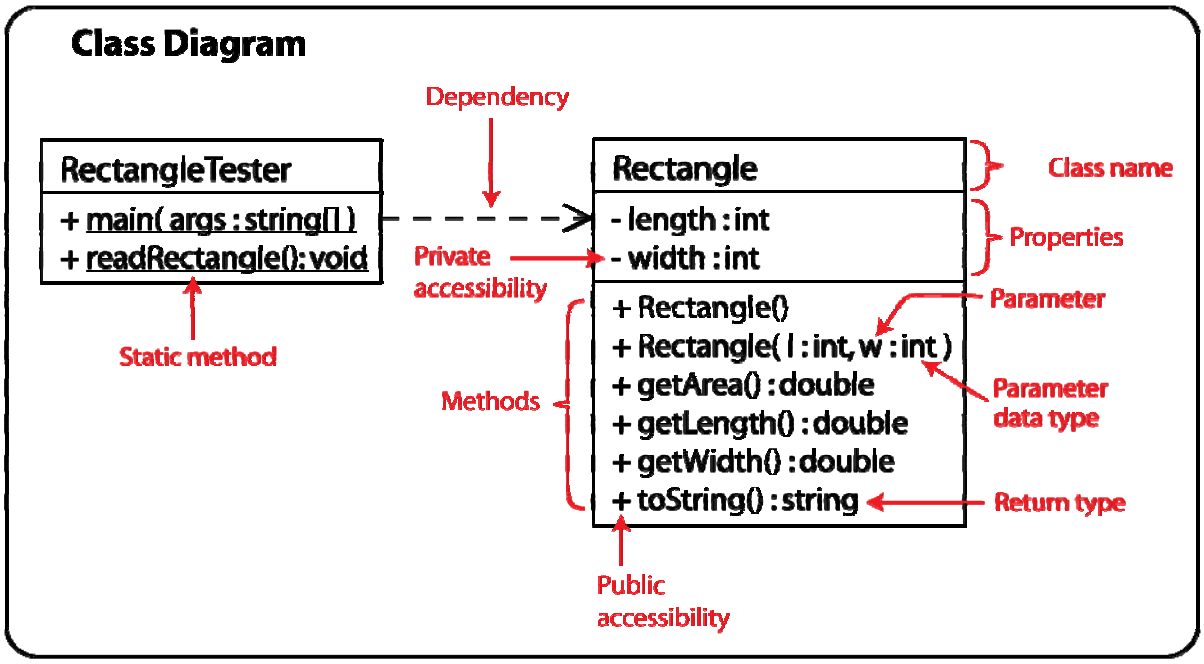
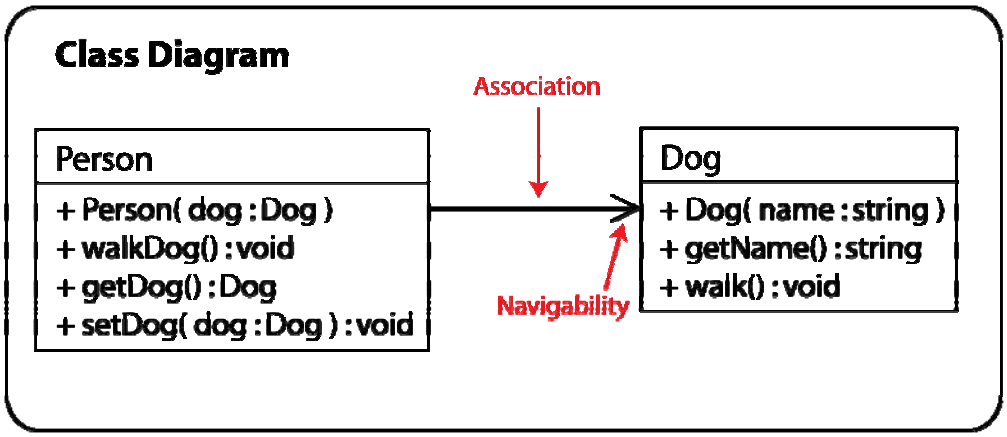
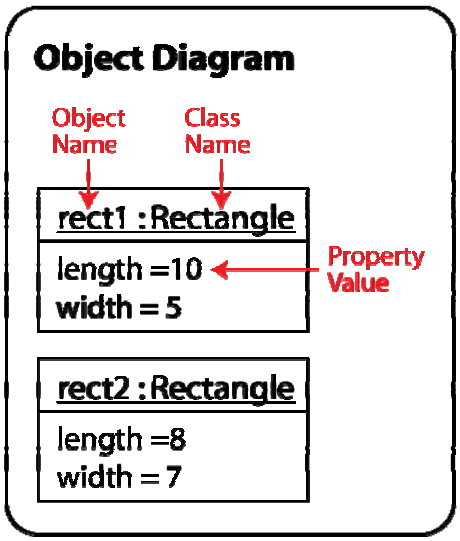
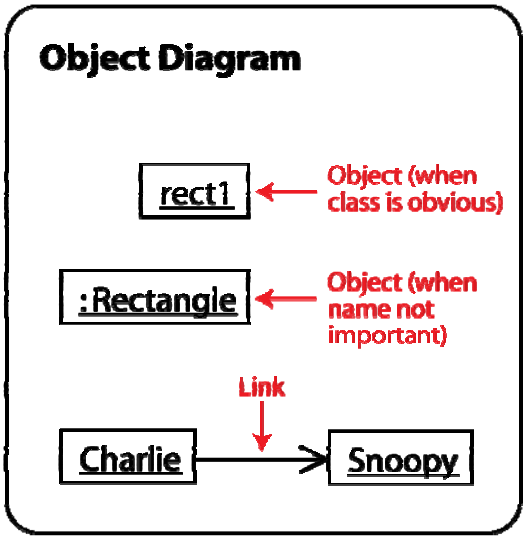
CS 3410 – Programming Review

