Inheritance

Introduction to Inheritance

- Subclass and superclass are closely related
  - Subclass share fields and methods of superclass
  - Subclass can have more fields and methods
  - Implementations of a method in superclass and subclass can be different
  - An object of subclass is automatically an object of superclass, but not vice versa
    - The set of subclass objects is a subset of the set of superclass objects. (E.g. The set of `Managers` is a subset of the set of `Employees`.) This explains the term subclass and superclass.

Introduction to Inheritance

Why inheritance?

- `Employee` class:

  `name, salary, hireDay;`

  `getName, getSalary(),raiseSalary(), getHireDay().`

- `Manager` **is-a** `Employee`, has all the above artifacts, and
  - Has a bonus
  - getsalary() is computed differently

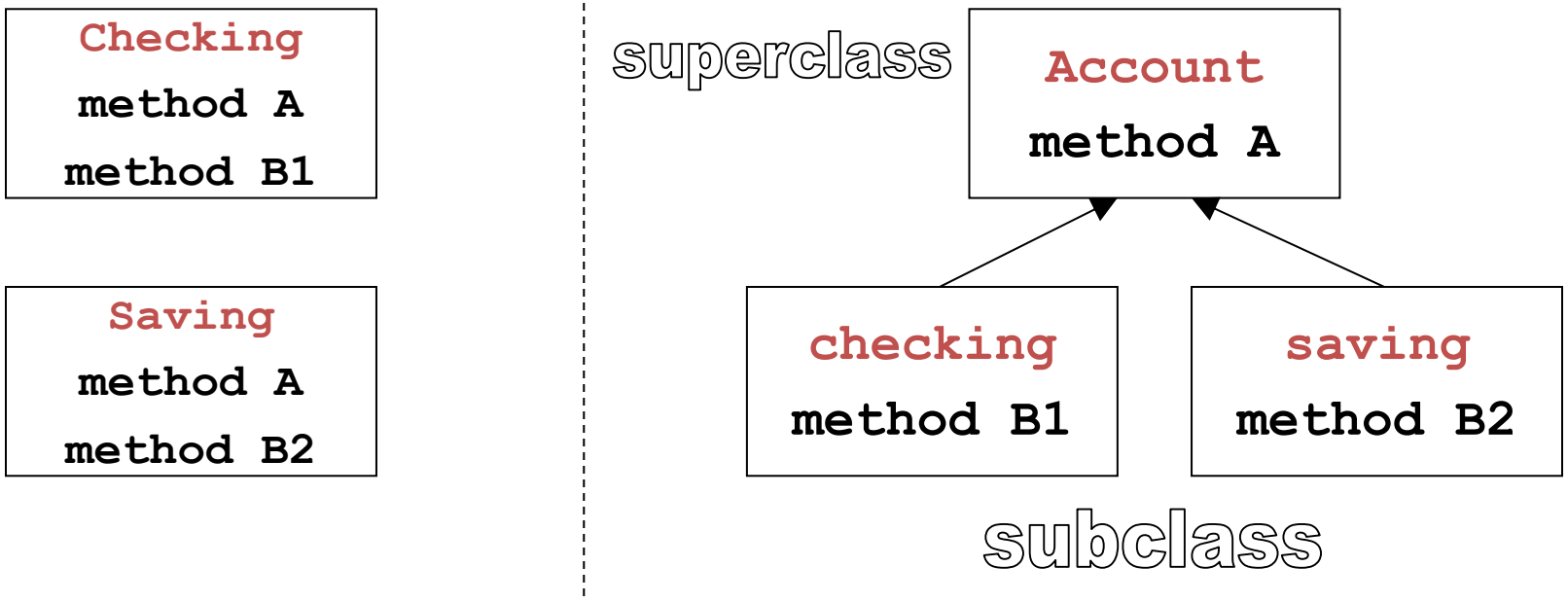
- Instead of defining `Manager` class from scratch, one can derive it from the `Employee` class. Work saved.

