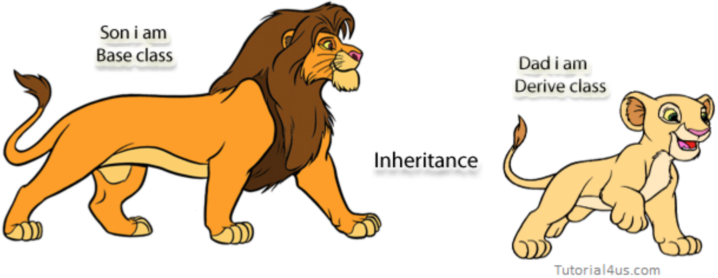
Software Construction
Inheritance

Dhammika Elkaduwe

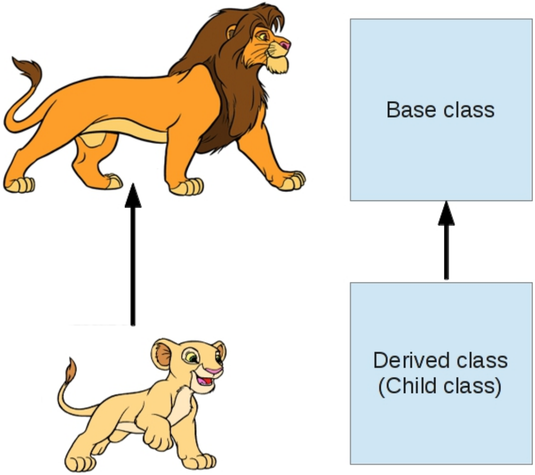
Department of Computer Engineering
Faculty of Engineering

University of Peradeniya

Inheritance: in nutshell



Inheritance: Notation



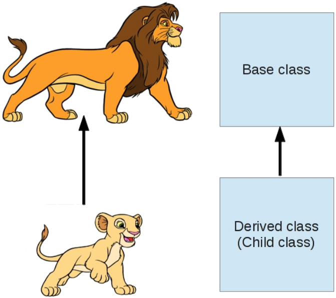
Base class

Derived class
(Child class)

Basic idea

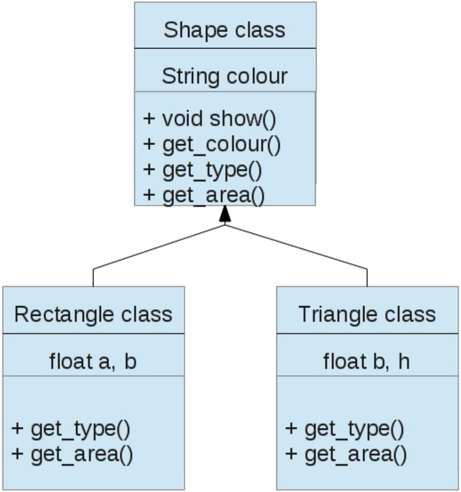
“class B is similar to A but with some differences”

- Why should we implement the same functionality again?
- class A is called the base class and B is the child class (or derived class).
- In Java A is called the **supperclass** and B is called the **subclass**.



Example1: Shapes class

We are interested in the colour, type and area of a shape.



Example1: Implementation

```java
public class Shapes {
    private String colour;
    public Shapes(String colour) { this.colour = colour; }

    public void show() {
      System.out.println(get_colour()+ " " +
                        get_type() + " which has area of " +
                        get_area());
    }

    public String get_colour() { return this.colour; }

    /* not the best way. Interface is better */
    public String get_type() { return "Unknown"; }
    public float get_area() { return 0f; }
}
```

Example1: Implementation
Rectangle class

```
public final class Rectangle extends Shapes {
  /* You cannot extend Rectangle since it is final */
```

- *extends* key words is used to derive the *Rectangle* class from *Shapes* class.
- So, now I have all the fields/methods of *Shapes* class.
- Then we can change just the fields/methods we want.
- if a class is declared as *final* you cannot *extend* that (i.e. cannot inherit)

Example1: Implementation
Rectangle class

```
   private float a, b; // additional to shapes

    public Rectangle(String colour, float a, float b) {
        super(colour); /* call the constructor from super class
* has to be done first */
        this.a = a;
        this.b = b;
    }
```

- the *float a, b* are in addition to the field in the *Shapes* class.
- *super(colour)* calls the constructor from the parent (super) class.
- You have to call the super class constructor first.

