Implementing Inheritance

INHERITANCE

- Relationship between two classes which enables one class to acquire attributes and functions of another class
- IS-A type of relationship : Subclass IS-A type of Super Class

Super class

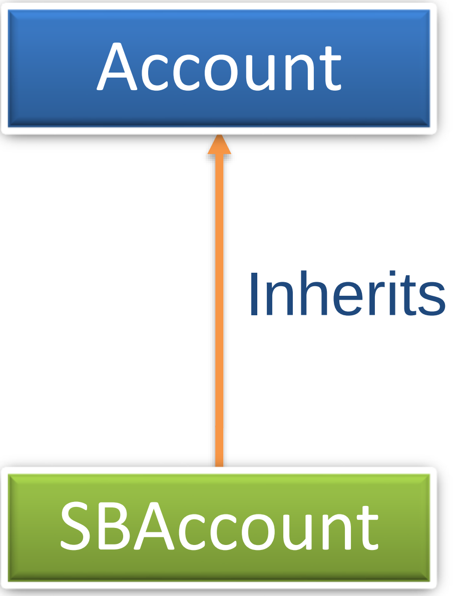
(parent/Base Class)

- Class which is inherited
- Contains the generalized/common attributes and functions of a group of classes

Sub class

(child /Derived Class)

- Class which is inheriting
- Acquires the accessible common attributes(variables) and functions(methods) of superclass through inheritance
- Also contains specific attributes and functions



