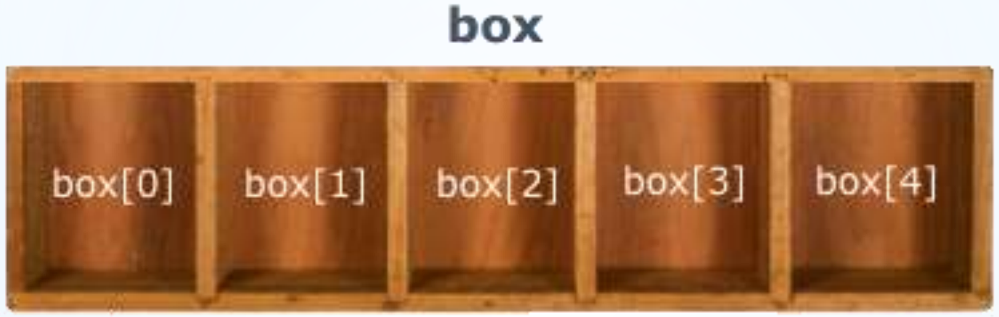
Programming 1

Further Java:

Lecture #2: Arrays and Inheritance

Arrays – a recap

- An array is a special type of variable in that it can contain many values
- If a standard variable is like a box, think of an array as being like a box with compartments:



- One of these "compartments" is more correctly referred to as an *element* of the array
  - Each element has a unique number (or *index*)
  - In most programming languages element indexes start at 0

Arrays in Java

- Arrays store a set of objects in elements
  - Arrays in Java are actual *objects*
  - Arrays can contain any type of element value (primitive or objects) but a single array must contain elements of the same type
    - (although you could have an array of `Object`)
- To declare an array:
  - declare an array variable
  - create an array object and assign it to this variable
  - store things in the array elements

Arrays

- To declare an array variable:

```
// int array
int[] banana;
```

- This declares an array of integers, `arr`:
  - Note that the number of integers in the array is not specified at this stage.

Arrays

- To Create an Array Object
  - Use the **new** operator

```
int[] arr;
arr =  new int[3];
or
int[] arr = new int[3];
```

create array of 3 ints:

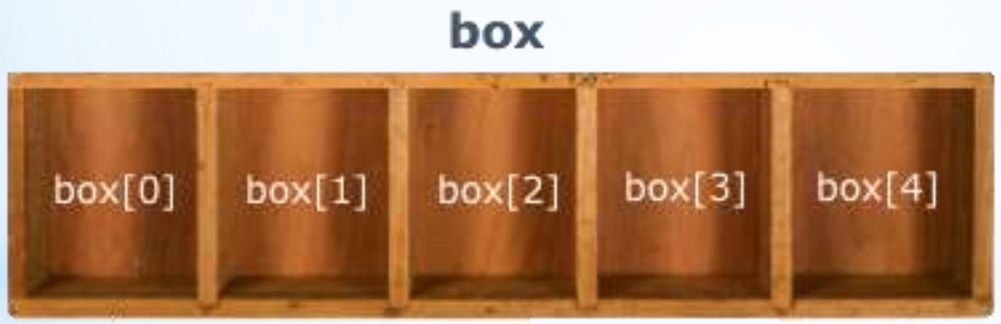
arr[0], arr[1], arr[2]

Can be combined in one statement

Arrays in “boxspeak”