Example1: Implementation
Rectangle class

```java
private float a, b; // additional to shapes
// constructor taken out.
@Override
public String get_type() { return "Rectangle"; }
@Override
public float get_area() { return a * b; }
```

- You can change the behaviour of required functions by *overriding* them.
- When one calls the methods *get_type()* or *get_area()* on a *Rectangle* object you will invoke the above implementation (as opposed to the implementation in the base class)
- If there is no suitable methods in the derived class you will search in the base class (example: calling *show()* on a *Rectangle* object.

Example1: Implementation
Rectangle class

```
    private float a, b; // additional to shapes
    @Override
    public String get_type() { return "Rectangle"; }
```

Additional notes:

- The *@Override* is called **annotations**.
- Java annotation gives the compiler *meta* information about the function.
- So, here we are saying the “following function will override”.
- If it does not compiler will complain.
- Examples: @Author(name = ...), @SuppressWarnings(“unchecked”), ..

Example1: invoking method from parent class

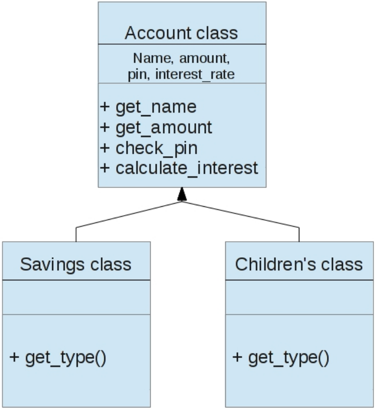
```
public class Shapes {
    // .. more code here
    public void show() {
          System.out.println(get_colour()+ " " +
               get_type() + " which has area of " + get_area());
    }
```

```
Rectangle r = new Rectangle("Red", 2f, 3f);
r.show();
```

- The *show* implementation is provided by the parent class.
- (even from this function) when *get_area()* is called, you will end up in overridden function in the *Rectangle* class.

Example2: Accounts class

Model the accounts in a bank.



- Most of the functionality is the same
- Different accounts have different interest rates
- When creating an account you can give different information (some have a pin, some do not etc.)

Example2: Overload constructor

At the time of creating the object you might have different information. Example:

- Set the PIN number later on
- Set the PIN number at the time of creating etc.

```
public Account (String name, int pin, float amount, float ir) {
     this.name = name; this.pin = pin;
     this.amount = amount; this.interest_rate = ir;
  }
  // overloading the constructor
  public Account(String n, float a, float ir){
    this(n, 0, a, ir);
  }
  public Account(String n, int p, float a) {
    this(n, p, a, common_ir);
  }
```

Example2: Checking the PIN

- PIN is only used for *Savings* accounts.
- Children accounts has no PIN (set of 0 at the time of creating)

```
public boolean check_pin(int pin) {
   if(pin == 0) return false;
   return this.pin == pin;
}
```

Should this function be overridden?

Example2: Checking the PIN

In some cases, you want to make sure that:

- parent class provides the functionality
- derived class uses it, but cannot modify it
- done using the *final* key word.

```
public final boolean check_pin(int
    pin) {
  if(pin == 0) return false;
  return this.pin == pin;
}
```

```
public final class
    Childrens extends
    Account {
....
```

Notes:

- if a function is *final* you cannot override
- if a class is *final* you cannot inherit

Example2: Childrens class

Children's accounts have a guardian.

```java
public final class Childrens extends Account {
    private String guardian;
    public Childrens(String child, String guadian, float
        amount) {
      // call constructor from super class
      // you HAVE TO call the super class first
      super(child, amount, interest_rate);
      this.guardian = guadian;
}
```

- Call the constructor of the parent class (where most of the information is kept).
- (You need to call the parent constructor first)
- In the parent class, if we do not provide a PIN it will be set to zero.
- Then we set the guardian.

