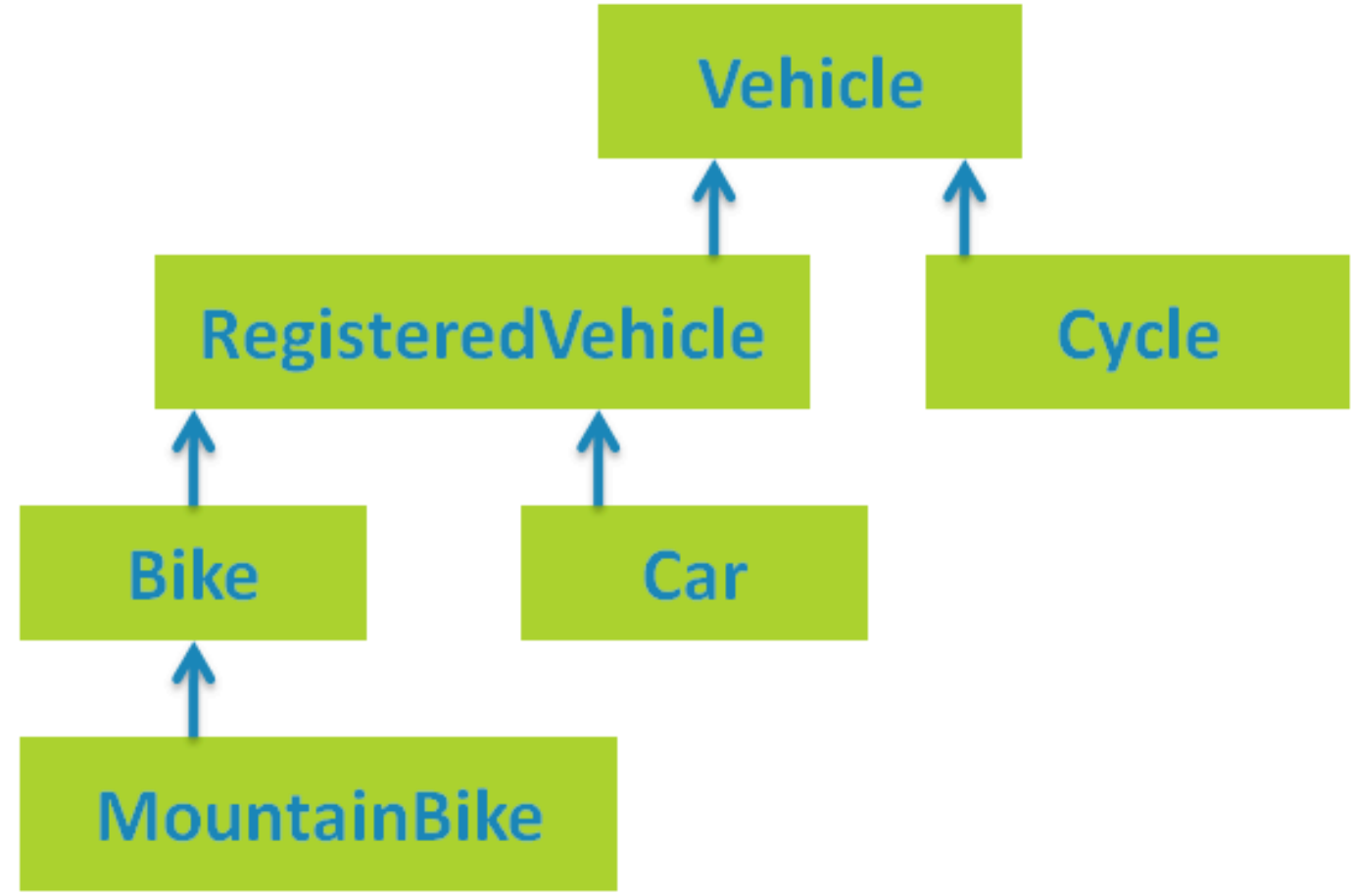
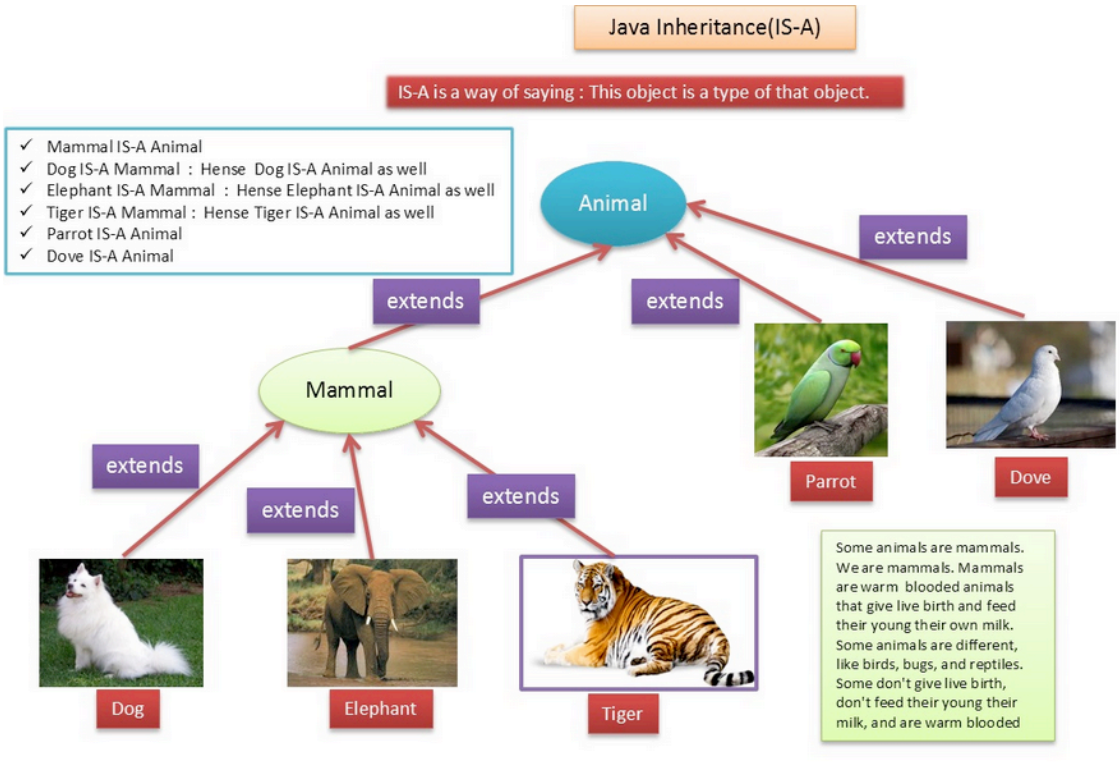