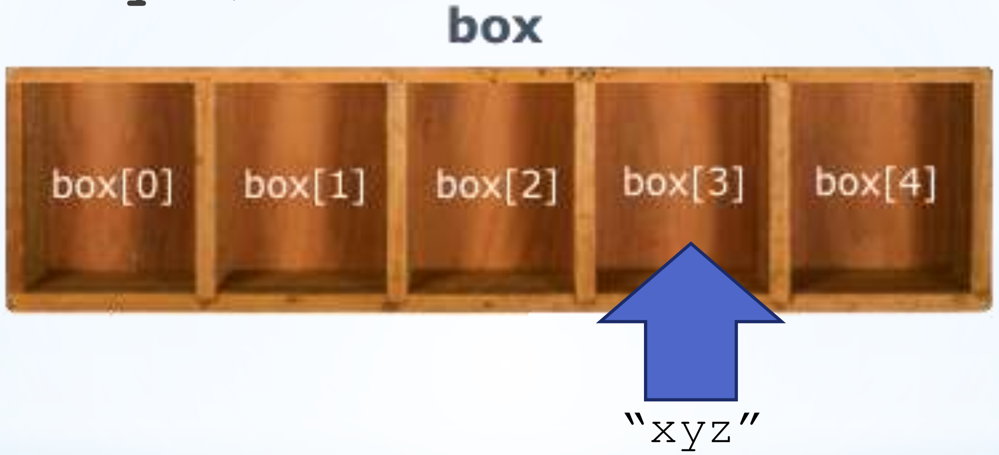
- If a variable is like a box, then an array is like a box with numbered compartments…

```
String[] box = new String[5];
```



Putting things into the boxes

- Place elements into a “compartment” of the array by specifying the compartment number:
  - `box[3] = "xyz";`



- Until you assign something to an array element, it will contain the default value for that data type or class
  - Primitive data types have default values – google "default primitive values in java"
  - Arrays of objects (i.e. that have a type that is a class) contain a default value of null

Putting things into the boxes

- You can combine declaration, creation and initialisation in one statement:

```
// 3 ints
int[] arr = {15,3,56};

// 3 strings
String[] strs = {“Paul”,”Fred”,”Bill”};
```

Looping (iterating) through the elements of an array

- You can get the length of an array with `.length`
  - `int[] arr = {1,6,8,24};`
- **`arr.length`** would be 4

- We could use that in a **`for`** loop:

```
for (int i = 0; i < arr.length; i++)
{
   System.out.println(arr[i]);
}
```

Interating through an array with a for/each loop

- Or, we can use an alternative loop construct called a for/each loop:
  - Verbal equivalent: **`For each element in an array`**
- Syntax example (assuming the array arr from the previous slide)

```
 for (int single : arr)
{
  System.out.println(single);
}
```

- The loop iterates once for each element in `arr`
- The element is copied into a variable (`single` in this case)
- Then we can do something with `single`

Arrays of objects

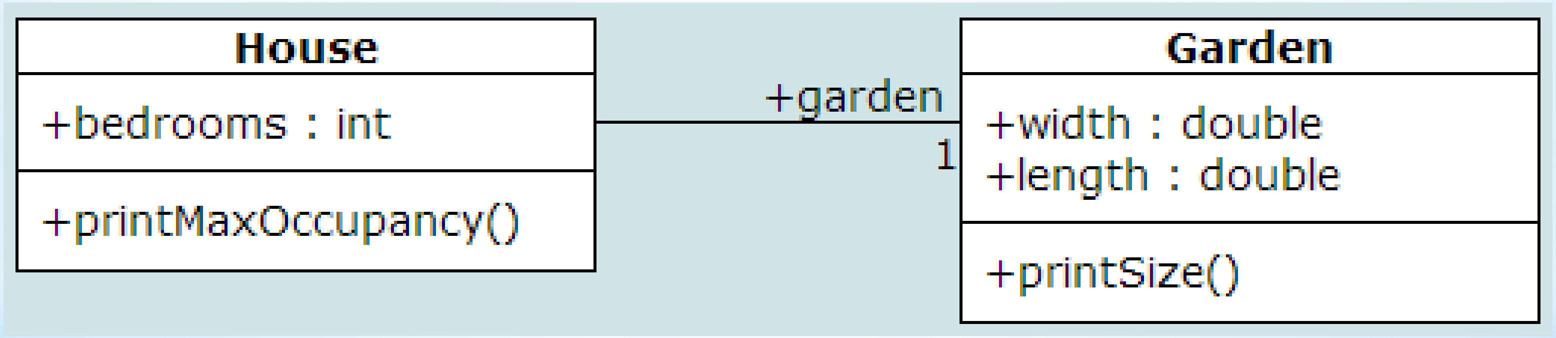
- We can have an array of objects

```
Ball[] ballsOnASnookerTable = new Ball[22];
ballsOnASnookerTable[0] = new Ball();
ballsOnASnookerTable[0].setColour(“white”);
```

…and so on…!