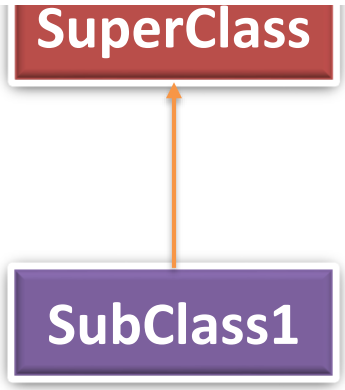
Account

Inherits

SBAccount

TYPES OF INHERITANCE

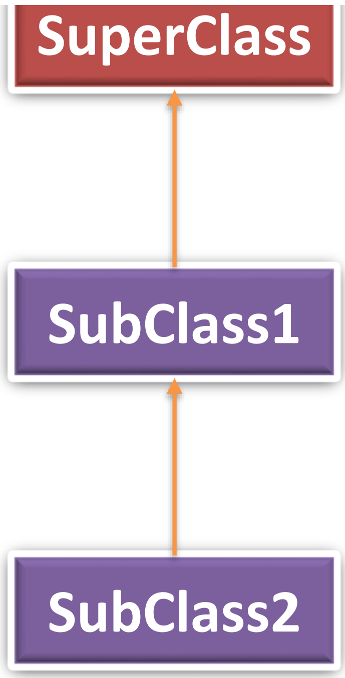
Single Inheritance



SuperClass

SubClass1

Multi -level Inheritance

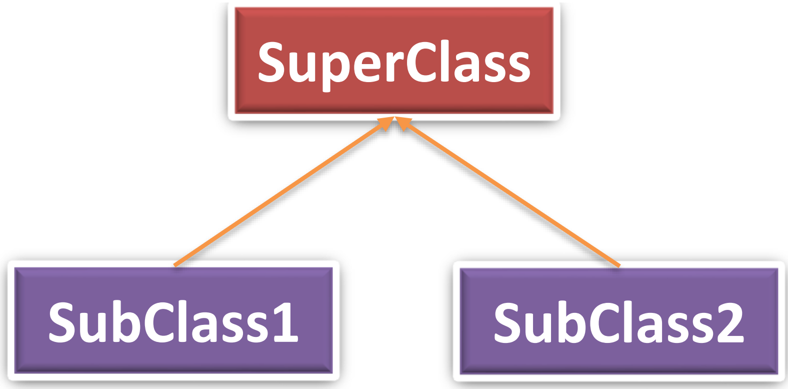


SuperClass

SubClass1

SubClass2

Hierarchical Inheritance

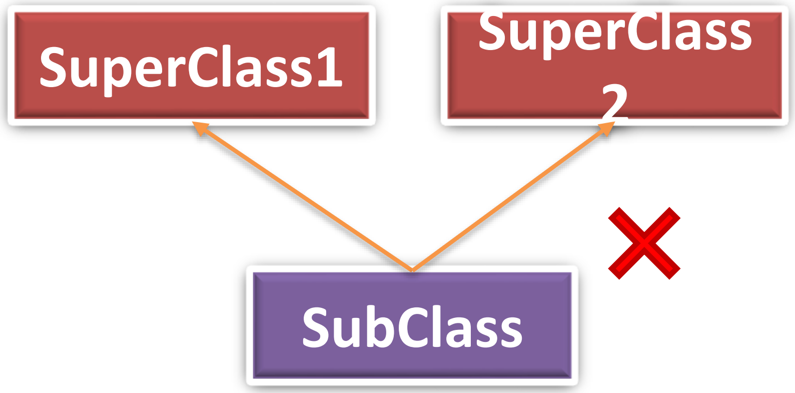


SuperClass

SubClass1

SubClass2

Multiple Inheritance



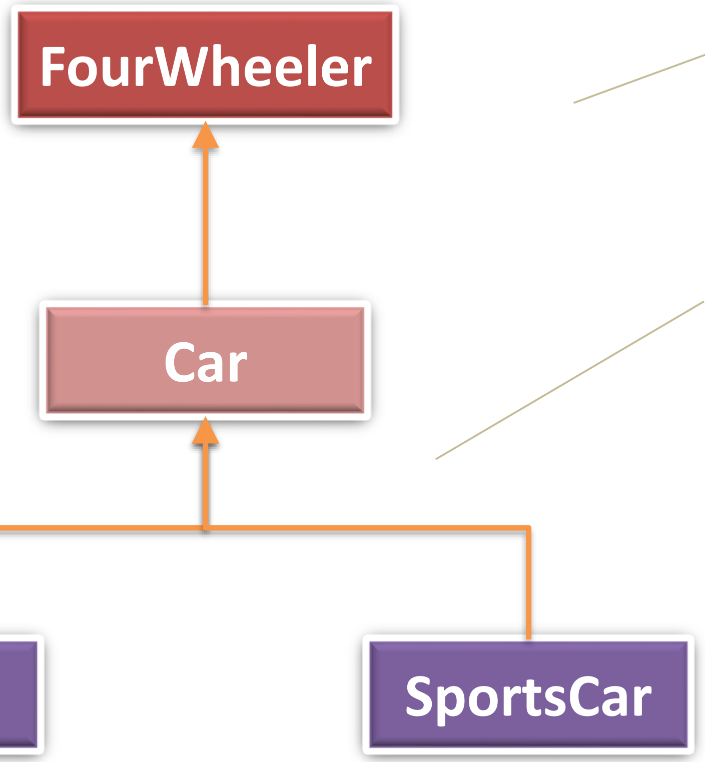
SuperClass1

SuperClass 2

SubClass

Java doesn’t Support through class

INHERITANCE HIERARCHY



Car **IS-A** type of FourWheeler

Sedan **IS-A** type of Car
Sedan **IS-A** type of FourWheeler