Example2: Childrens class

Display the data in the account:

- The parent's *show* method does most of the work.
- Except, display the guardian's name.
- Solution: call the parent's method and do the rest of the work in the overriding function.

```java
public final class Childrens extends Account {
....
    @Override
    public void show() {
        super.show(); // calling the show method of parent
        System.out.println("Guardian: " + get_guardian());
    }
}
```

Example2: Access restrictions

set_interest function (and few others) you do not want to expose to the world but only for derived classes.
protected access controller can be used for that.

```
protected void set_interest(float rate) {
  this.interest_rate = rate;
}
protected boolean set_pin(int newpin, int oldpin) {
    if(check_pin(oldpin)) {
        this.pin = newpin;
        return true;
    }
    return false;
 }
```

Summary of access modifiers

- **private**: only available for the methods in the class
- **protected**: only available for the methods in the class, derived classes (and package).
- **public**: any one can access

Working with types

```
public class Savings extends Account {
```

Basic idea: *Savings* is *Account* and more.
So *Savings* object can be stored in an *Account* variable.

```
   public static void main(String [] args) {
       Account [] accounts = new Account[3];
       accounts[0] = new Account("Dhammika Elkaduwe", 1234,
           10000f);
       accounts[1] = new Savings("Gihan Sandirigama", 4321,
           12323f);
       accounts[2] = new Childrens("Sam Samarasekara",
                            "Gihan Samarasekara", 100);
       for(int i=0; i<3; i++)
           accounts[i].show(); // call the appropriate show
               method.

   }
```

Improvements

```
public class Shapes {
   public String get_type() { return "Unknown"; } // not the
       best way
   public float get_area() { return 0f; } /* not the best way.
       */
```

```
public class Account {

  .....
   public float get_amount() { return this.amount;}
   public String get_type() { return "Unknown"; }
```

Issue: sub-classes will have different implementations for the function.
So what do you do in the parent class?

Abstract class and methods

```
abstract class Vehicle { // abstract class, no instance
   protected String reg_number;
   protected String reg_date;
   protected String owner;
   private static int vehicles = 0;

   public Vehicle(String r, String d, String o) {
       reg_number = r; reg_date = d; owner = o;
       vehicles++;
   }

   public abstract String type(); // need to provide
       implementation
   public static int number_of_vehicles() { return vehicles; }

}
```

Abstract class and methods

```java
abstract class Vehicle { // abstract class, no instance
    private static int vehicles = 0;

    public abstract String type(); // need to provide
        implementation
    public static int number_of_vehicles() { return vehicles; }

}
```

Notes:

- an *abstract* class cannot be instanced
- if the child class provides an implementation for the *abstract* method(s) you can instance the child.
- if child does not implement abstract method, then child is *abstract* as well.

Extending a abstract class

```
class Car extends Vehicle {
    private static int cars = 0;
    public Car(String reg, String regDate, String owner) {
        super(reg, regDate, owner);
        cars ++;
    }

    public String type() { return "Car"; }
    public static int number_of_cars() { return cars; }
}
```

The *Car* provides an implementation for *type()* so we can make a *Car* instance.

```
        Vehicle a;
        a = new Car("KX2121", "12/2/12","Sam");
        a.show();
```

Static fields in parent class

```
abstract class Vehicle { // abstract class, cannot make instance
    private static int vehicles = 0;

    public Vehicle(String reg, String date, String owner) {
        .....
        vehicles++;
    }

    public static int number_of_vehicles() { return vehicles; }
```

Notes:

- Only one copy for all child classes
- That copy is incremented when the constructor is called
- (parent class constructor is called from child)

Static fields in parent class: Example

```
Vehicle a;
//a = new Vehicle("KX2121", "12/2/12","Sam");
/* this will not work since Vehicle is abstract */
a = new Car("KX2121", "12/2/12","Sam");
a.show();
a = new Bike("1/1/10");
a.show();
System.out.println("We have " + Vehicle.number_of_vehicles() +
                 " vehicles");

System.out.println("We have " + Car.number_of_cars() +
                 " cars");
```

Exercise 1

Suppose you want to model a library. Each library book has an author, title and an ISBN number. Books belongs to two categories; lending and reference. Lending books can be taken out from the library for a period of 3weeks but a reference book cannot be taken out.

Create a suitable *object* to represent the library. You should be able to search the library for books using the title of the book.

Advance: Look at Java Collections. Can we use Paris?

ILOs:

- Inheritance
- Overriding
- Annotations
- Overloading constructor
- Calling constructor form parent class
- Calling methods from parent (constructor and other)
- *static* fields in parent class
- How the type system works with derived classes
- Key words: extends, @Override, final, protected, super, abstract