Arrays to represent a “has-a” relationship

- Remember the House “has a” Garden exercise a couple of weeks ago?

- 

- We could make the `garden` attribute in house an array of type `Garden`
- A house could then have several gardens…

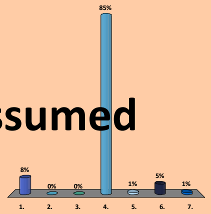
Arrays to represent a “has-a” relationship

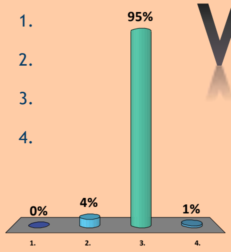
```
public class House
{
    public int bedrooms;
    public Garden[] garden;

   // …rest of class…
}
```

Given the two classes below, which statement is most correct?

1. There is no relationship between Foo and Bar
2. Foo has one single Bar
3. Bar has one single Foo
4. Foo has one or multiple Bars
5. Bar has one or multiple Foos
6. Both 4 and 5
7. Paul has found yet another innovative way to cock up an orange slide

```
public class Foo
{
    private int bar;
    private Bar[] moo;

    // getters and setters assumed
}

public class Bar
{
    private String[] foo;
    private int moo;

    // getters and setters assumed
}
```





Which of the below code excerpts most appropriately represents the assertion *Flibble has many Wibbles*?

1

```
public class Wibble
{
    private int bibble;
    private Flibble[] nibble;

    // getters and setters assumed
}
```

3

```
public class Flibble
{
    private int bar;
    private Wibble[] nibble;

    // getters and setters assumed
}
```

2

```
public class Wibble
{
    private int bar;
    private Flibble nibble;

    // getters and setters assumed
}
```

4

```
public class Flibble
{
    private int bar;
    private Wibble nibble;

    // getters and setters assumed
}
```

Inheritance

What is inheritance?

- Inheritance is an OO technique that lets you use an existing class as the basis for a new class
  - The existing class is called the base class, **superclass** or parent class
  - The new class is called the **subclass** or **child class**
- The subclass *inherits* all the methods and attributes of the superclass

Consider our balls… (ooer)

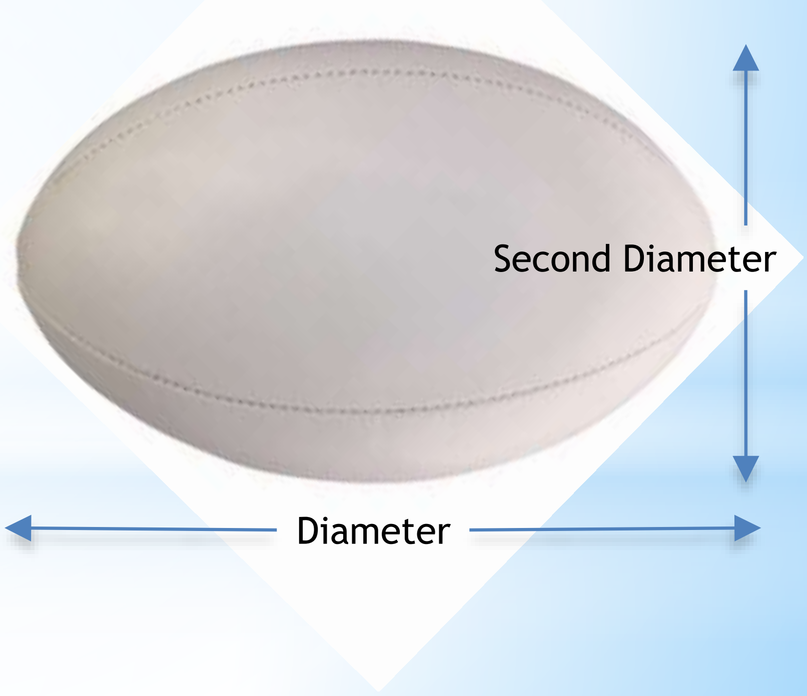
- We can have different *instances* of ball
  - They all have a colour
  - They all have one diameter

- But what about if they aren't *quite* the same as every other ball?