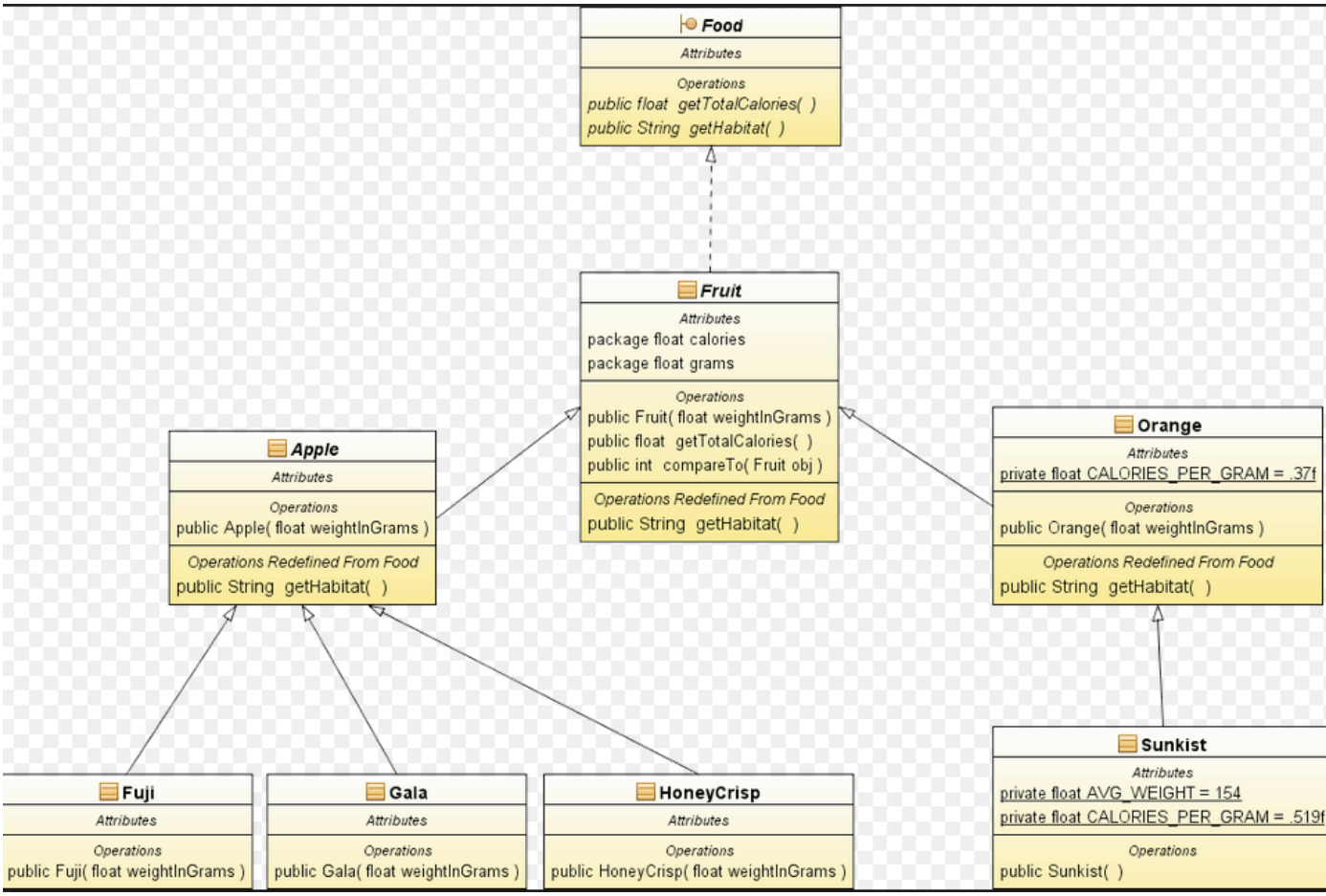
Here are some pictures of inheritance that I found on the web:

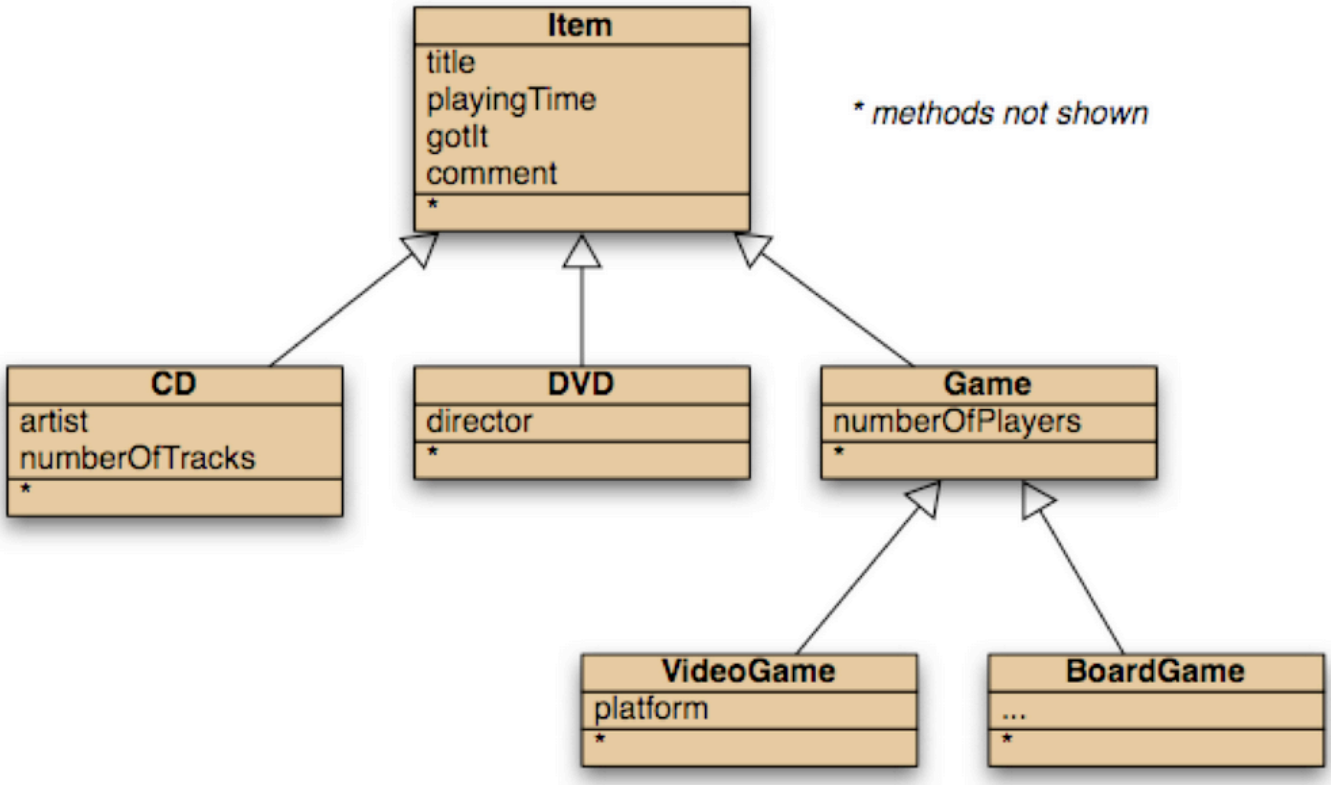
Example of Inheritance hierarchies from the web