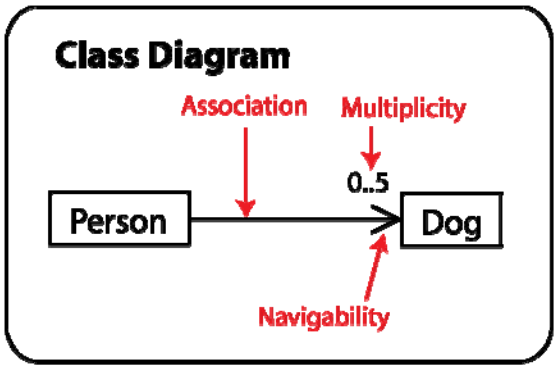
UML Primer



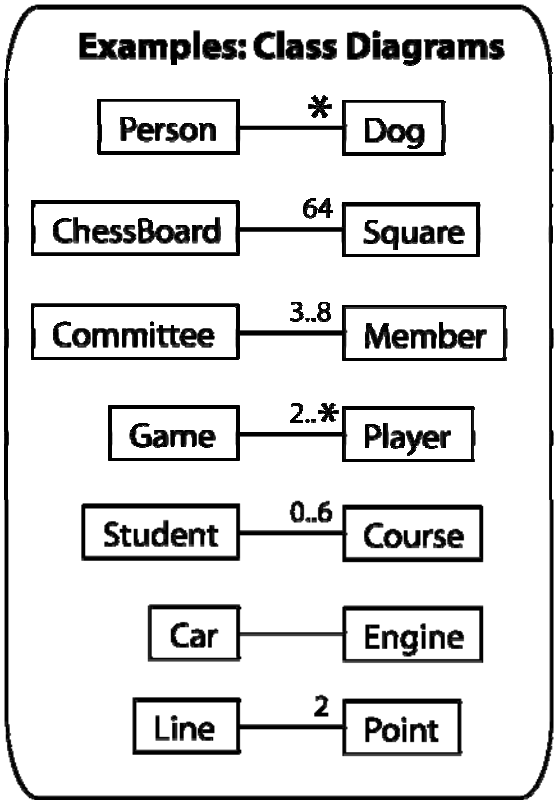


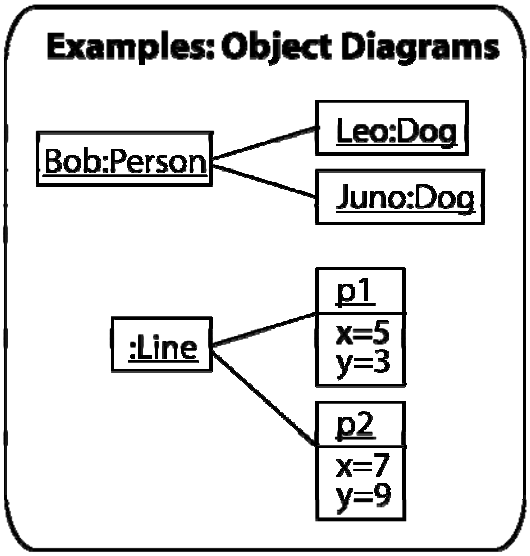


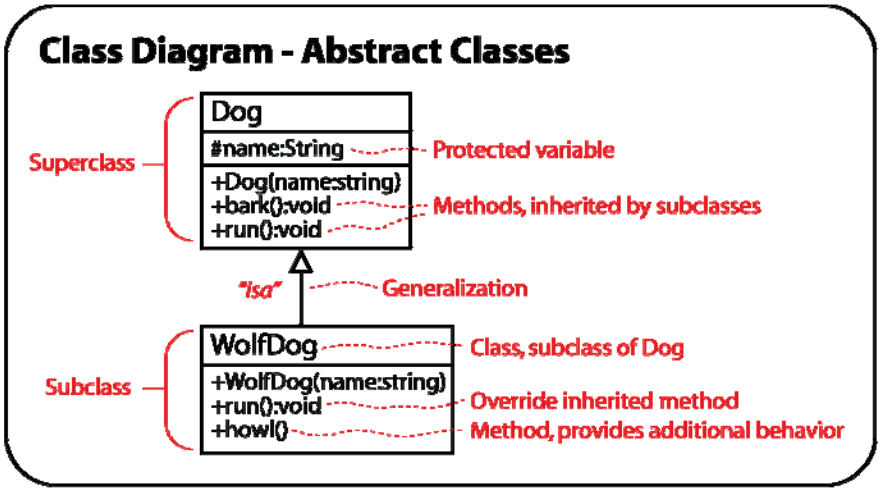


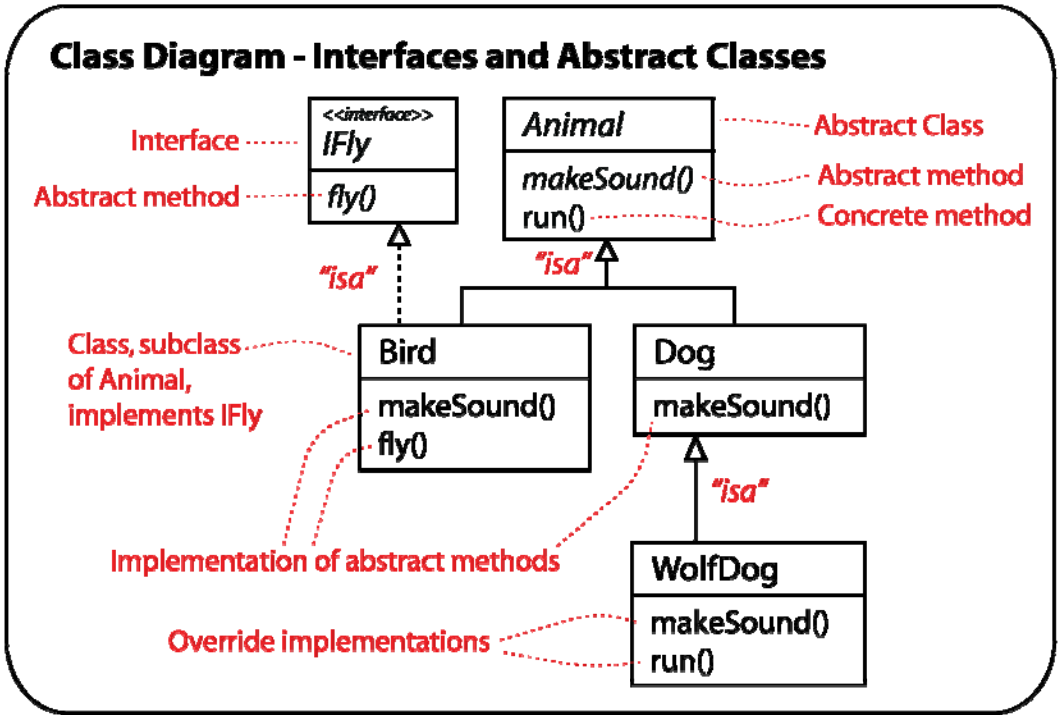


Multiplicity	Meaning
n	exactly *n* instances
a..b	anywhere from *a* to *b*
0..n	up to *n* including 0
*	many or none
*a..**	at least *a*
(blank)	zero or one



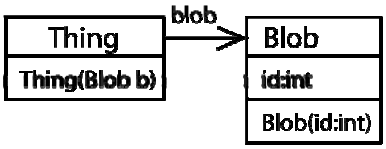



Class Diagram - Abstract Classes
Dog
#name:String
+Dog(name:string)
+bark():void
+run():void
Superclass
Protected variable
Methods, inherited by subclasses
"isa"
Generalization
WolfDog
+WolfDog(name:string)
+run():void
+howl()
Subclass
Class, subclass of Dog
Override inherited method
Method, provides additional behavior