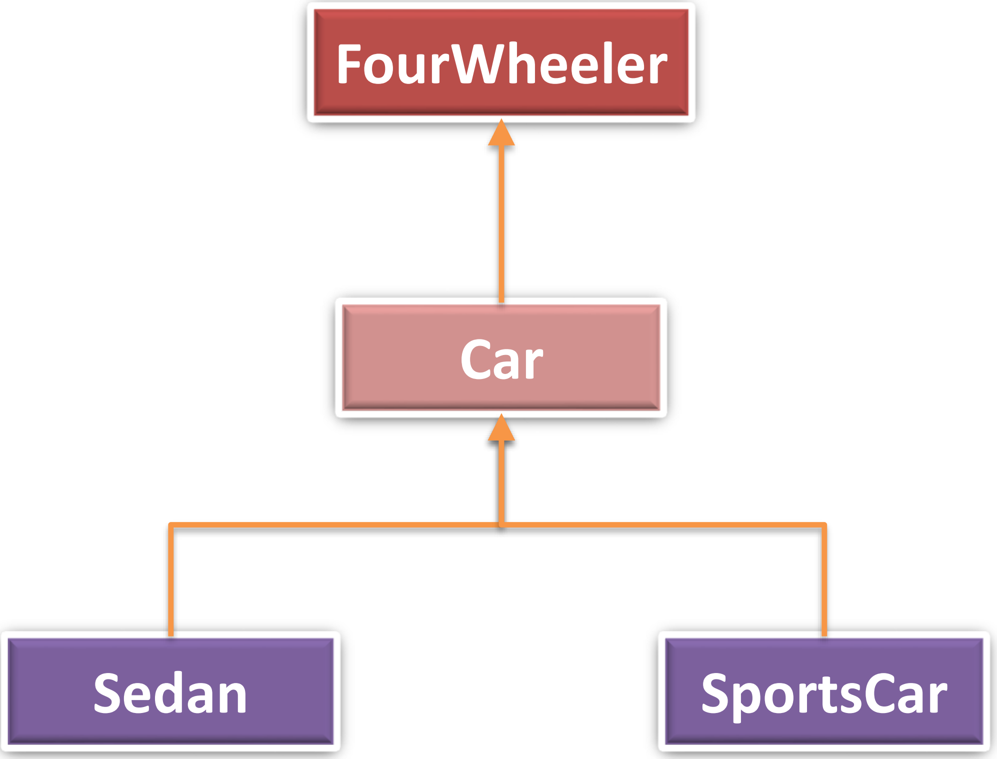
SuperClass of **Car** and its child classes

Car class inherits attributes and methods from **FourWheeler.**
Also a SuperClass of **Sedan** and **SportsCar**

Inherits attributes and methods from **FourWheeler** and **Car**

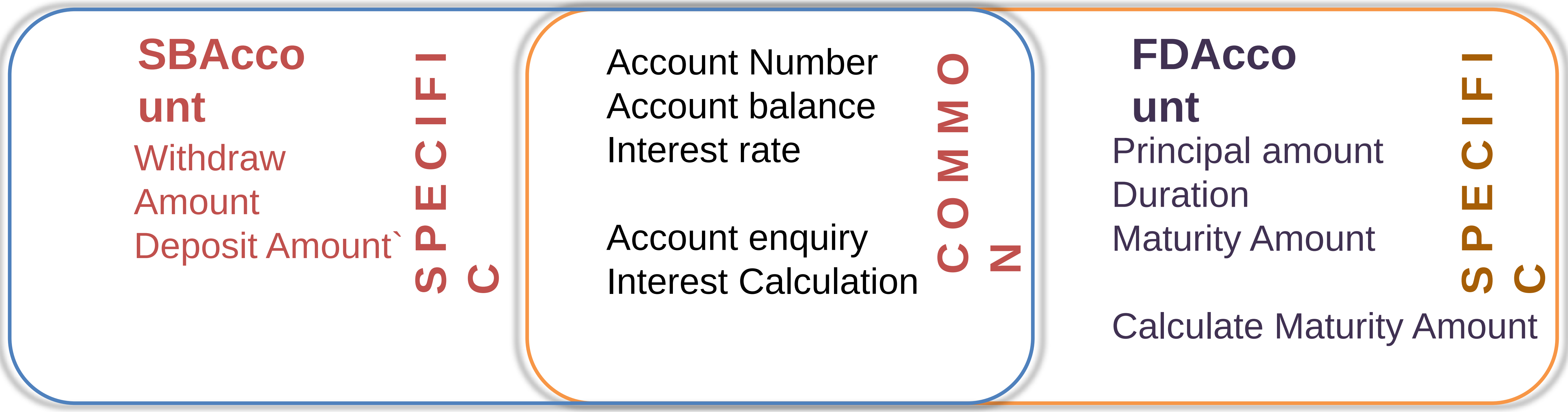
IMPLEMENTING INHERITANCE

```
public class FourWheeler {
  //Fourwheeler's attributes and
methods
}

public class Car extends FourWheeler {
  //inherits Fourwheeler's attributes
and methods
  //Car's specific attributes and
methods
}

public class SportsCar extends Car {
  //inherits Fourwheeler's attributes
and methods
```



