Announcements

```
class TAs extends Human {
  DateTimeInterval office_hours;

  String ask_for_help(Student s, DateTime t)
  {
    if (!t.in(office_hours) || !s.respects(this))
    {
      this.go_nuts(Student s);
    }
    return “Sure, I’ll help you”;
  }

  void go_nuts(Student s)
  {
    s.fail();
  }
}
```

Abstract classes

- An abstract class has abstract methods and can have non-abstract methods (which usually represent the “default behaviour” of a method:)

```
abstract class Creature
{
  boolean alive, hungry;
  abstract void move();
  void eat()
  {
    System.out.println(“Hmmm...”);
    hungry = false;
  }
}
```

- The abstract methods *must* be implemented in the subclasses of an abstract class (unless the subclass itself is also abstract.) This is, there is no default behaviour for an abstract method.

Interfaces

- Interfaces are (equivalent to) purely abstract classes, i.e. classes where all the methods are abstract

```
interface Creature
{
  void move();
  void eat();
}
```

is (almost) the same as

```
abstract class Creature
{
  abstract void move();
  abstract void eat();
}
```

Interfaces

```
class Human implements Creature
{
  void move()
  {
    System.out.println(“I’m walking...”);
  }
  void eat()
  {
    System.out.println(“I’m eating...”);
  }
  void jump()
  {
    System.out.println(“Up and down...”);
  }
}
```

Using interfaces for generalization

```
class CDPlayer {
  int song;
  boolean stopped;
  CDPlayer()
  {
    stopped = true;
    song = 0;
  }
  void play() { stopped = false; }
  void ff() { song++; }
  void pause() { stopped = true; }
  void stop()
  {
    stopped = true;
    song = 0;
  }
}
```

Using interfaces for generalization

```
class TapeRecorder {
  boolean stopped, recording;
  Tape t;
  TapeRecorder() {
    stopped = true;
    recording = false;
    t = null;
  }
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record(Tape x) {
    recording = true;
    t = x.clone();
  }
}
```

Interfaces

```
interface MusicPlayer {
  void play();
  void ff();
  void pause();
  void stop();
}
```

Interfaces

```
class CDPlayer implements MusicPlayer {
  int song;
  boolean stopped;
  CDPlayer()
  {
    stopped = true;
    song = 0;
  }
  void play() { stopped = false; }
  void ff() { song++; }
  void pause() { stopped = true; }
  void stop()
  {
    stopped = true;
    song = 0;
  }
}
```

Interfaces

```
class TapeRecorder implements MusicPlayer {
  boolean stopped, recording;
  Tape t;
  TapeRecorder() {
    stopped = true;
    recording = false;
    t = null;
  }
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record(Tape x) {
    recording = true;
    t = x.clone();
  }
}
```

Interfaces

```
class PlayerTest {
  static void test(MusicPlayer p)
  {
    p.play();
    p.ff();
    p.pause();
    p.play();
    if (p instanceof TapeRecorder) {
      ((TapeRecorder)p).record(new Tape());
    }
    p.stop();
  }
}
```

Interfaces

```
class SoundStudio {
  public static void main(String[] args)
  {
    MusicPlayer[] players = { new CDPlayer(),
                              new TapeRecorder(),
                              new CDPlayer() };
    for (int i = 0; i < players.length; i++) {
      PlayerTest.test(players[i]);
        // polymorphic call.
    }
  }
}
```

Abstract classes

```
abstract class MusicPlayer {
  boolean stopped;
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  abstract void stop();
}
```

Abstract classes

```
class CDPlayer extends MusicPlayer {
  int song;
  CDPlayer()
  {
    stopped = true;
    song = 0;
  }
  void ff() { song++; }
  void stop()
  {
    stopped = true;
    song = 0;
  }
}
```

Abstract classes

```
class TapeRecorder extends MusicPlayer {
  boolean recording;
  Tape t;
  TapeRecorder() {
    stopped = true;
    recording = false;
    t = null;
  }
  void ff() { }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record(Tape x) {
    recording = true;
    t = x.clone();
  }
}
```