Introduction to Inheritance

Why inheritance?

Avoids Duplication!

Inheritance allows one to factor out common functionality by moving it to a superclass, results in better program.



Introduction to Inheritance

- Multiple inheritance
  - A class extends >1 superclasses

- Java does not support multiple inheritance
  - A java class can only extend ONE superclass
  - Functionality of multiple inheritance recovered by interfaces.

Deriving a class

```
class Employee
{
  public Employee(String n, double s, int year, int
  month, int day) {…}

  public String getName(){…}
  public double getSalary() {…}
  public Data getHireDay(){…}
  public void raiseSalary(double byPercent) {…}

  private String name;
  private double Salary;
  private Date hireDay;
}
```

Deriving a class

- Extending `Employee` class to get `Manager` class

```
class Manager extends Employee
{ public Manager(…) {…}          // constructor

  public void getSalary(…) {…}   // refined method

  // additional methods
  public void setBonus(double b){…}

  // additional field
  private double bonus;
}
```

Fields of subclass

- Semantically: Fields of superclass + additional fields
  - `Employee`
    - `Name, salary, hireday`
  - `Manager`
    - `name, salary, hireday`
    - `bonus`

- Methods in subclass cannot access private fields of superclass.
  - After all, subclass is another class viewed from super class.

Constructors of Subclass

- Every constructor of a subclass must, directly or indirectly, invoke a constructor of its superclass to initialize fields of the superclass. (Subclass cannot access them directly)

- Use keyword `super` to invoke constructor of the superclass .

```
public Manager(String n, double s, int
  year, int month, int day)
{
    super(n, s, year, month, day);
    bonus = 0;
}
```

Constructors are not inherited

Overriding Methods

- Salary computation for managers are different from employees. So, we need to modify the `getSalary`, or provide a new method that overrides `getSalary`

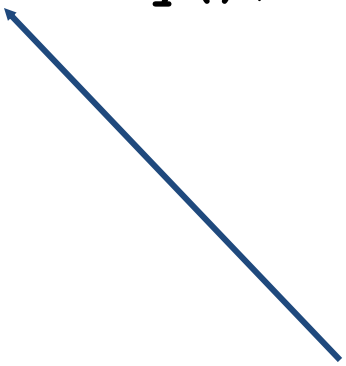
```
public double getSalary( )
{   double baseSalary = super.getSalary();
    return basesalary + bonus;
 }
```

- Cannot replace the last line with

```
salary += bonus;
```

  Because **`salary`** is `private` to `Employee`.

- Cannot drop “super.”, or else we get an infinite loop



Call method of superclass

Overriding Methods

- An overriding method must have the same **signature** (name and parameter list) as the original method. Otherwise, it is simply a new method:

  – Original Method in `Employee`:

```
public double getSalary( ){…}
public void raiseSalary(double byPercent){…}
```

  – New rather than overriding methods in `Manager`:

```
public void raiseSalary(int byPercent){…}
public void raiseWage(double byPercent){…}
```

Overriding Methods

– An overriding method must have the same return type as the original method:

  • The following method definition in `Manager` would lead to compiler error:

```
public int getSalary( ){…}
```

– An overriding method must be at least as visible as the superclass method.

- `private` methods cannot be overridden, but others (public, protected, default-access methods) can.

Class Compatibility

- **Object of a subclass can be used in place of an object of a superclass**

```
Manager harry = new Manager(...);
Employee staff = harry;
Employee staff1 = new Manager(...);
```

`harry` automatically cast into an `Employee`, widening casting.