GENERALIZATION AND SPECIALIZATION

- Consider the below scenario of Banking which has different types of Accounts
  - Saving Bank Account (SBAccount)
  - Fixed Deposit Account (FDAccount)
  - Each of these accounts have some common and some specific attributes and behaviours



SBAccount

Withdraw Amount
Deposit Amount`

SPECIFIC

Account Number
Account balance
Interest rate

Account enquiry
Interest Calculation

COMMON

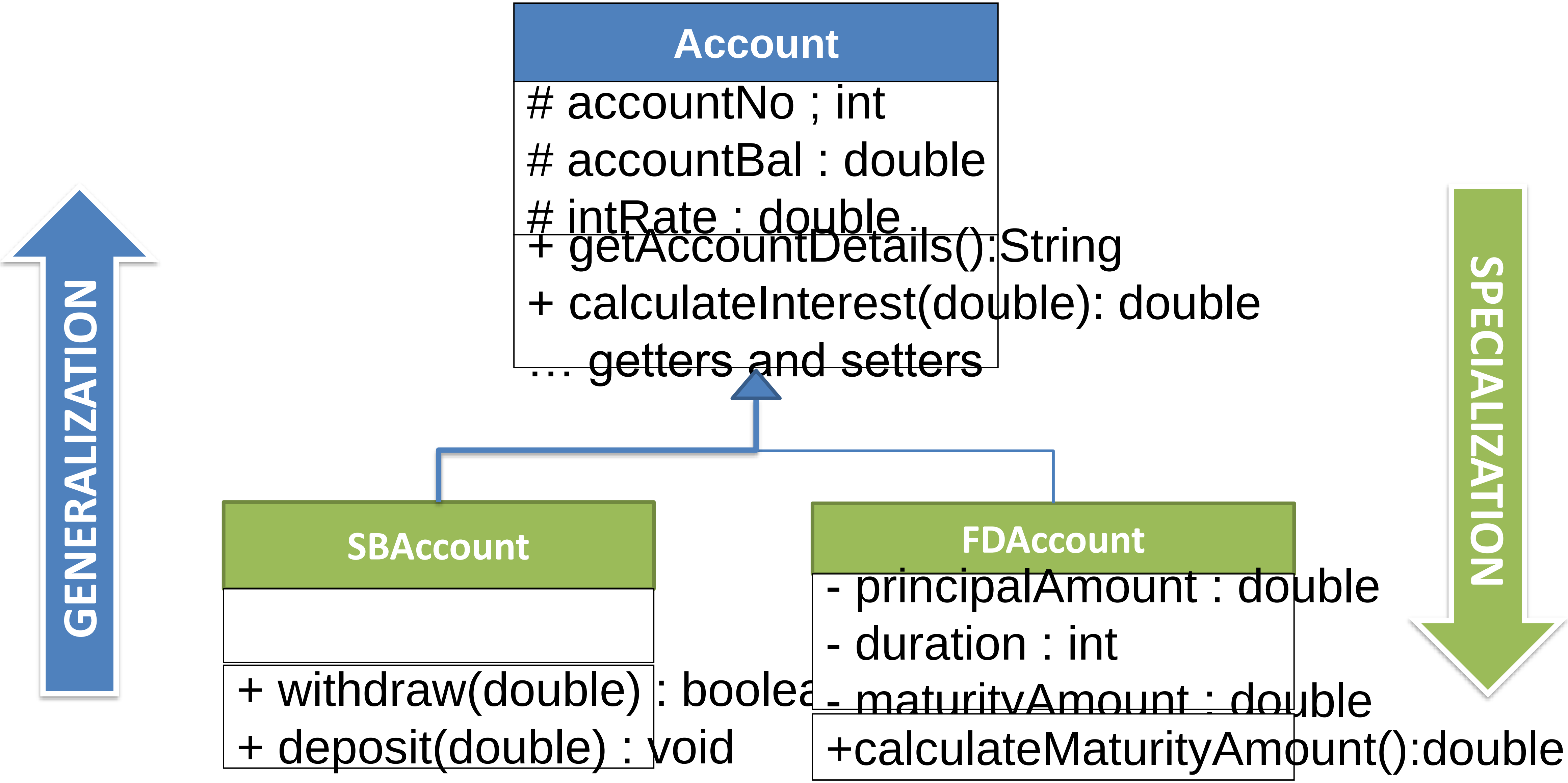
FDAccount

Principal amount
Duration
Maturity Amount

Calculate Maturity Amount

SPECIFIC

GENERALIZATION AND SPECIALIZATION



GENERALIZATION

Account

# accountNo ; int
# accountBal : double
# intRate : double

+ getAccountDetails():String
+ calculateInterest(double): double
… getters and setters