Class Diagram - Interfaces and Abstract Classes
<<interface>>
IFly
fly()
Interface
Abstract method
Animal
makeSound()
run()
Abstract Class
Abstract method
Concrete method
"isa"
"isa"
Class, subclass
of Animal,
implements IFly
Bird
makeSound()
fly()
Dog
makeSound()
"isa"
Implementation of abstract methods
WolfDog
makeSound()
run()
Override implementations

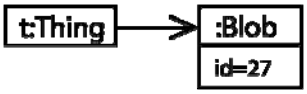
Modeling Association

1. Modeling association



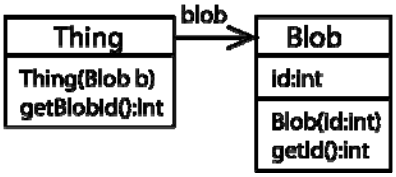

```
class Thing
{
   Blob blob;

   public Thing ( Blob blob ){this.blob = blob;}
}

class Blob
{
    int id;

    public Blob( int i ){ id = i; }
}
```

To use:



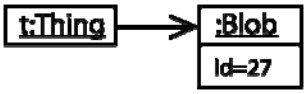

```
Thing t = new Thing( new Blob(27) );

System.out.println( t.blob.id );
```

2. How do we code an association using delegation (and encapsulation)?



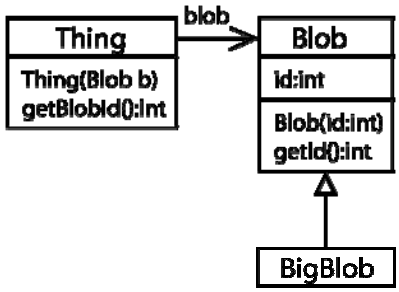

```
class Thing
{
   private Blob blob;

   public Thing ( Blob blob ) {
                                this.blob = blob;}

   public int getBlobId() {
                return blob.getId(); }
}

class Blob
{
   private int id;

   public Blob( int i ) { id = i; }

   public int getId() { return id; }
}
```

To use:



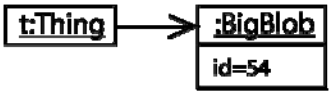

```
Thing t = new Thing( new Blob(27) );

System.out.println( t.getBlobId() );
```

3. Suppose that a Blob can have subclasses and we want the Thing class to work with Blob and any subclasses. Because BigBlob is a Blob we don't have to do anything to the Thing class.




```
class BigBlob extends Blob
{
   public BigBlob( int i ) { super(i*2); }
}
```

To use:



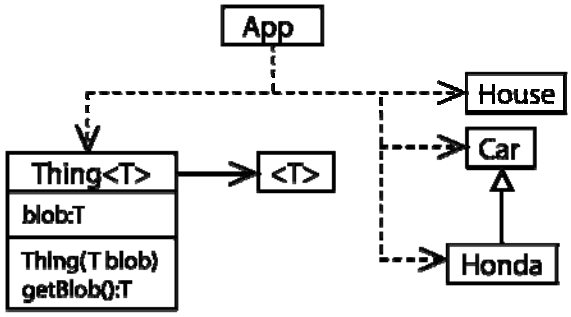

```
Thing t = new Thing( new BigBlob(27) );

System.out.println( t.getBlobId() );
```

Generics

1. How do you create an ArrayList of Blob's using the generics features of Java.

```
// Old style – ArrayList of Objects
ArrayList al = new ArrayList();
al.add( new Blob(32) );
// Must cast
Blob b = (Blob)al.get(0);

// Preferred-safer – ArrayList of Blobs
ArrayList<Blob> al2 = new ArrayList<Blob>();
al2.add( new Blob(8) );
// No Cast needed
Blob b2 = al2.get(0);

// Also allows subclasses
al2.add( new BigBlob(14) );
```

2. What is a generic class? How do you write a generic class?

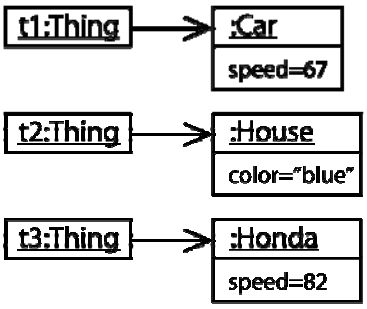



```
class Thing<T>
{
   private T blob;

   public Thing ( T blob ) {
          this.blob = blob; }

   public T getBlob() {
            return blob; }
}
```

```
class Car
{
   private int speed;
   public Car( int i ) {
          speed = i; }
   public int getSpeed() {
              return speed; }
}
```

```
class Honda extends Car
{
   public Honda( int i ) {
          super(i); }
}

class House
{
   private String color;
   public House( String c ) {
          color = c; }
   public String getColor() {
                  return color; }
}
```