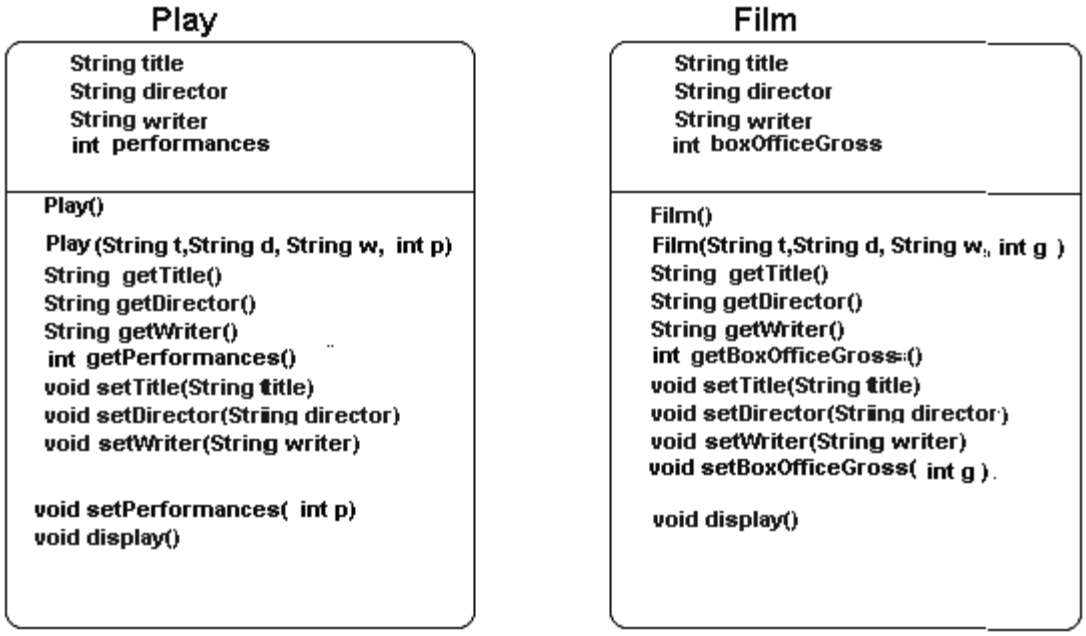
A


Java Inheritance(IS-A)
IS-A is a way of saying : This object is a type of that object.
✓ Mammal IS-A Animal
✓ Dog IS-A Mammal : Hense Dog IS-A Animal as well
✓ Elephant IS-A Mammal : Hense Elephant IS-A Animal as well
✓ Tiger IS-A Mammal : Hense Tiger IS-A Animal as well
✓ Parrot IS-A Animal
✓ Dove IS-A Animal
Animal
extends
extends
extends
Mammal
extends
extends
extends
Parrot
Dove
Dog
Elephant
Tiger
Some animals are mammals. We are mammals. Mammals are warm blooded animals that give live birth and feed their young their own milk. Some animals are different, like birds, bugs, and reptiles. Some don't give live birth, don't feed their young their milk, and are warm blooded
public class Animal
{
}
public class Mammal extends Anima
{
}
public class Dog extends Mammal
{
}
public class Elephant extends Mamm
{
}
public class Tiger extends Mammal
{
}
public class Parrot extends Animal
{
}
public class Dove extends Animal
{
}


Food
Attributes
Operations
public float getTotalCalories( )
public String getHabitat( )
Fruit
Attributes
package float calories
package float grams
Operations
public Fruit(float weightInGrams)
public float getTotalCalories( )
public int compareTo(Fruit obj)
Operations Redefined From Food
public String getHabitat( )
Apple
Attributes
Operations
public Apple(float weightInGrams)
Operations Redefined From Food
public String getHabitat( )
Orange
Attributes
private float CALORIES_PER_GRAM = .37f
Operations
public Orange(float weightInGrams)
Operations Redefined From Food
public String getHabitat( )
Fuji
Attributes
Operations
public Fuji(float weightInGrams)
Gala
Attributes
Operations
public Gala(float weightInGrams)
HoneyCrisp
Attributes
Operations
public HoneyCrisp(float weightInGrams)
Sunkist
Attributes
private float AVG_WEIGHT = 154
private float CALORIES_PER_GRAM = .519f
Operations
public Sunkist( )


Item
title
playingTime
gotIt
comment
*
* methods not shown
CD
artist
numberOfTracks
*
DVD
director
*
Game
numberOfPlayers
*
VideoGame
platform
*
BoardGame
...
*

Once upon a time there were two classes



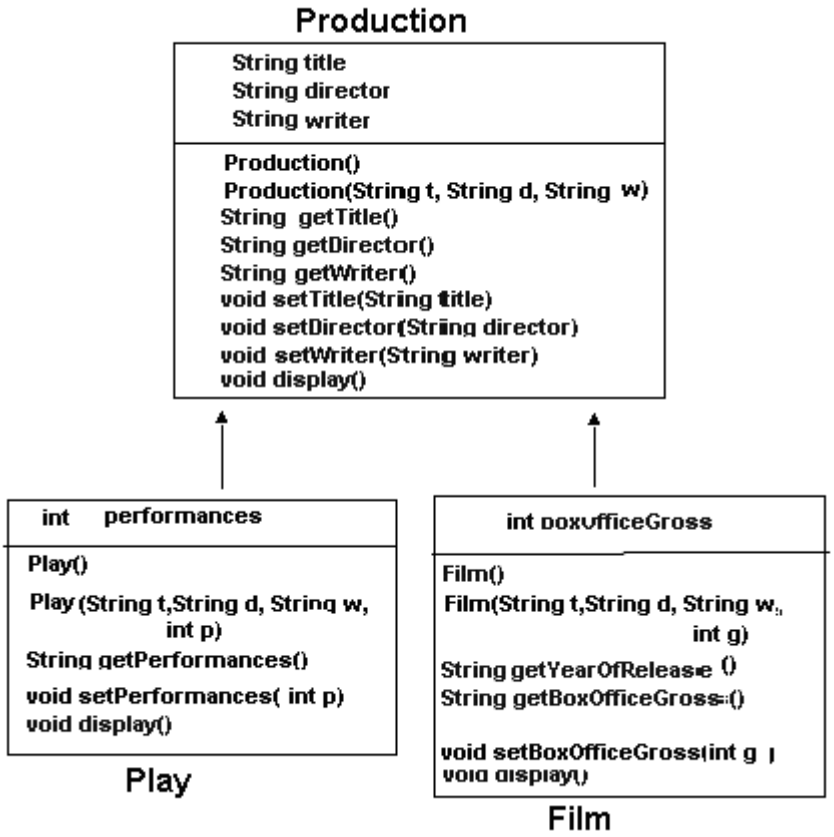
A *Play* class **and** **a *Film* class**

Look! The Play and the Film Class have much in common.