Dealing with funny shaped balls

- A rugby ball has the same attributes as a "normal" ball
  - It has a colour
  - It has a diameter
- However, it has a SECOND, additional diameter attribute
  - ...which we'll call `secondDiameter`

Colour: **white**
Diameter: **27cm**
Second Diameter: **19cm**



Extending our balls

- We can use inheritance to create a new class, OvalBall
- OvalBall will *extend* Ball
  - This means it will *inherit* all of its existing attributes and methods
- We can then add additional attributes and methods to the subclass OvalBall that get added to the ones that have been inherited

```
public class Ball
{
  private double diameter;
  private String colour;

  public void setDiameter(double d)
  {
     this.diameter = d;
  }

  // assume other methods follow
  // including constructors
  // and getter/setters...
}
```

```
public class OvalBall extends Ball
{
  private double secondDiameter;

  public void setSecondDiameter(double sd)
  {
    this.secondDiameter = sd;
  }

  // …rest of class methods etc here
}

public class Main
{
  public static void main(String[] a)
  {
    OvalBall rugby = new OvalBall();
    rugby.setDiameter(27);
    rugby.setSecondDiameter(19);
  }
}
```

```
public class Ball
{
  private double diameter;
  private String colour;

  public void setDiameter(double d)
  {
     this.diameter = d;
  }

  // assume other methods follow
  // including constructors
  // and getter/setters...
}
```

`OvalBall` *inherits* `setDiameter` from `Ball`

```
public class OvalBall extends Ball
{
  private double secondDiameter;

  public void setSecondDiameter(double sd)
  {
    this.secondDiameter = sd;
  }

  // …rest of class methods etc here
}

public class Main
{
  public static void main(String[] a)
  {
    OvalBall rugby = new OvalBall();
    rugby.setDiameter(27);
    rugby.setSecondDiameter(19);
  }
}
```

```
public class Ball
{
  private double diameter;
  private String colour;

  public void setDiameter(double d)
  {
     this.diameter = d;
  }

  // assume other methods follow
  // including constructors
  // and getter/setters...
}
```

`OvalBall`'s own attributes and methods are combined with those from the superclass, `Ball`