To use:




```
   Thing<Car> t1 = new Thing<Car>( new Car(67) );

   Thing<House> t2 = new Thing<House>
       (new House("Blue") );

   House house = t2.getBlob();

   Thing<Honda> t3 = new Thing<Honda>
       ( new Honda(82) );

   Honda honda = t3.getBlob();
```

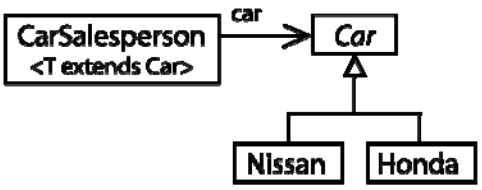
3. What is a generic method? How do we write a generic method?

```
public static <T> void printThing( T blob )
{
   System.out.println( blob );
}
```

To use:

```
House house = new House( "blue" );
Car car = new Car(22);
Honda honda = new Honda(67);

GenericsTests6.<House>printThing( house );
GenericsTests6.<Car>printThing( car );
GenericsTests6.<Honda>printThing( honda );
```

4. How do we restrict (bound) a generic type?

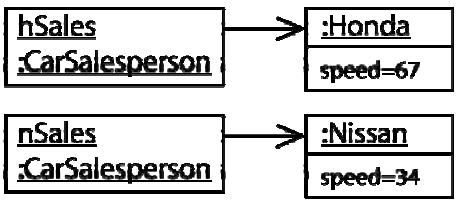



```
class CarSalesperson<T extends Car>
{
   private T car;

   public CarSalesperson( T car ) {
          this.car = car; }

   public T getCar() { return car; }
}
```

```
abstract class Car
{
   private int speed;

   public Car( int i ) {
          speed = i; }
}
```

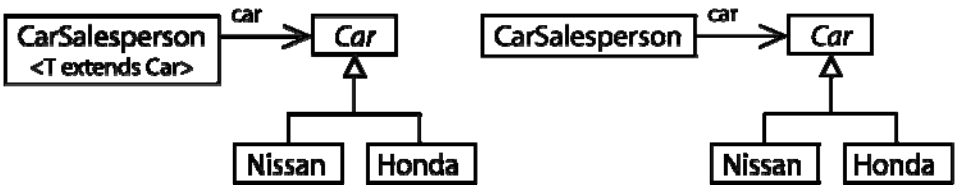
```
class Honda extends Car
{
   public Honda( int i ) { super(i); }
}

class Nissan extends Car
{
   public Nissan( int i ) { super(i); }
}
```

To use:




```
Honda honda = new Honda(67);

CarSalesperson<Honda> hSales = new
        CarSalesperson<Honda>( honda );

Honda h = hSales.getCar();

Nissan nissan = new Nissan(34);

CarSalesperson<Nissan> nSales = new
        CarSalesperson<Nissan>( nissan );

Nissan n = nSales.getCar();
```

How are these two situations different?



5. How do we put a restriction on the generic types of parameters?

```
public static void printCars( Collection<? extends Car> cars )
{
   for( Car car : cars ) System.out.println( car );
}
```

To use:

```
ArrayList<Honda> hondas = new ArrayList<Honda>();
hondas.add( new Honda(52) ); hondas.add( new Honda(43) );
printCars( hondas );

ArrayList<Nissan> nissans = new ArrayList<Nissan>();
nissans.add( new Nissan(33) ); nissans.add( new Nissan(27) );
printCars( nissans );

ArrayList<Car> cars = new ArrayList<Car>();
cars.add( new Nissan(73) ); cars.add( new Honda(44) );
printCars( cars );
```

Suppose that we did not use the restriction on generic type above as in this print method:

```
public static void printCars( Collection<Car> cars )
{
   for( Car car : cars ) System.out.println( car );
}
```

Then this will work:

```
ArrayList<Car> cars = new ArrayList<Car>();
cars.add( new Nissan(73) ); cars.add( new Honda(44) );
printCars( cars );
```

But this will generate a compile error:

```
ArrayList<Nissan> nissans = new ArrayList<Nissan>();
nissans.add( new Nissan(33) ); nissans.add( new Nissan(27) );
printCars( nissans );
```