Class 9 Notes

Review:

The concepts from the last 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.

  - 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**
  - Downcasting means casting an object to a derived, child or more specialized type.
    - Example:
    - (1) Production p = new Film();
    - (2) p.getWriter();
    - (3)((Film)p).getBoxOfficeGross();

  - Declared type vs real type
    - Animal rex = new Dog()
    - Declared type of rex is Animal
    - Real type of rex is Dog
    - The compiler looks at the declared type

Here is an example of an abstract class:

```
public abstract  class Animal
{
     protected String name;
     public Animal(String name)
    {
         this.name = name;
    }
   public void myName()
   {
     System.out.println("My name is "+ name);
  }
 public abstract void speak();
}
```

```
public  class Dog extends
Animal
{
    public Dog( String name)
   {
       super(name);
   }

  public void speak(
  {
   System.out.println(“Woof”);
   }

   public void eat()
   {
     System.out.println("Chomp");
   }
}
```

```
public  class Cat extends
Animal
{
 public Cat( String name)
{
     super(name);
}

public void speak()
 {

System.out.println("Meow");
 }

public void eat()
{
  System.out.println("Slurp ");
 } }
```

```
public  class Bird extends
Animal
{
 public Bird( String name)
 {
       super(name);
 }

public void speak()
{
  System.out.println("Chirp ");
}

public void eat()
{
   System.out.println("Peep ");
 }
}
```

```
public  class Puppy extends Dog
{
    public Puppy( String name)
    {
       super(name);
    }
    public void speak()
    {
       System.out.println("Squeak ");
    }
}
```

```
public class TestAnimal1
{
   public static void main(String[] args)
   {
     // Where do we need to downcast?


      Animal fido = new Dog("Fido");              //  upcast Dog is-a Animal
      Dog prince = new Dog("Prince");
      Animal scamp = new Puppy("Scamp");    // upcast Puppy is-a Animal
      Dog bingo = new Puppy("Bingo");          // upcast Puppy is-a Dog
      Puppy sparky = new Puppy("Sparky");

      // where are the errors?????

      fido.speak();
      fido.eat();

      prince.speak();
      prince.eat();

      scamp.eat();
      scamp.speak();

      bingo.eat();
      bingo.speak();

      sparky.speak();
      sparky.eat();

   }
}
```

```
public class TestAnimal2
{
   public static void main(String[] args)
   {
     // Will this compile?

      Animal fido = new Dog("Fido");       //  upcast Dog is-a Animal
      Dog prince = new Dog("Prince");
      Animal scamp = new Puppy("Scamp");     // Puppy is-a Animal
      Dog bingo = new Puppy("Bingo");
      Puppy sparky = new Puppy("Sparky");


      fido.speak();
      ((Dog)fido).eat();

      prince.speak();
      prince.eat();

      ((Puppy)scamp).eat();
      scamp.speak();

      bingo.eat();
      bingo.speak();

      sparky.speak();
      sparky.eat();

   }
}
```

```
public class TestAnimal3
// is this OK?
{
    public static void main(String[] args)
    {
        Animal[] animals = new Animal[4];
        animals[0] = new Dog("Fido");  //  upcast Dog is-a Animal
        animals[1]= new Puppy("Bingo"); // Puppy is-a Animal
        animals[2] = new Bird("Tweety");
        animals[3] = new Bird("Felix");
        Puppy sparky = new Puppy("Sparky");

        for (int i = 0; i < 4; i++)
        {
          animals[i].myName();
          animals[i].speak();
          animal[s].eat();
        }


    }
}
```