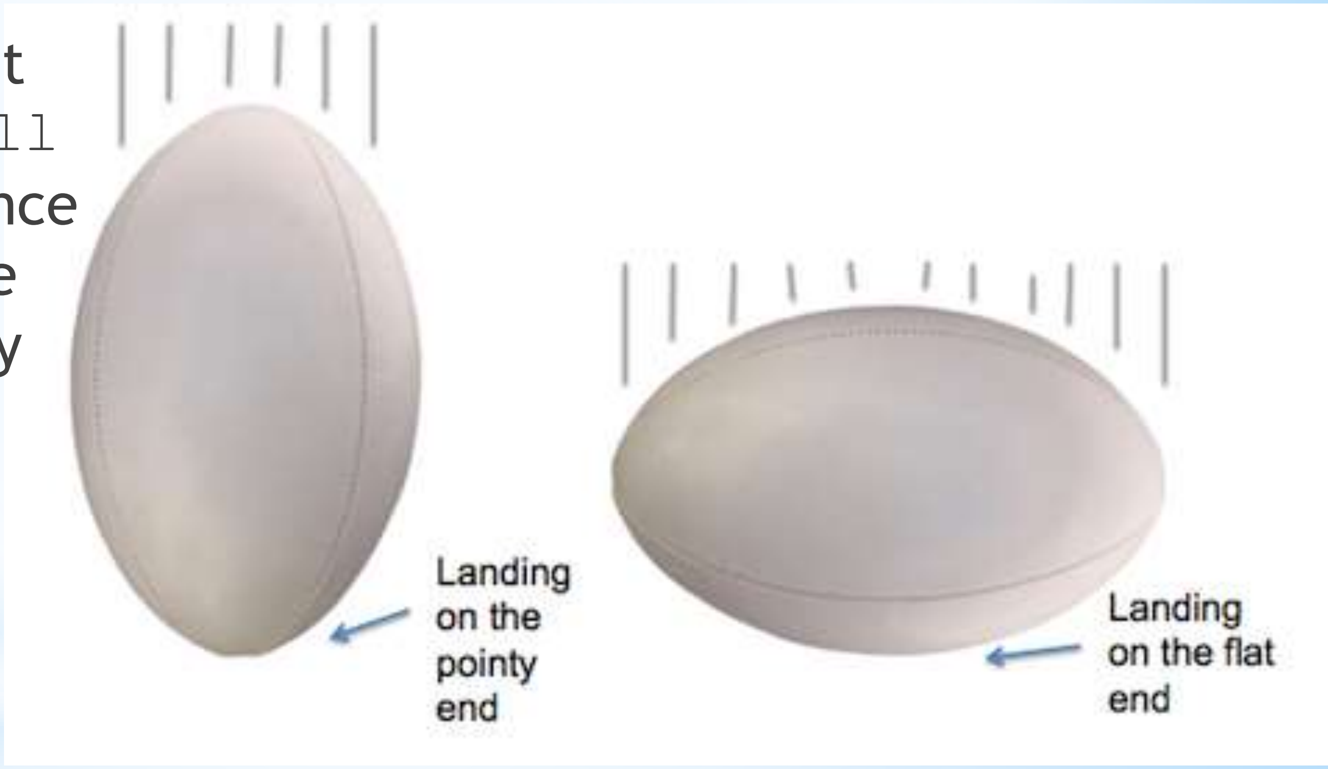
```
public class OvalBall extends Ball
{
  private double secondDiameter;

  public void setSecondDiameter(double sd)
  {
    this.secondDiameter = sd;
  }

  // …rest of class methods etc here
}

public class Main
{
  public static void main(String[] a)
  {
    OvalBall rugby = new OvalBall();
    rugby.setDiameter(27);
    rugby.setSecondDiameter(19);
  }
}
```

Method overriding

- We can replace an existing method from the superclass with a new or altered version in the subclass
- Consider our `OvalBall`
  - Ball has a method bounce that returns the height a ball bounces based on its diameter
  - …when an `OvalBall` bounces, would it bounce the same way as a “normal” ball would?

Method overriding

- Let's say for the sake of argument that an `OvalBall` has a 50/50 chance of landing on the flat or the pointy end…



Method overriding

- …if it lands on the pointy end, it should use `diameter` to calculate the bounce height
- …if it lands on the flat end, it should use `secondDiameter` to calculate the bounce height

Method overriding

- We can supply a version of bounce in `OvalBall` that will *override* the one from the superclass

```
public double bounce()
{
  if (Math.random() > 0.5)
  {
    return this.getDiameter() * 2;
  }
  else
  {
    return this.getSecondDiameter() * 2;
  }
}
```

super

- You can use `super` to refer to the superclass from the subclass
- So, we might write our bounce method in OvalBall like this

```
public double bounce
{
  if (Math.random() > 0.5)
  {
    return super.bounce();
  }
  else
  {
    return this.getSecondDiameter() * 2;
  }
}
```

super

- You can use `super` to refer to the superclass from the subclass
- So, we might write our bounce method in OvalBall like this

```
public double bounce
{
  if (Math.random() > 0.5)
  {
    return super.bounce();
  }
  else
  {
    return this.getSecondDiameter() * 2;
  }
}
```