We can factor out the commonality of the two classes and make a new class ***Production*** which will serve as a base class or parent class to Play and Film.

A Play *is-a* Production

A Film *is-a* Production



Play extends Production; Film extends Production

```
1.  public class Production
2.  {
3.      protected String title;
4.      protected String director;
5.      protected String writer;
6.      public Production() // default constructor
7.      {
8.          title= "";
9.          director = "";
10.         writer = "";
11.     }

12.     public Production(String t, String d, String w)  // three argument constructor
13.     {
14.         title= t;
15.         director = d;
16.         writer = w;
17.     }

18.     public String getTitle()
19.     {
20.         return title;
21.     }

22.     public String getDirector()
23.     {
24.         return  director;
```

```
25.     }

26.     public String getWriter()
27.     {
28.        return writer;
29.     }

30.     public void setTitle(String t)
31.     {
32.         title = t;
33.     }

34.     public void setDirector(String d)
35.     {
36.         director = d;
37.     }

38.     public void setWriter(String w)
39.     {
40.         writer = w;
41.     }

42.     public void display()
43.     {
44.         System.out.println("Production class");
45.     }
46. }

47. public class Play extends Production
48. {
49.     protected int performances;
50.     public Play()
51.     {
52.         super();                          //  call Production default constructor
53.         performances = 0;
54.     }

55.     public Play(String t, String d, String w, int p)
56.     {
57.         super(t,d,w);                 // call Production  constructor
58.         performances = p;
59.     }

60.     public int getPerformances()
61.     {
62.         return performances;
63.     }

64.     public void setPerformances(int p)
65.     {
66.         performances = p;
67.     }

68.     public void display()
69.     {
70.         System.out.println("Title:        "+ title);
71.         System.out.println("Director:     "+ director);
72.         System.out.println("Playwright: "+ writer);
73.         System.out.println("Performances: " +    performances);
74.       }
75. }
```

```
76. public class Film extends Production
77. {
78.     protected int boxOfficeGross;
79.     public Film()
80.     {
81.         super();                    // call Production default constructor
82.         boxOfficeGross = 0;
83.     }

84.     public Film(String title, String director, String writer, int gross)
85.     {
86.         super(title, director,writer);          // call Production constructor
87.         boxOfficeGross = g;
88.     }

89.     public int getBoxOfficeGross()
90.     {
91.         return boxOfficeGross;
92.     }

93.     public void setBoxOfficeGross(int gross)
94.     {
95.         boxOfficeGross = gross;
96.     }

97.     public void display ()
98.     {
99.         System.out.println("Title:        "+ title);
100.        System.out.println("Director:     "+ director);
101.        System.out.println("Screenwriter:  "+  writer);
102.        System.out.println("Total gross:  $"+   boxOfficeGross+" million");
103.    }
104.}
```

```
1.  public class Entertainment
2.  {
3.      public static void main(String[] args)
4.      {
5.          Film film =      new Film("Titanic", "James Cameron",
                                  "James Cameron",2245);
6.          Play play =      new Play("Bus Stop","Harold Clurman","William Inge",478);

7.          film.display();
8.          System.out.println();
9.          play.display();
10.     }
11. }
```

Output

```
Title:        Titanic
Director:     JamesCameron
Screenwriter:  James Cameron
Total gross:  $2245 million

Title:        Bus Stop
Director:     Harold Clurman
Playwright: William Inge
Performances: 478
```

Abstract Classes

An abstract class is a class that cannot be instantiated. However, an abstract class can be inherited.

In general, an abstract class has the following properties:

- The keyword **abstract** denotes an abstract class. For example,

```
public abstract class Production
{
        // abstract class
}
```

  specifies that Production is an abstract class.

- An abstract class **cannot be instantiated**. You cannot create an object of an abstract class.
- An abstract class **can be inherited** by other classes. **An abstract class is designed for inheritance not instantiation.**
- An abstract class ***may*** contain abstract methods. An abstract method is a method with no implementation. For example, the method display in Production

```
Public abstract class Production
{
      // other stuff the same
      public abstract void display();  // method has no body
```

  is an abstract method. Notice the keyword abstract and the terminal semicolon.

- If an abstract class contains abstract methods, those methods ***must*** be overridden in any non-abstract subclass; otherwise the subclass is also abstract.
- All abstract classes and methods are public.
- To be of any use, an abstract class must be extended.