The instanceof operator

Syntax

    boolean object instanceof class

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

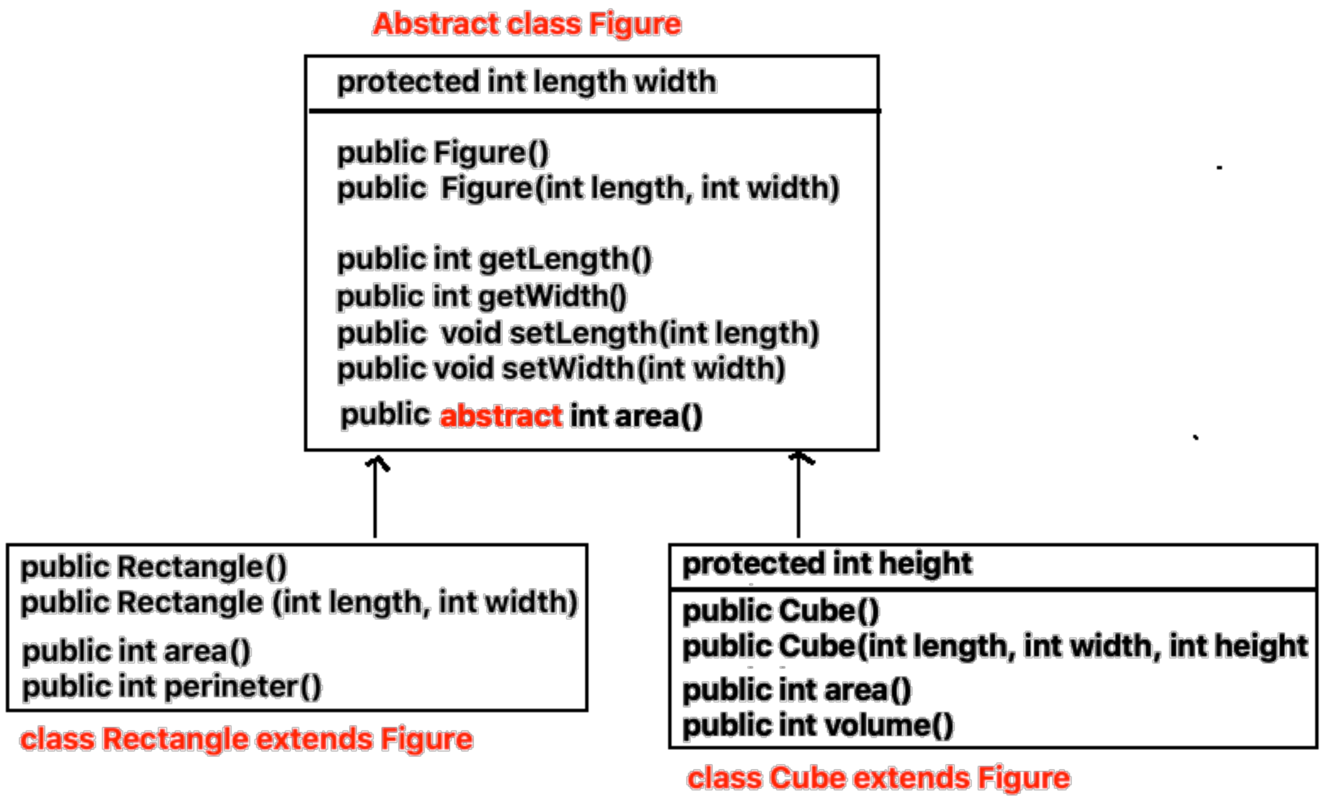
Example: Using the Production hierarchy

Play p = new Musical

p instanceof Musical *returns* true // notice the format **object instanceof class**
p instanceof Play --> true
p instanceof Production --true
p instanceof Film --> false

Example

Here is a simple hierarchy:


Abstract class Figure
protected int length width
public Figure()
public Figure(int length, int width)
public int getLength()
public int getWidth()
public void setLength(int length)
public void setWidth(int width)
public abstract int area()
public Rectangle()
public Rectangle (int length, int width)
public int area()
public int perineter()
class Rectangle extends Figure
protected int height
public Cube()
public Cube(int length, int width, int height
public int area()
public int volume()
class Cube extends Figure

This class uses the instanceof operator:

```
public class Figures
{
 public static void calculate(Figure x)  // we can pass in any object in the hierarchy
 {

            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(1,1);
   figures[1] = new Cube(2,3,4);
   figures[2] = new Rectangle(2,4);

   // Notice the declared type of each object  is Figure
   for (int i = 0; i < 3; i++)
     calculate(figures[i]);
 }
}
```

Notice the downcast in the calculate method

Output:

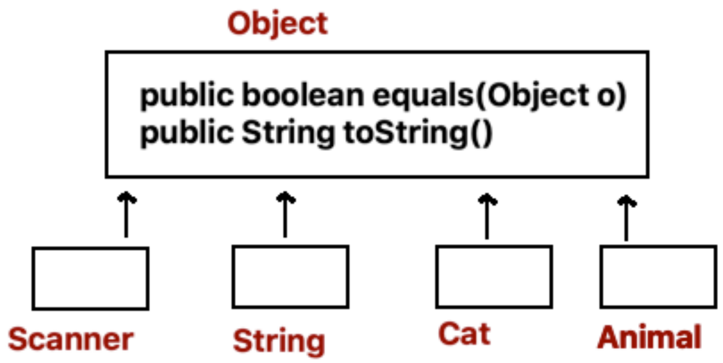
```
Perimeter is 4
Volume is 24
Perimeter is 12
```

The Object class -- with uppercase "O"

Object is a class provided by Java.

EVERY class is a subclass/child of Object and every class inherits the methods of Object. These are:

1. boolean equals (Object o)
2. String toString()



Since Object is the parent of all classes, every class can be upcast to Object

Object s = new String("Hello"); // Every String is-a Object

However, s.charAt(2) is illegal.
Why?
Because the declared type of s is Object and the compiler looks at Object for charAt(..).

So you would need a downcast: ((String)s).charAt(2)
Here is another example:
Object[] x= new Object[3];
X[0] = new Dog("Fido", 3,"woof");
X[1] =new String("hello");
X[2] = new Rectangle(2,5);

But this is not particularly useful because the declare type of each of these is simply Object.

So what good is the Object class? How do we use the Object class?

EVERY class inherits
    boolean equals (Object o)
from Object
But this **equals(Object o) compares references**

Usually a class overrides the equals it inherits from Object.
String does this.

