sleeper();
  Sleeper programmingClass = new Sleeper();
  me.sleep();
  programmingClass.sleep();
  a.setAlarm(new Time(7, 0, 0), me); // Wake up at 7am, plenty of time before lecture
  b.setAlarm(new Time(10, 50, 0), programmingClass); // Wake up just before end of lecture
}
```

Multiple Inheritance

```
interface Wakeable
  void wake();
```

```
interface AlarmClock
  start() setTimer() setAlarm()
```

```
interface StopWatch
  start() stop()
```

```
MultiFunctionWatch
```

```
Java does not support multiple inheritance, but...
```

Polymorphism at work. Using a digital clock as in implementation for AlarmClock
Multiple Interfaces

- Classes are allowed to implement multiple interfaces

```java
interface StopWatch
+start()
+stop();

interface AlarmClock
+setTime()
+setAlarm();
```

Q: Why is this not the same as multiple inheritance?

A: There is no implementation to inherit

---

Conflict resolution rules

- Classes can implement multiple interfaces
  - Name conflicts: multiple methods with the same name
  - Name conflict resolution:
    - different signatures: overloading
    - same signature and return type: same method
    - same signature and different return type: compile error

---

Abstract classes vs. Interfaces

- Can have data fields
- Methods may have an implementation
- Classes and abstract classes extend abstract classes.
- Class cannot extend multiple abstract classes
- Substitution principle is assumed

- Can only have constants
- Methods have no implementation
- Classes and abstract classes implement interfaces
- Interfaces can extend multiple interfaces
- A class can implement multiple interfaces
- Substitution principle not assumed

---

Abstract Classes or Interfaces?

- If there is common implementation -> AC
- If there is no common implementation ->
  - Interfaces allow classes implementing multiple interfaces...
  - Abstract classes can be subsequently extended without breaking subclasses...
  - No clear cut decision...
- Let look at what the Java Standard Class Library developers do...
Collections

Extract from the Java Standard Library hierarchy for collections.

Abstract classes and methods
Concrete classes
Interfaces
Interface Hierarchies
Multiple Inheritance
Multiple interfaces
Abstract classes vs. Interfaces
Specialisation (extends) vs. Realization (implements)