As an abstract class, Production has the following form:

```
public abstract class Production  // notice the keyword abstract
{
   // all attributes and methods, except display(), are as before

    public abstract void display(); // Look! No implementation
}
```

- Every non-abstract or concrete subclass that extends Production must implement the abstract method display().

- Any non-abstract subclass of Production is guaranteed to have a display() method. That's the contract.

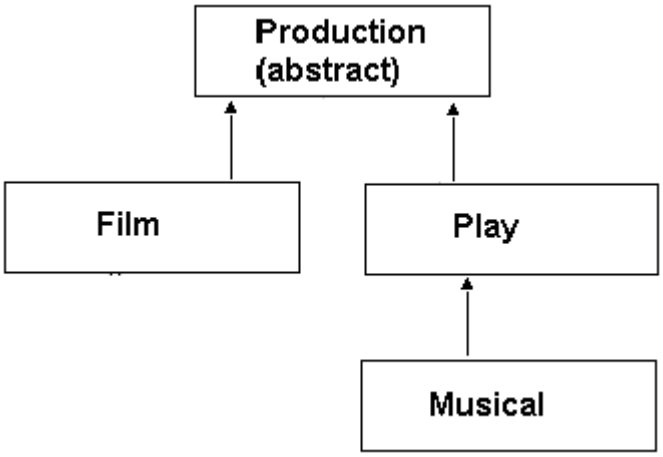
- A subclass that does not implement every abstract method of its parent class is also abstract and cannot be instantiated.

- Adhering to this rule, both Play and Film, being non-abstract subclasses of Production, implement display().

Extending the Hierarchy

```
1.  class Musical extends Play // has everything Play has and more
2.  {
3.      protected  String composer;
4.      protected String lyricist;

5.      public Musical()                    // default constructor
6.      {
7.          super();                        // invokes the default constructor of Play
8.          composer = "";
9.          lyricist = "";
10.     }

11.     public Musical(String t, String d, String w, String c, String l, int p)
12.      // t(itle), d(irector), w(riter), c(omposer), l(yricist), p(erformances)
13.     {
14.         super(t,d,w,p);                 // invokes the 4-argument constructor of Play
15.         composer = c;
16.         lyricist = l;
17.     }

18.     public String getComposer()
19.     {
20.         return composer;
21.     }

22.     public void setComposer(String c)
23.     {
24.         composer = c;
25.     }

26.     public String getLyricist()
27.     {
28.         return lyricist;
29.     }

30.     public void setLyricist(String l)
31.     {
32.         lyricist = l;
33.     }
```

```
34.     public void display()              // overrides the display() method of Play
35.     {
36.         super.display();                // call display of the parent class, Play
37.         System.out.println("Lyricist: "+ lyricist);
38.         System.out.println("Performances: " + performances);
39.     }
40. }
```



The *Production* hierarchy

Summary
Inheritance

Review of the basic ideas

If X is a base/parent Class and Y is a derived/child class, we call the relationship between X and Y an **is-a relationship**

Dog is-a Animal
Rectangle is-a Figure
Movie is-a Production

- **Protected variables** and methods are visible and accessible to a class's subclasses

- A child class inherits each public and protected method of a parent class *unless* the subclass provides its own implementation. That is unless the child class overrides the mehtod

- A subclass does ***not*** **inherit the constructors** of the base class. To invoke the constructors of the base class, a subclass uses the keyword super.
  A call to the parent class' default constructor is
  super()

- If a constructor of a derived class calls a superclass constructor, the call must be made before any other code is executed in the constructor of the derived class.

- An **abstract class** is a class that cannot be instantiated
  public abstract class Dumb.

- An abstract class may contain abstract methods. An abstract has no implementation.
  public abstract void aMethod(); --- no code in the method

- A class that inherits from an abstract class is required to override and implement all the abstract class's methods, otherwise the inherited class is also abstract.

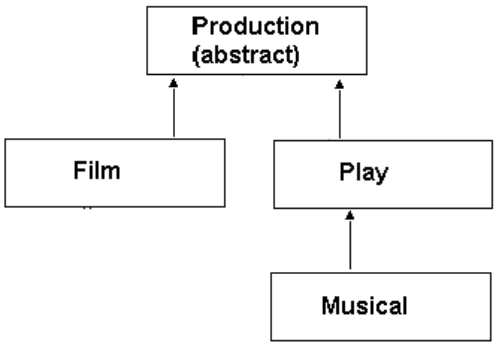
- If a child class does not call a parent constructor , then an implicit call is made to the default constructor of the parent class. **It is always good practice to define a default constructor in any base class.**

UPCASTING

Objects of a derived/child class are also objects of the base/parent class.
For example,

Production f = new Film(.....) // A Film is-a production
Cat c = new Leopard(....) // A Leopard is-a Leopard

Upcasting means casting an object to a parent or more general type.



A Production reference may **point** to a (a) Film object (b) a Play object or (c) a Musical object, even though Production is abstract.