SBAccount

+ withdraw(double) : boolea
+ deposit(double) : void

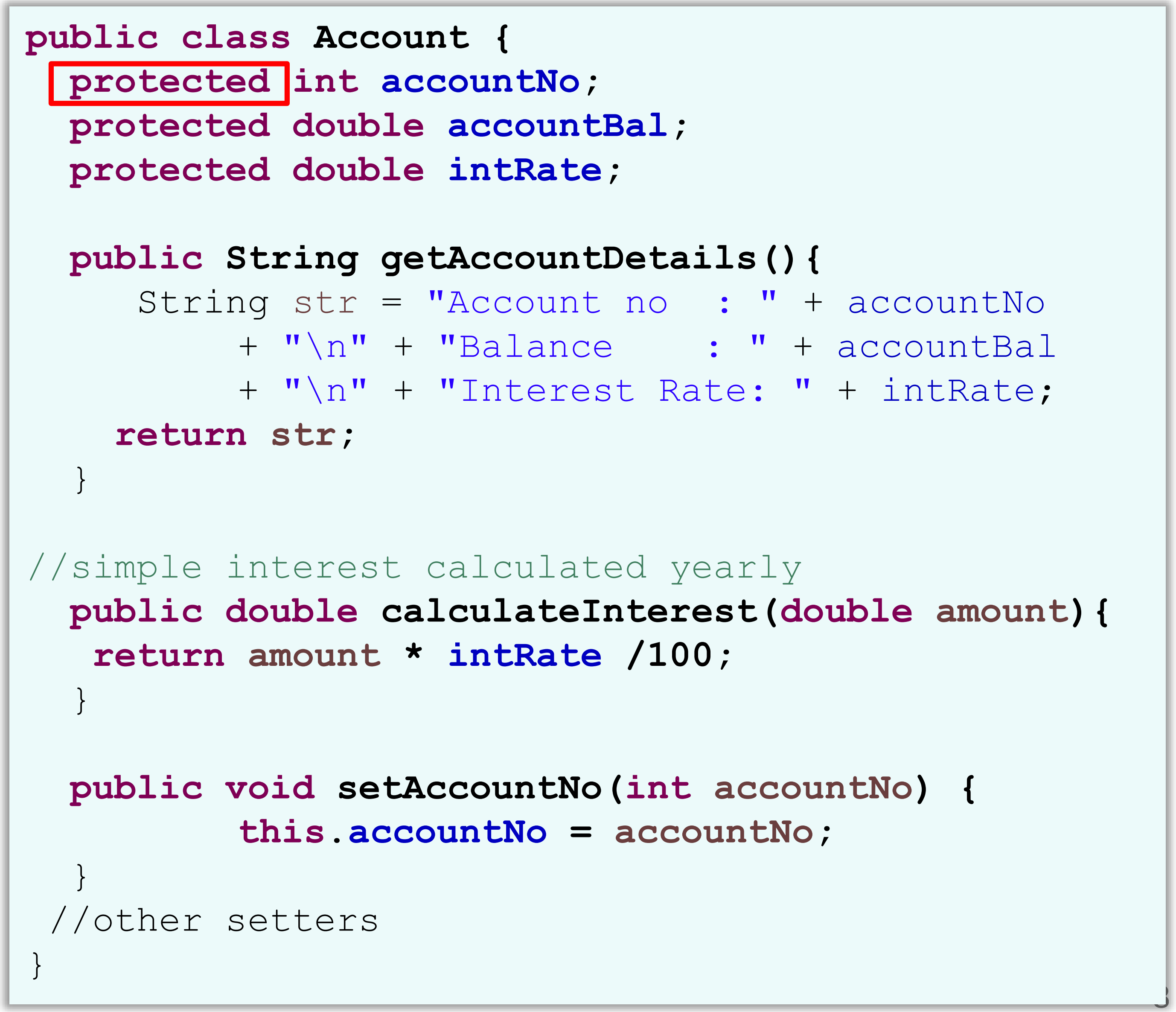
FDAccount

- principalAmount : double
- duration : int
- maturityAmount : double

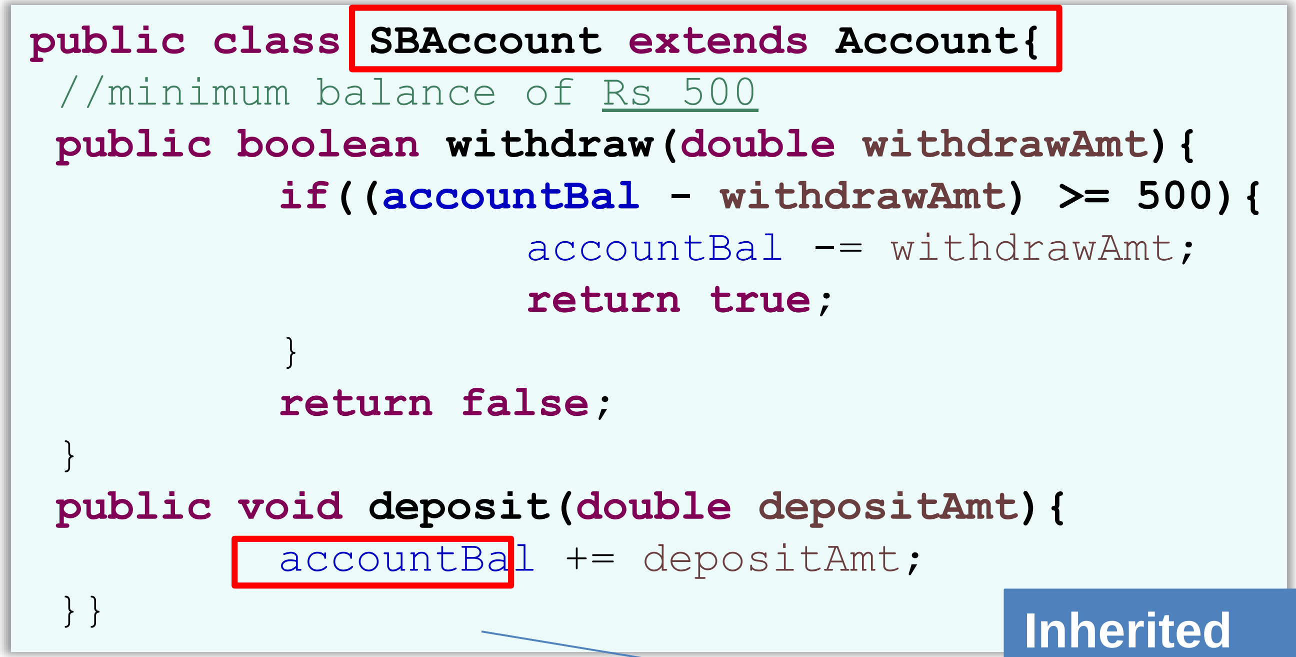
+calculateMaturityAmount():double

SPECIALIZATION

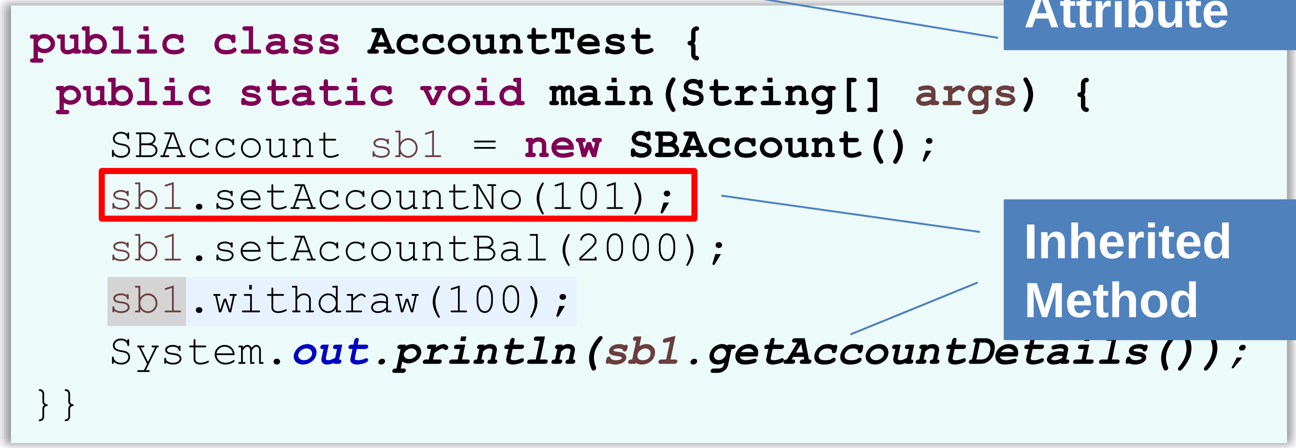
IMPLEMENTING INHERITANCE EXAMPLE



```java
public class Account {
  protected int accountNo;
  protected double accountBal;
  protected double intRate;

  public String getAccountDetails(){
     String str = "Account no  : " + accountNo
         + "\n" + "Balance    : " + accountBal
         + "\n" + "Interest Rate: " + intRate;
    return str;
  }

//simple interest calculated yearly
  public double calculateInterest(double amount){
   return amount * intRate /100;
  }

  public void setAccountNo(int accountNo)  {
         this.accountNo = accountNo;
  }
 //other setters
}
```



```java
public class SBAccount extends Account{
 //minimum balance of Rs 500
 public boolean withdraw(double withdrawAmt){
         if((accountBal - withdrawAmt) >= 500){
                  accountBal -= withdrawAmt;
                  return true;
         }
         return false;
 }
 public void deposit(double depositAmt){
         accountBal += depositAmt;
 }}
```



```java
public class AccountTest {
 public static void main(String[] args)  {
   SBAccount sb1 = new SBAccount();
   sb1.setAccountNo(101);
   sb1.setAccountBal(2000);
   sb1.withdraw(100);
   System.out.println(sb1.getAccountDetails());
}}
```