Subclass and superclass type compatibility

- A superclass will “fit” into a subclass, e.g.
  - `Ball rugbyBall = new OvalBall();`
- …but not the other way round…
  - `OvalBall rugbyBall = new Ball();`

Subclass and superclass type compatibility

- But we could do

```
Ball firstBall = new Ball();
Ball secondBall = new Ball();
Ball thirdBall = new OvalBall();
// below will call bounce from Ball
secondBall.bounce();
// below will call bounce from OvalBall
thirdBall.bounce();
```

- This is called dynamic **polymorphism**

Constructors and inheritance

- If your superclass does not have the default (parameterless) constructor, then you MUST have a constructor in your subclass that uses super

Inheritance and constructors

```
public class Ball
{
  private double diameter;
  private String colour;

  public Ball(double d, String c)
  {
     this.diameter = d;
     this.colour = c;
  }

  // assume other methods follow
  // including constructors
  // and getter/setters...
}
```

```
public class OvalBall extends Ball
{
  private double secondDiameter;

  public void setSecondDiameter(double sd)
  {
    this.secondDiameter = sd;
  }

  // …rest of class methods etc here
}
```

Inheritance and constructors

```
[illegible]lic class Ball
{
  privat[illegible]ble diameter;
  private Str[illegible]lour;

  public Bal[illegible] c)
  {
     this.diameter = d;
     this.colour = c;
  }

  // assume other [illegible]fo[illegible]
  // includin[illegible]structors
  // and[illegible]er/setters...
}
```

```
public class OvalBall extends B[illegible]
{
  private double sec[illegible]ameter;

  [illegible]ub[illegible]ic [illegible]et[illegible]eco[illegible]dDi[illegible]meter(double sd)

     this.secondDiameter = sd;

     …re[illegible]t[illegible]s methods etc here
}
```

WOULD NOT COMPILE

Inheritance and constructors

```
public class Ball
{
  private double diameter;
  private String colour;

  public Ball(double d, String c)
  {
     this.diameter = d;
     this.colour = c;
  }

  // assume other methods follow
  // including constructors
  // and getter/setters...
}
```

```
public class OvalBall extends Ball
{
  private double secondDiameter;

  public OvalBall(double d, String c, double sd)
  {
    super(d,c);
    this.secondDiameter = sd;
  }

  public void setSecondDiameter(double sd)
  {
    this.secondDiameter = sd;
  }

  // …rest of class methods etc here
}
```

Inheritance and constructors

```
public class Ball
{
  private double diameter;
  private String colour;

  public Ball(double d, String c)
  {
     this.diameter = d;
     this.colour = c;
  }

  // assume other methods follow
  // including constructors
  // and getter/setters...
}
```

```
public class OvalBall extends Ball
{
  private double secondDiameter;

  public OvalBall(double d, String c, double sd)
  {
    super(d,c);
    this.secondDiameter = sd;
  }

  public void setSecondDiameter(double sd)
  {
    this.secondDiameter = sd;
  }

  // …rest of class methods etc here
}
```

Call the superclass's constructor with these parameters

Inheritance and constructors

```
public class Ball
{
  private double diameter;
  private String colour;

  public Ball(double d, String c)
  {
     this.diameter = d;
     this.colour = c;
  }

  // assume other methods follow
  // including constructors
  // and getter/setters...
}
```

```
public class OvalBall extends Ball
{
  private double secondDiameter;

  public OvalBall(double d, String c, double sd)
  {
    super(d,c);
    this.secondDiameter = sd;

  public void setSecondDiameter(double sd)
  {
    this.secondDiameter = sd;
  }

  // …rest of class methods etc here
}
```