So this is legal:

```
Production p = new Film(...)  // p is  a Production reference that references a Film
Production p = New Musical(....)
Play play = new Musical // Play is a parent of Musical
```

In general, Objects of a derived (child) type can be considered objects of the base (parent) type

A parent can refer to a child.

```
Cat c = new Leopard() // OK
Leopard l = new Cat() // error  Every Cat IS NOT a Leopard
```

Example

```
Production[]  p = new Production[3];  // an array of three Production references
p[0] = new Film(...); // Film is-a Production..upcasting
p[1] = new Play(...);
p[2] = new Musical(...);
```

Note: p is an array of Production REFERENCES. No Production objects have been created using "new." Production is abstract and cannot be instantiated.

Example

(1) Play play = new Musical(...);
(2) Musical m = play; // child referring to a parent

(1) is OK. A musical is-a Play. A play reference can refer to a Musical
(2) is ILLEGAL --> m is a Musical reference; p lay is a Play reference
a child cannot explicitly refer to (point to) a parent
Every Play is NOT a Musical

DOWNCASTING

Downcasting means casting an object to a derived, child or more specialized type.

Example:

(1) Production p = new Film();
(2) p.getWriter();
(3)p.getBoxOfficeGross();

(1) Legal -- a Parent (Production) can refer to a child(Film).
A Film is-a Production
To the Java compiler p is a Production reference.
Production is the declared type of p.

But, a Film object was actually created.
We might say that Film is the real type of p.

(2) Legal -- The compiler sees p as a Production reference and
Production objects have a getWriter() method. No problem.

(3) ILLEGAL -- The compiler sees p as a Production reference
and Production objects do NOT have getBoxOfficeGross() methods.
Here you need a DOWNCAST

((Film)p).getBoxOfficeGross()

Using a downcast tell the compile not to fret the real type of p is Film. That is, a Film object was actually created and p has all the features of Film. The compiler does not know this and is informed with a downcast.

Example:
(1) Play p = new Musical(...); //OK
(2) String name = p. getComposer(); //Error

(2) is an error because, to the Java compiler p refers to a Play object (the declared type).
That is all the java compiler looks at--what has been declared. And, the Play class does not have a getComposer() method. The java compiler does not see that a Musical object (with a getComposer() was actually created.

This can be fixed with an explicit downcast informing the compiler that p is really a musical:
(1) Play p = new Musical(...); //OK
(2) String name = ((**Musical**)p). getComposer(); //OK with downcast

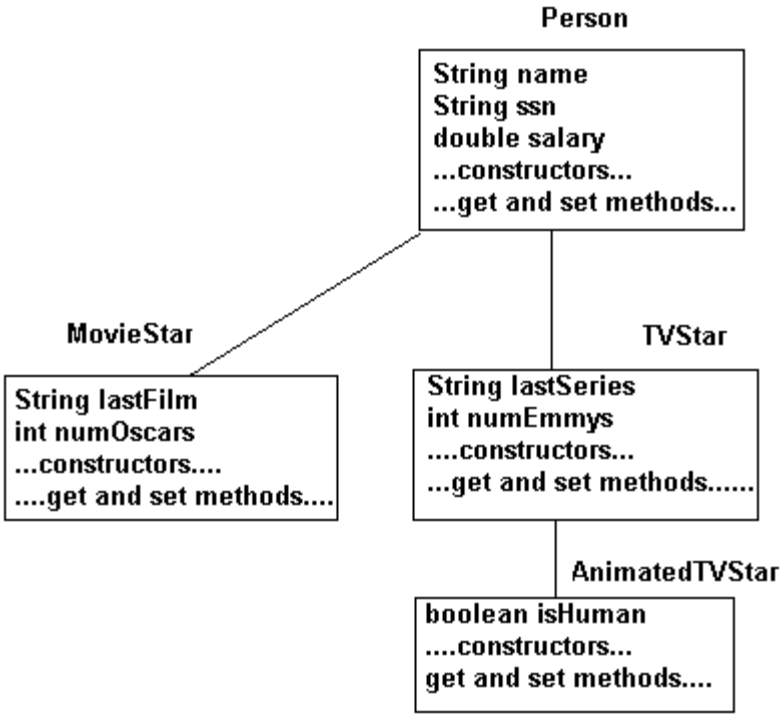
Example:
(1) Play play = new Musical (..); // upcast is ok
(2) Musical m = play; // Error

In (2) m is a Musical. A Musical cannot refer to a Play, which is what Play has been declared. Child does not refer to a parent.

But in fact, play has been created as a Musical, so a downcast will work:

(1) Play play = new Musical (..); // upcast is ok
(2) Musical m = (Musical)play; // OK

The downcast tells the compiler 'Don't be upset, all is OK>"
Again, the compiler looks at what has been declared, not what has been created or instantiated.