Inherited Attribute

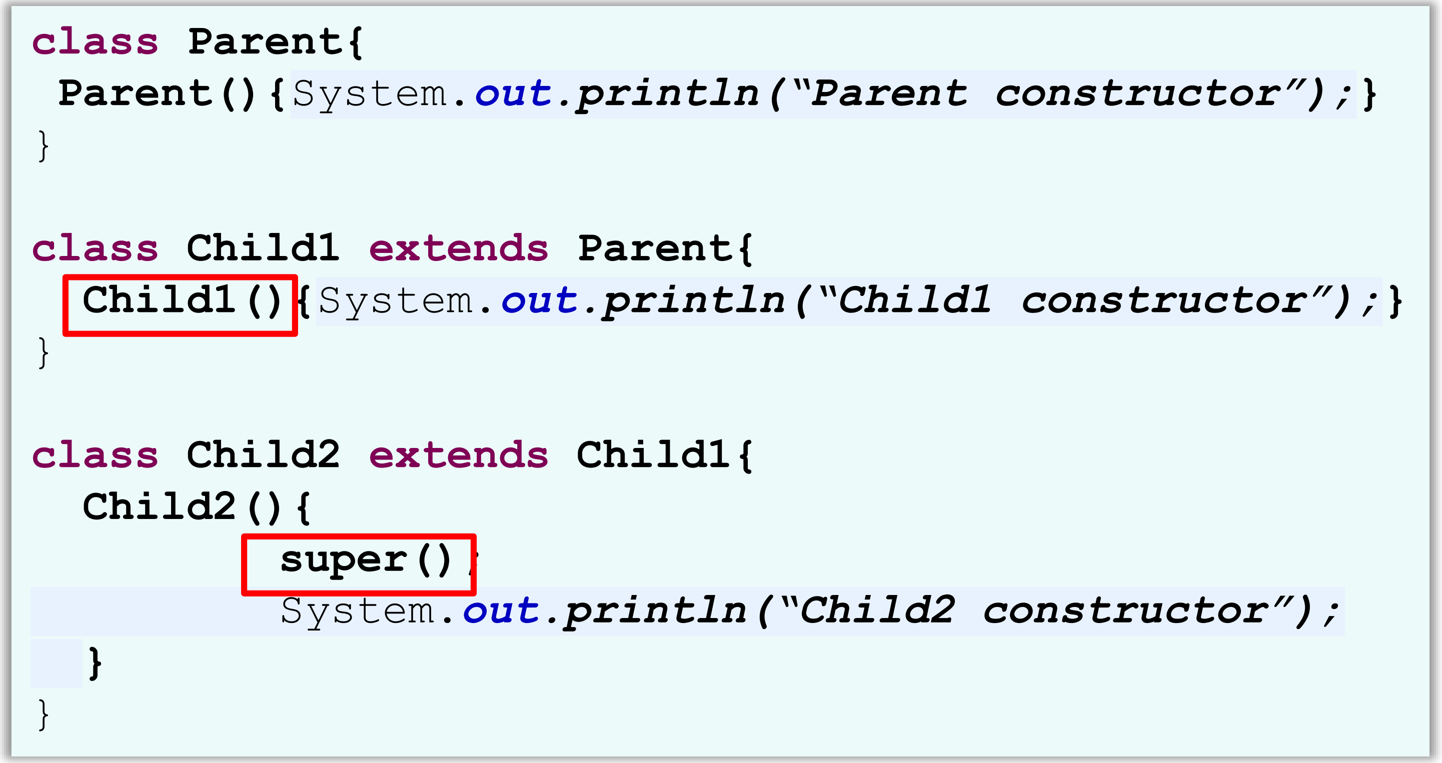
Inherited Method

PROTECTED ACCESS MODIFIER

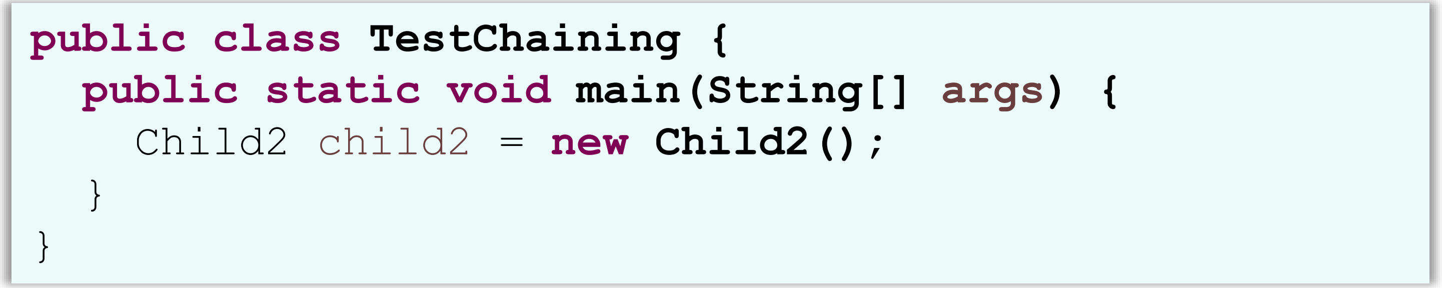
- Protected access modifier
  - allows subclasses in same or other packages to access the inherited attribute/methods directly

Visibility of attributes/methods	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No

CONSTRUCTOR CHAINING



```
class Parent{
 Parent(){System.out.println(“Parent constructor”);}
}

class Child1 extends Parent{
  Child1(){System.out.println(“Child1 constructor”);}
}

class Child2 extends Child1{
  Child2(){
          super();
          System.out.println(“Child2 constructor”);
  }
}
```



```
public class TestChaining {
  public static void main(String[] args) {
    Child2 child2 = new Child2();
  }
}
```

OUTPUT

Parent Constructor
Child1 Constructor
Child2 Constructor

- Creation of child2 object requires constructor of Child2, Child1 and Parent to be executed
- **super()** is used to call a super class constructor
- java places super() as the first statement in the constructor, If super() is not coded by programmer

- Constructor Execution sequence
  - Child2’s constructor invokes child1’s constructor
  - Child1’s constructor invokes Parent’s constructor
  - Parent constructor executes and assigns values to its instance variables
  - Child1 Constructor executes and assigns values to its instance variables
  - Child2 Constructor executes and assigns values to its instance variables
  - Object is created with all inherited attributes initialized

CONSTRUCTOR CHAINING WITH PARAMETERIZED CONSTRUCTOR’S

```
class Parent{
 protected int var1;
 public Parent(int var1) {
  this.var1 = var1;
 }}

class Child1 extends Parent{
  protected int var2;
  public Child1(int var1, int var2) {
    super(var1);
    this.var2 = var2;
  }}

class Child2 extends Child1{
  int var3;
  public Child2(int var1, int var2, int var3) {
    super(var1, var2);
    this.var3 = var3;
  }
  void display(){
    System.out.println("Parent's var1 value = "+ var1);
    System.out.println("Child1's var2 value = "+ var2);
    System.out.println("Child2's var3 value = "+ var3);
  }}
```

```
public class TestChaining {
  public static void main(String[] args) {
    Child2 child2 = new Child2(5, 10, 15);
    child2.display();
  }
}
```

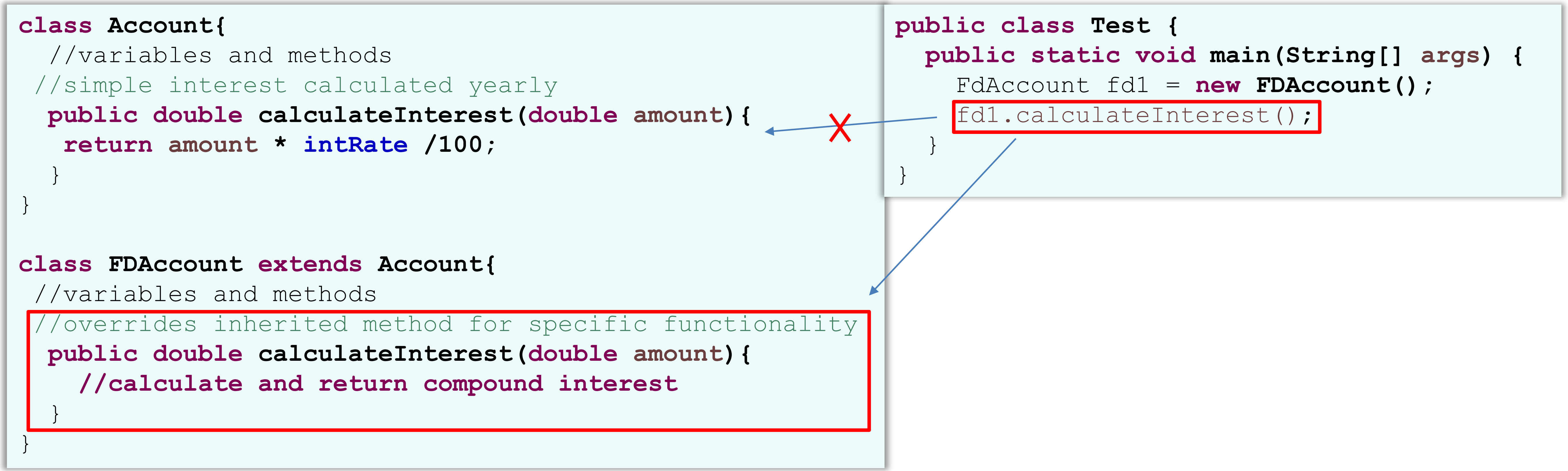
OUTPUT

```
Parent's var1 value = 5
Child1's var2 value = 10
Child2's var3 value = 15
```