It is common to override equals(Object o) in the classes that we make.

Example:

```
public class Rectangle
{
   Private int length, width;
   // constructors
   // other methods

   public boolean equals (Object o)
   {
      return this.length == ((Rectangle)o).length &&  // notice the downcast
               this.width == ((Rectangle)o).width;
    }
}
```

So two Rectangle objects are equal if they have the same length and width

```
Rectangle x = new Rectangle (3,5);
Rectangle y = new Rectangle (3,5);
Rectangle z = new Rectangle (4,6);
```

x.equals(y) returns true but x.equals(z) returns false

Example

```
public class Leopard extends Cat
{
    protected in numspots;
    // other code
    public  boolean equals(Object o)
    {
         Return yhis.numSpots == ((Leopard)o).numSpots);
     }
}
```

String toString()

Every class inherits

```
                    Public String toString()
```

From Object

Every class we have written has a toString() methods

```
Film f = new Film(.....);
String s = f.toString()
System.out.println(s)

Dog rex = new Dog(....)
System.out.println(rex.toString())
```

toString() has been hiding in all our classes but as it stands it is not very useful

For example:

```
public class Square
{
  public int side;
  public Square()
  {
    side = 1;
  }
  public Square(int s)
  {
    side = s;
  }

  public int area()
  {
    return side * side;
  }

public static void main(String [] args)
{
  Square s = new Square(5);
  System.out.println("toString() returns : "+ s.toString());
 }
}
```

The output is toString() returns : Square@56f2c96c
The toString() method give the name of the class and a memory location
This is not particularly informative .
So to be of any use, we override toString()

Example:

```
public class Square
{
  public int side;
  public Square()
  {
    side = 1;
  }
  public Square(int s)
  {
    side = s;
  }

  public int area()
  {
    return side * side;
  }

  public String toString()
  {
    return "The side of the square is "+side+
            " and the area is "+ area();
   }

public static void main(String [] args)
{
  Square s = new Square(5);
  System.out.println( s.toString());
 }
}
```

Now the output is

```
The side of the square is 5 and the area is 25
```

Shortcut and convenience:

If x is an object then

```
System.out.println(x)
```

Is the same as

```
System.out.println(x.toString())
```

When you pass an object to println or print toString() is automatically called

Example: What is the output?

```
public class Pupil
{
	private String name;
	private double gpa;
	private String idNumber;

	 public Pupil()
	{
		 name = "";
		gpa = 0.0;
		 idNumber  = "";
	}

	 public Pupil(String name, double gpa, String id)
	 {
		 this.name = name;
		 this.gpa = gpa;
		 idNumber = id;
	 }
	//getters and setters go here

	public boolean equals(Pupil o)
	{
		return (this.idNumber).equals(o.idNumber); // uses String equals
	 }

	 public String toString()
	 {
		 return name + " "+ idNumber + "  " + gpa;
	}

	 public static void main(String[] args)
	{
		 Pupil bart = new Pupil("Bart Simpson", 1.3, "12345");
		 Pupil anotherBart = new Pupil("Bart ", 1.9, "12345");
		 Pupil lisa = new Pupil("Lisa Simpson", 3.8, "54321");

		 System.out.println(bart); //calls bart.toString()
		 System.out.println(lisa);  // notice I did not need toString()
		 System.out.println(bart.equals(lisa));
		 System.out.println(bart.equals(anotherBart));
	}
}
```

One last note about equals()

Why bother overriding the equals of Object when a class can just have it own equals

The following class has its own equals method. So that equals is **overloaded** -- There are two versions

1. The equals inherited from Object: boolean equals(Object o)
2. The equals defined in Person : boolean equals(Person p)

```
public class Person
{
  private String last;
  private String first;

  public Person()
  {
    first = "";
    last = "";
  }

  public Person(String first, String last)
  {
    this.first = first;
    this.last = last;
  }

  public boolean equals(Person p)  // note the parameter is Person
  {
     return first.equals(p.first) && last.equals(p.last);
  }

  public static void main(String [] args)
  {
     Person p = new Person("Sheldon", "Cooper");
     Object q = new Person("Sheldon", "Cooper");
     System.out.println(p.equals(q));
  }
}
```

The output is false.

Person has two versions of equals(..): the one written in the program

equals(Person p)

and the one inherited.

Equals (Object o)

The parameter q has declared type Object so the inherited one is chosen. Probably not what you want

Here the equals from Object is overridden. There is just ONE version of equals.
This is how it should be done.

```
public class Person1
{
  private String last;
  private String first;

  public Person1()
  {
    first = "";
    last = "";
  }

  public Person1(String first, String last)
  {
    this.first = first;
    this.last = last;

  }

  public boolean equals(Object p)  // note the parameter is Object
  {
     return first.equals(((Person1)p).first) && last.equals(((Person1)p).last);
  }

  public static void main(String [] args)
  {
     Person1 p = new Person1("Sheldon", "Cooper");
     Object q = new Person1("Sheldon", "Cooper");
     System.out.println(p.equals(q));
  }
}
  Output is true
```