The superclass's constructor only knows about and sets diameter and colour…

Inheritance and constructors

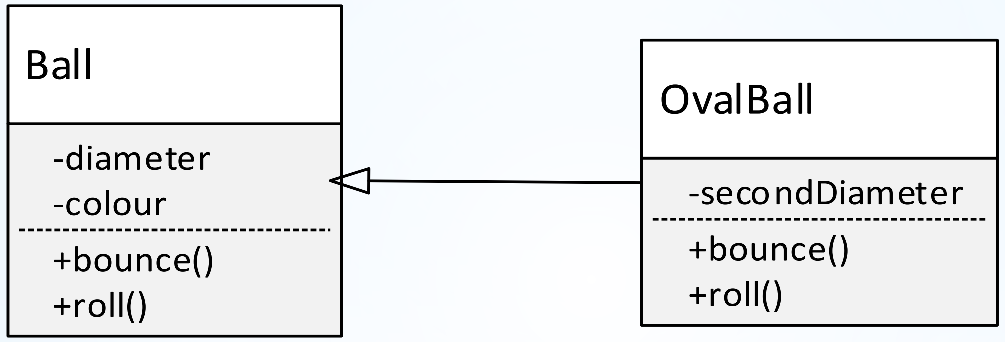
```
public class Ball
{
  private double diameter;
  private String colour;

  public Ball(double d, String c)
  {
     this.diameter = d;
     this.colour = c;
  }

  // assume other methods follow
  // including constructors
  // and getter/setters...
}
```

```
public class OvalBall extends Ball
{
  private double secondDiameter;

  public OvalBall(double d, String c, double sd)
  {
    super(d,c);
    this.secondDiameter = sd;
  }

  pu
  {
    this.secondDiameter = sd;
  }

  // …rest of class methods etc here
}
```

…so after the call to the superclass's constructor, we return to the subclass's constructor and set the parameter(s) that are specific to the subclass

UML Class Diagram notation for inheritance

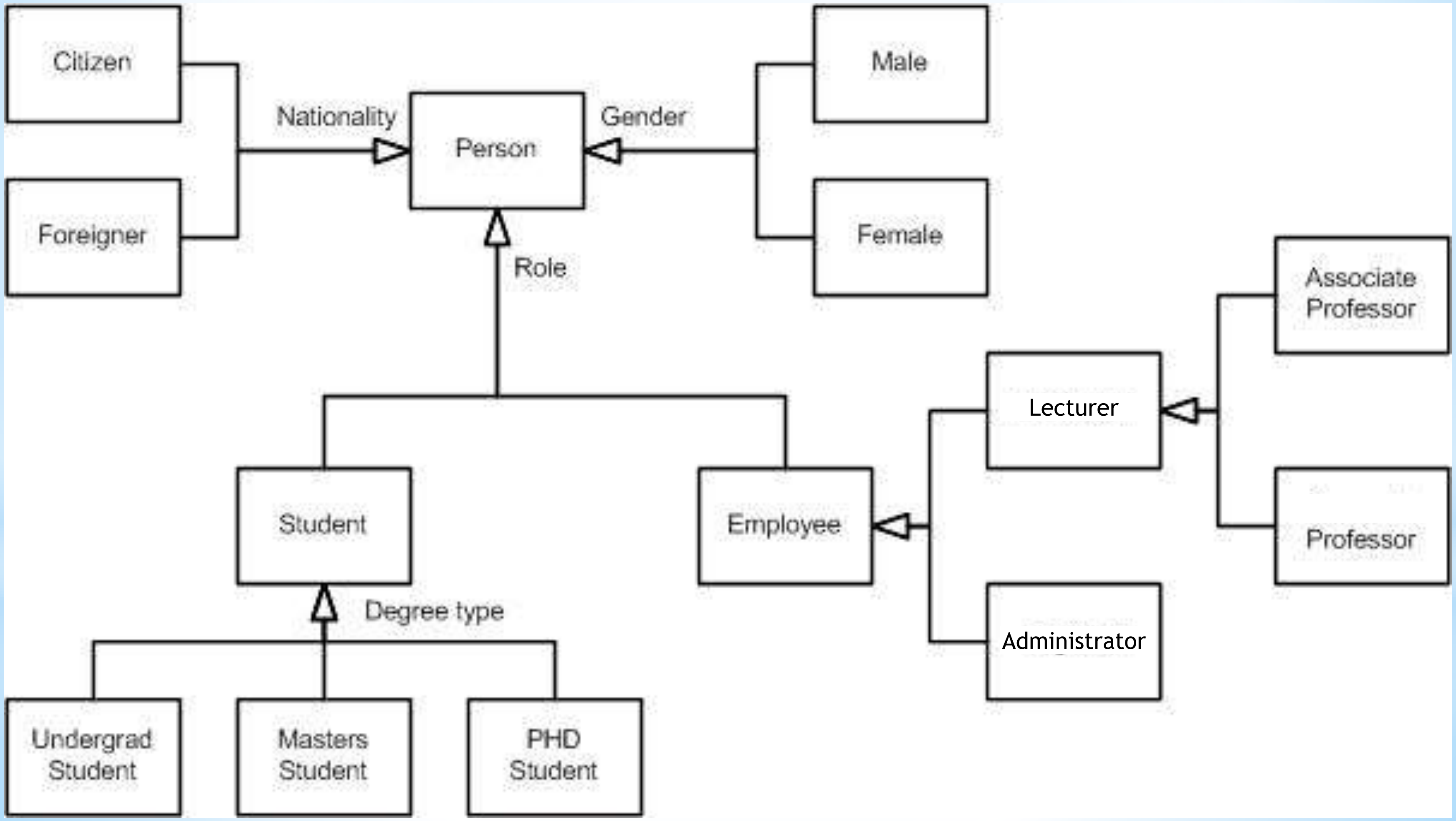
- A triangular arrow signifies that the relationship between two classes involves inheritance