- **Programmer has to explicitly code call to superclass parameterized constructor using super(arguments)**

METHOD OVERRIDING

- Subclass can override inherited methods of Superclass
- Why
  - To define behavior that's specific to a particular subclass



```
class Account{
   //variables and methods
  //simple interest calculated yearly
   public double calculateInterest(double amount){
    return amount * intRate /100;
   }
}

class FDAccount extends Account{
  //variables and methods
  //overrides inherited method for specific functionality
   public double calculateInterest(double amount){
     //calculate and return compound interest
   }
}
```

```
public class Test {
  public static void main(String[] args) {
    FdAccount fd1 = new FDAccount();
    fd1.calculateInterest();
  }
}
```

METHOD OVERRIDING RULES

- The overridden method in the Subclass should have the following

Same Method Name

Same parameter list

Same return type

- Access modifier of the overridden method can be less restrictive
  - Ex. If superclass inherited method access is protected, subclass overridden method can have the access as protected and public but not default

METHODS THAT CANNOT BE OVERIDDEN

Private Methods

Final Methods

Static Methods

INVOKING SUPER CLASS METHOD

- super.<methodname>()
  - Inherited superclass method can be invoked in the subclass overridden method by using ‘ super. ‘
  - Done to use the existing functionality of the superclass and add specific functionality

```
class Account{
  //variables and methods
  public String getAccountDetails(){
      //code for formatting Account variables
  }
}

class FDAccount extends Account{
 //variables and methods
 //overrides inherited method for specific functionality
   public String getAccountDetails(){
     String str = super.getAccountDetaiils();
    //Add code to format FDAccount specific variables
}}
```

FINAL KEYWORD

- Final keyword can be used with variable , method declaration and class declarations

  - When used with Variables

    ```
    final int i = 10;
    ```

    Value of variable cannot be changed

  - When used with Methods

    ```
    final void method(){ }
    ```

    Method cannot be overridden in a subclass

  - When used with Class

    ```
    final class MyClass{  }
    ```

    Class cannot be extended/inherited

Cosmic Class

COSMIC CLASS

- Every class in java implicitly inherits from the a class called Object
- A class inheriting from a different super class, still inherits from Object through multi-level Inheritance

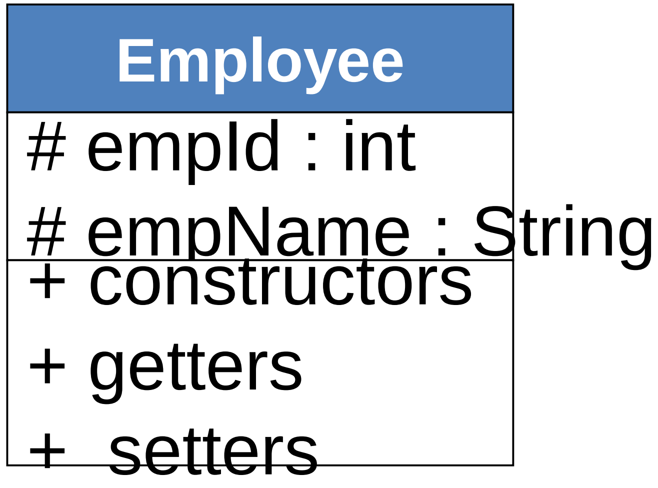
- Java.lang.Object
  - Doesn’t have any super class and hence, often referred as Cosmic class
  - Does not have member variables
  - Has some important methods like toString(), equals(), hashcode() which should typically be overridden by every class

STRING REPRESENTATION OF OBJECT

public String toString()

- Returns a String representation of an Object
- Default implementation of toString() method in Object class returns a String containing the classname and hashcode in hex format

```
Employee e1 = new Employee(100,”John”);
String str = e1.toString();
System.out.println(str);

// Prints Employee@3C45BCD
```



Employee

# empId : int

# empName : String

+ constructors

+ getters

+ setters

STRING REPRESENTATION OF OBJECT

- Typically, toString() method
  - Should return a string that textually represents the object
  - Should give concise and informative result for a person to read
  - Should be overridden to achieve the above

```
public String toString() {
return "Employee [Employee ID :" + empId
        + ", Employee Name :" + empName + "]"; }
```

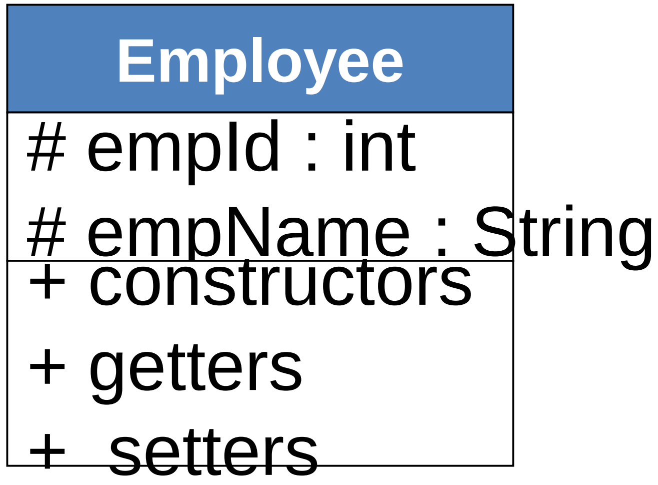
```
Employee e1 = new Employee(100,"John");
String str = e1.toString();
System.out.println(str);
// Prints Employee [Employee ID :100, Employee Name :John]

System.out.println(e1); //invokes toString() automatically
```

OBJECT EQUALITY

public boolean equals(Object object)

- Default implementation of equals() method in Object class compares the references
  - Any two distinct objects compared using the default equals method always returns false

```
Employee e1 = new Employee(100,”John”);
Employee e2 = new Employee(100,”John”);

System.out.println(e1.equals(e2));
// PRINTS FALSE (DEFAULT EQUALS IMPLEMENTATION)
```