Person
String name
String ssn
double salary
...constructors...
...get and set methods...
MovieStar
String lastFilm
int numOscars
...constructors....
....get and set methods....
TVStar
String lastSeries
int numEmmys
....constructors...
...get and set methods.....
AnimatedTVStar
boolean isHuman
....constructors...
get and set methods....

```
public class Person
{
 protected String name;
 protected double salary;

 public Person()
 {
  name = "";
  salary = 0.0;
 }
  public Person(String name, double salary)
 {
  this.name= name;
  this.salary = salary ;
 }

  public String getName()
  {
   return name;
  }

  public double getSalary()
  {
   return salary;
  }

  public void setName(String name)
  {
     this.name = name ;
  }

  public void setSalary(double salary)
  {
   this.salary = salary;
  }
}
```

```
public class  MovieStar extends Person
{
  protected int numOscars;
  protected String lastFilm;

  public MovieStar()
  {
    super();
    numOscars = 0;
  }
   public MovieStar(String name,  double salary, String lastFilm, int numOscars)
  {
    super(name,salary);
    this.numOscars = numOscars;
    this.lastFilm = lastFilm;
  }

  public int getNumOscars()
  {
    return numOscars;
  }

  public void setNumOscars(int n)
  {
    numOscars = n;
  }
   public String getLastFilm()
  {
    return lastFilm;
  }

  public void setLastFilm(String lastFilm)
  {
    this.lastFilm = lastFilm;
  }
}
```

```
public class  TVStar extends Person
{
 protected int numEmmys;
 protected String lastSeries;
 public TVStar()
 {
  super();
  numEmmys = 0;
  lastSeries = "";
 }
  public TVStar(String name, double salary,  String lastSeries, int numEmmys)
 {
  super(name,salary);
  this.numEmmys = numEmmys;
  this.lastSeries = lastSeries;
 }

 public int getNumEmmys()
 {
  return numEmmys;
 }

 public void setNumEmmys(int n)
 {
  numEmmys = n;
 }

 public String getLastSeries()
 {
  return lastSeries;
 }

 public void setLastSeries(String lastSeries)
 {
  this.lastSeries = lastSeries;
 }
}
```

```
public class  AnimatedTVStar extends TVStar
{
  protected boolean isHuman;
  public AnimatedTVStar()
  {
    super();
    isHuman = false;  //Homer Simpson is human; Sponge Bob  Squarepants is not
  }
   public AnimatedTVStar(String name,  double salary, int numEmmys, String lastSeries,
                                  boolean isHuman)
  {
    super(name, salary, lastSeries, numEmmys);
    this.isHuman = isHuman;
  }

  public boolean getIsHuman()
  {
    return isHuman;
  }

  public void setIsHuman(boolean isHuman)
  {
    this.isHuman = isHuman;;
  }


}
```

```
public class TestPerson
{
  public static void main(String[] args)
  {
    Person[] people = new Person[3];
    people[0] = new MovieStar("Jennifer Lawrence",  10000000.00, "American Hustle", 1);
    people[1] = new TVStar("Jim Parsons",  2000000.00, "The Big Bang Theory", 3);
    people[2] = new AnimatedTVStar("Homer Simpson", 40000, 0, "The Simpsons",true);

    System.out.println( people[0].getName()+ "'s  last film  is " +
                                                          ((MovieStar)people[0]).getLastFilm());

    System.out.println( people[1].getName()+ " has won  " +
                                              ((TVStar)people[1]).getNumEmmys()   + " Emmy Awards");
  }
}

      //Notice where we had to downcast
      // To the compiler people is an array of Person--- and that is all the compiler knows
      //It must be told that p[0] is a MovieStar if p[0] calls any method not in Person
      //That is what the downcast does
```

The instanceof operator
Syntax
  boolean object instanceof class

instanceof is a boolean operator like <, ==, or >

Example:

Play p = new Musical

p instanceof Musical --> true
p instanceof Play --> true
p instanceof Production --true
p instanceof Film --> false

```
public abstract class Figure
{
  protected int length, width;
  public Figure()
  {
     length= width  = 1;
  }
  public Figure (int length, int width)
  {
    this.length = length;
    this.width = width;
  }

   public int getLength()
   {
     return length;
   }
   public void setLength(int length, int width)
   {
      this.length = length;
      this.width = width;
  }
  public abstract int area();
  }

 public  class Rectangle extends Figure
{
   public Rectangle()
   {
      super();
   }
  public Rectangle(int length, int width)
  {
   super(length, width);
  }


  public int area()
  {
    return length*width;
  }
```

```
   public int perimeter()
   {
     return 2*(length+ width);
  }
}

class Cube extends Figure
{
  protected  int height;
  public Cube()
  {
     super();
    height = 1;
  }
   public Cube(int length, int width, int height)
   {
     super(length, width);
    this.height = height;
  }

  public int area()
   {
    return 2*length*width + 2*height*length + 2*height*width;
  }
  public int volume()
   {
    return length*width*height;
  }
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
public class Figures
{
 public static void calculate(Figure x)
 {

  if (x instanceof Rectangle)
       System.out.println("Perimeter is "+ ((Rectangle)x).perimeter() );
   else if (x instanceof Cube)
       System.out.println("Volume is "+((Cube)x).volume());
   else
   System.out.println("No calculation performed");
 }

  public static void main(String args[])
  {
  Figure[] figures = new Figure[3];
  figures[0] = new Rectangle();
  figures[1] = new Cube(2,3,4);
  figures[2] = new Rectangle(2,4);

  for (int i = 0; i < 3; i++)
    calculate(figures[i]);
 }
}
```