- The triangle goes at the superclass end
- We also refer to this as an “is a” relationship
  - i.e. OvallBall “is a” Ball

A more complex example

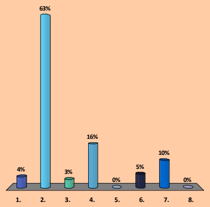
- Consider a university system…



Given the two classes below, which statement is most correct?

1. There is no relationship between Googoo and Gaga
2. Googoo is a Gaga
3. Googoo has one single Gaga
4. Googoo has many Gagas
5. Gaga has one single Googoo
6. Gaga has many Googoos
7. Gaga is a Googoo
8. Paul has found yet another innovative way to cock up an orange slide

```
public class Googoo extends Gaga
{
   private int meep;
   private String[] gaga;

   // getters and setters assumed
}

public class Gaga
{
   private String mope;
   private int googoo;

   // getters and setters assumed
}
```



Summary

- If a variable is like a box, then an array is like a box with compartments
- Declare an array with

  ```
  int[] arr = new int[3];
  ```

  or

  ```
  int[] arr = {1,7,9,2,5,20};
  ```
- Get the length of an array with `.length`
- You can iterate (loop) through an array with the for/each loop
- Attributes in a class can be of type array
  - ..this is one way to represent the a "has-a" relationship when there are several of the same objects involved in the attribute, e.g.
    - A university has many students - **`Student[] students`**
    - A house has several gardens - **`Garden[] gardens`**

Summary

- **Inheritance** lets us create new classes by using an existing one as a base
  - The existing class that we use is called the **superclass**
  - The new class we create (using the superclass as a base) is called the **subclass**
  - We use the Java keyword `extends` to specify that a new class uses an existing one as its superclass
  - The subclass *inherits* all of the methods and attributes of the superclass
  - If we add any new methods or attributes in the subclass, these are added alongside to the inherited ones from the superclass

Summary

- You can replace an existing superclass method in the subclass
  - This is called **overriding** a method
- You can call methods in the superclass from the subclass with `super`
- If you have constructors in your superclass, and one of them is not the default (parameterless) constructor..
  - any subclasses MUST include a constructor that calls the superclass's constructor via `super`