OBJECT EQUALITY

- Class needs to override the inherited equals() method to compare two object which are logically equal

- To compare two employees, Employee class has to override equals method

```
Employee e1 = new Employee(100,”John”);
Employee e2 = new Employee(100,”John”);

System.out.println(e1.equals(e2));
// PRINTS TRUE
```

```
@Override
public boolean equals(Object obj) {
     if (this == obj) return true;
     if (obj == null) return false;
     if (getClass() != obj.getClass())
     return false;
     Employee other = (Employee) obj;
     if (empId != other.empId)
     return false;
     if (empName == null) {
     if (other.empName != null)
     return false;
     } else if (!empName.equals(other.empName))
          return false;
     return true;
}
```

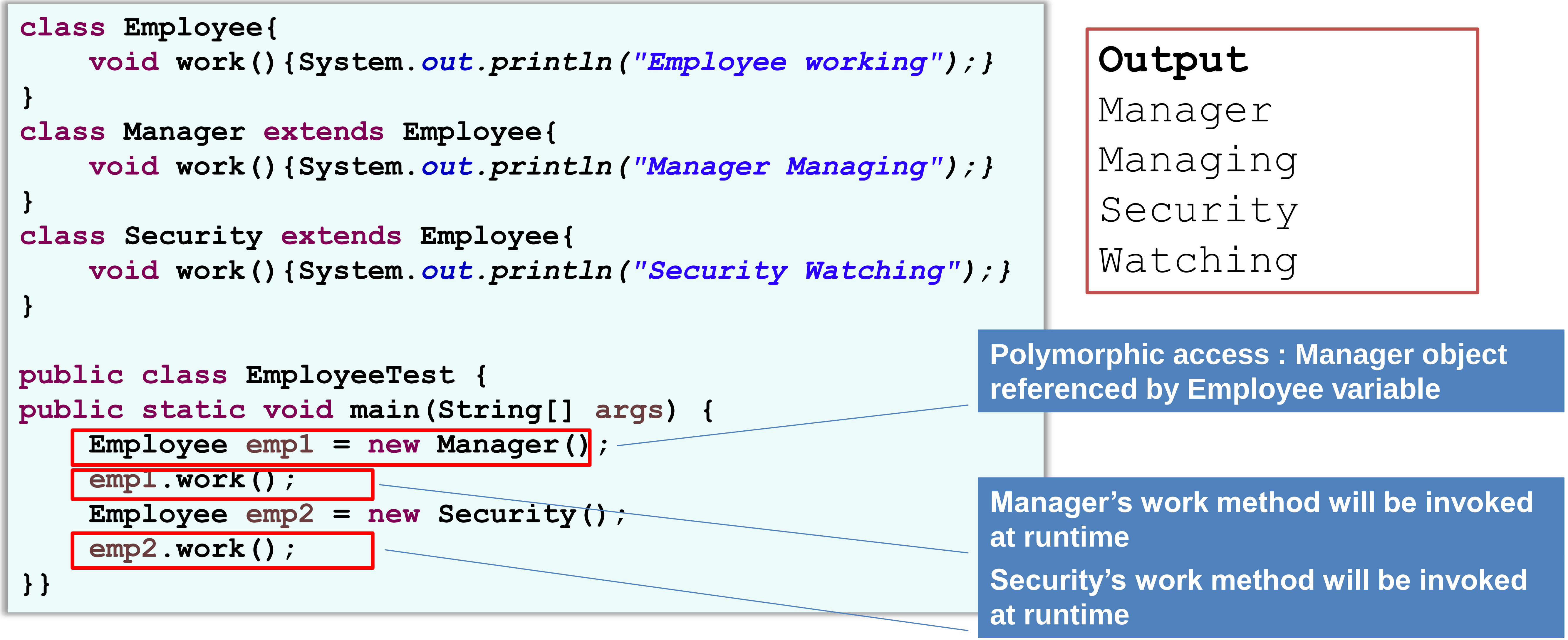
Implementing Polymorphism

POLYMORPHISM

- Is a concept by which a single action can be performed in various ways
- Ability of an object to take many forms
  - A superclass method can **adopt different forms**, depending on the subclass object

- Advantages
  - Allows objects in one inheritance hierarchy to share same interface(methods)
  - Allows superclass variable to reference its subclass object and invoke specific functionality at runtime

IMPLEMENTING POLYMORPHISM



```
class Employee{
    void work(){System.out.println("Employee working");}
}
class Manager extends Employee{
    void work(){System.out.println("Manager Managing");}
}
class Security extends Employee{
    void work(){System.out.println("Security Watching");}
}

public class EmployeeTest {
public static void main(String[] args) {
    Employee emp1 = new Manager();
    emp1.work();
    Employee emp2 = new Security();
    emp2.work();
}}
```

Polymorphic access : Manager object referenced by Employee variable

Manager’s work method will be invoked at runtime

Security’s work method will be invoked at runtime

Output

```
Manager
Managing
Security
Watching
```

REFERENCE VARIABLE EXPLICIT DOWNCASTING

```
class Employee{
    void work(){System.out.println("Employee working");}
}
class Security extends Employee{
    void work(){System.out.println("Security Watching");}
    void drill(){System.out.println("Performing drill");}
}

public class EmployeeTest {
public static void main(String[] args) {
    Employee emp2 = new Security();
    emp2.drill();
    if (emp2 instanceof Security){
     ((Security) emp2).drill();
    }
}}
```

Employee variable cannot access drill() method as it is not defined in Employee class

Explicitly downcast Employee to Security

DYNAMIC VS STATIC POLYMORPHISM

- Binding is the association of the Method Definition to the Method Call

Dynamic / Runtime polymorphism

- Process in which a call to an overridden method is resolved at **runtime**
- Binding at runtime
- Demonstrated by **method overriding**

Static / compile time polymorphism

- Methods invoked by checking method signatures at **compile time**
- Binding at compile time
- Achieved through **method overloading**