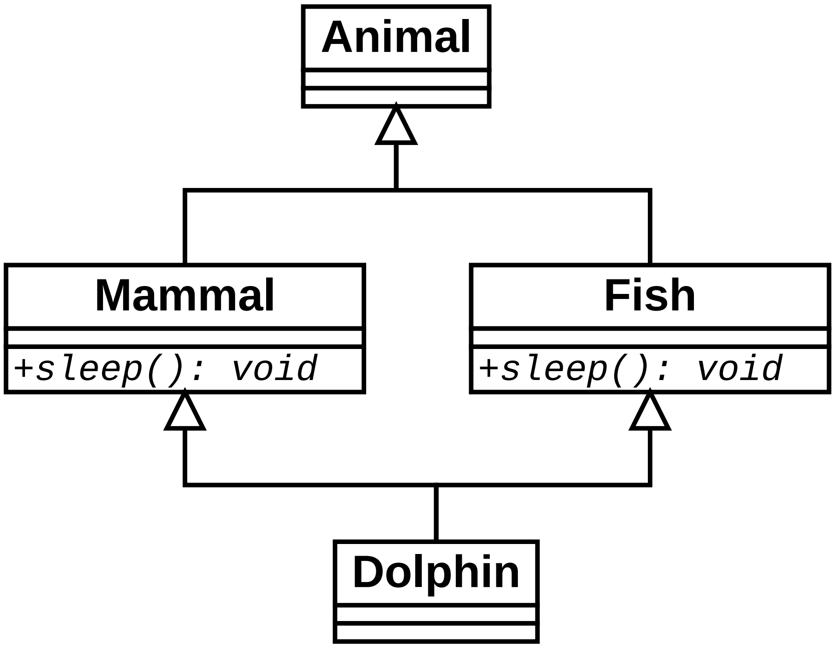
Interfaces

- Multiple inheritance is supported for interfaces

```
class A extends B implements C, D, E { ... }
```

- ...because the methods in the interfaces are abstract, which means that they must be implemented in A, so there is no ambiguity problem when calling a method.

- Interfaces and multiple inheritance



Generic programming

- A generic function/procedure/method is one whose algorithm doesn't depend on the types of its arguments

- Generic procedures are abstract, and therefore highly reusable

- In java generic procedures are implemented using parametric polymorphism

- Example generic sorting:

  - To sort an array of objects, where the objects have a key, and keys are comparable.

```
interface Comparable
{
  public int compareTo(Comparable obj);
}
```

Generic programming

```
class Student extends Human implements Comparable
{
  private String name;
  private long id;
  private int age;
  //...
  public int compareTo(Student s)
  {
    if (this.age < s.age) return -1;
    else if (this.age > s.age) return 1;
    if (this.name.compareTo(s.name) < 0)
      return -1;
    else if (this.name.compareTo(s.name) > 0)
      return 1;
    return 0;
  }
}
```

Generic programming

```
class SortAlgorithms {
  static void insertion_sort(Comparable[] a)
  {
    int i, j;
    Comparable key;
    for (j = 1; j < a.length; j++) {
      key = a[j];
      i = j - 1;
      while (i >= 0
        && key.compareTo(a[i]) < 0 ) {
        a[i+1] = a[i];
        i--;
      }
      a[i+1] = key;
    }
  }
}
```

Generic programming

```
static void insertion_sort(Movie[] a)
{
  int i, j;
  String key;
  for (j = 1; j < a.length; j++) {
    key = a[j].get_title();
    i = j - 1;
    while (i >= 0
      && key.compareTo(a[i].get_title()) < 0 ) {
      a[i+1] = a[i];
      i--;
    }
    a[i+1] = key;
  }
}
```

Generic programming

```
class GenericSortTest {
  public static void main(String[] args)
  {
    Student[] course = new Student[230];
    enter_info(course);
    SortingAlgorithms.insertion_sort(course);
    String[] words = {“one”,“two”,“three”,“four”};
    SortingAlgorithms.insertion_sort(words);
  }
  static void enter_info(Student[] course)
  {
    for (int i = 0; i < course.length; i++) {
      String name = Keyboard.readString();
      int age = Keyboard.readInt();
      long id = Keyboard.readLong();
      course[i] = new Student(name, age, id);
    }
  }
}
```

Generic programming

```
interface Indexed {
  public Comparable get_key();
}

class Student extends Human implements Indexed {
  private long id;
  private StudentKey key;
  public Student(String name, int age, long id)
  {
    key = new StudentKey(name, age);
    this.id = id;
  }
  //...
  public StudentKey get_key()
  {
    return key;
  }
}
```

Generic programming

```
class StudentKey implements Comparable {
  private String name;
  private int age;
  public StudentKey(String n, int a)
  {
    name = n;
    age = a;
  }
  public int compareTo(StudentKey s)
  {
    if (this.age < s.age) return -1;
    else if (this.age > s.age) return 1;
    if (this.name.compareTo(s.name) < 0)
      return -1;
    else if (this.name.compareTo(s.name) > 0)
      return 1;
    return 0;
  }
}
```

Generic programming

```
class SortAlgorithms {
  static void insertion_sort(Indexed[] a)
  {
    int i, j;
    Comparable key;
    for (j = 1; j < a.length; j++) {
      key = a[j].get_key();
      i = j - 1;
      while (i >= 0
        && key.compareTo(a[i].get_key()) < 0 ) {
        a[i+1] = a[i];
        i--;
      }
      a[i+1] = key;
    }
  }
}
```

Changing visibility in subclasses

- A public method cannot be overriden by a private or protected method:

```
class A {
    public void m()
    {
        System.out.println("A");
    }
}
class B extends A {
    private void m()
    {
        System.out.println("B");
    }
}
```

Changing visibility in subclasses

- A method can be overriden by method with weaker access priviledges:

```
class A {
    protected void m()
    {
        System.out.println("A");
    }
}
class B extends A {
    public void m()
    {
        System.out.println("B");
    }
}
```

Using the Object class

```
import java.util.Vector;
class Test {
  void p() {
    Vector v = new Vector();
    v.addElement(new Integer(2));
    v.addElement(new Integer(5));
    v.insertElementAt(new Integer(3), 1);
    Integer i = (Integer)v.elementAt(2);
    int n = i.intValue();
  }
}
```