- **Why does `staff.getSalary()` work correctly?**
  - **Employee has method `getSalary`. No compiling error.**
  - **Correct method found at run time via dynamic binding**

Class Compatibility

- The opposite is not true

```
Employee harry = new Employee(...);
Manager staff = harry;      // compiler error
Manager staff1 = new Employee(...); // compiler error
```

Polymorphism & Dynamic binding

- Method call: case 1

```
Employee harry = new Employee(...);
harry.getSalary();   // calls method of Employee
```

- Method call: case 2

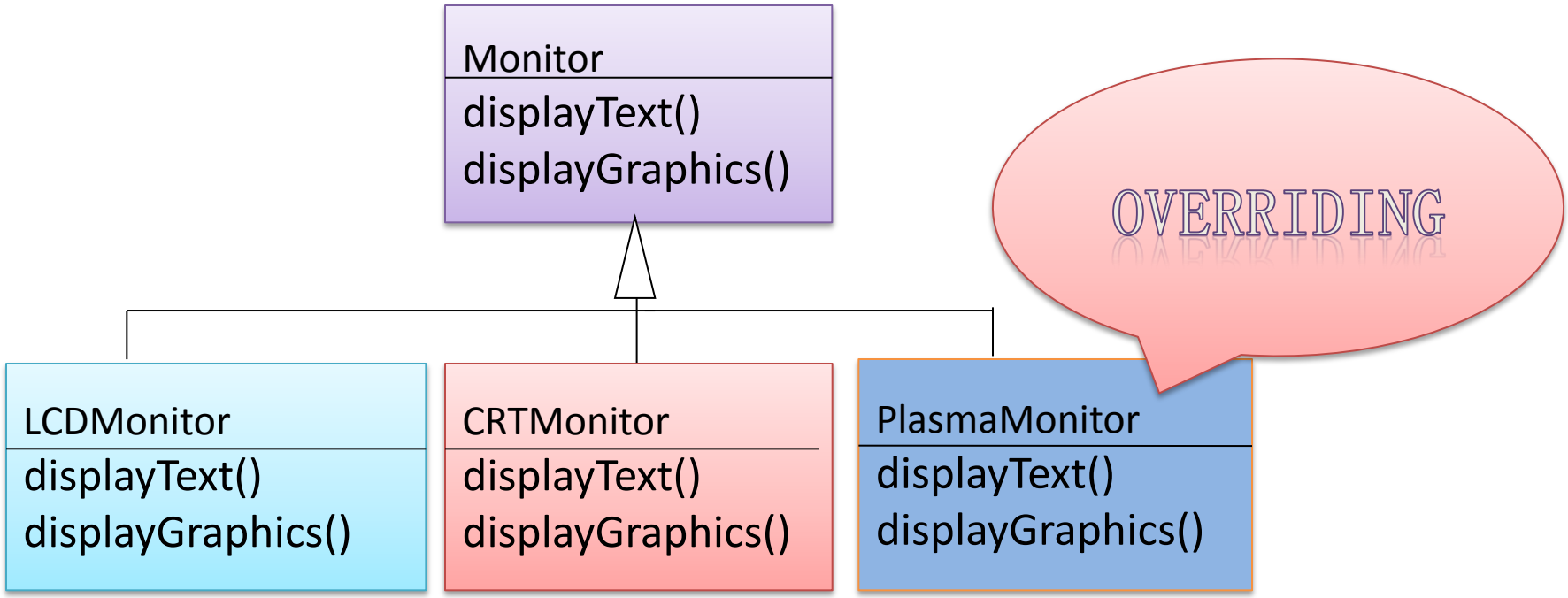
```
Manager carl = new Manager(...);
carl.getSalary();  // calls method of Manager
```

- Method call: case 3

```
Manager carl = new Manager(...);
Employee staff = carl;
staff.getSalary();
```

  - Calls method of `Employee` or `Manager`?
  - Answer: method of `Manager`.

Polymorphism Example



Abstract Classes

```
class Person
{  public Person(String n) {    name = n;}
    public String getName()
    { return name;}

    public String getDescription();
    // but how to write this?

   private String name;
}
```

- The `Person` class knows nothing about the person except for the name. We don't know how to implement the method in the `Person` class although we know it must be there.

Abstract Classes

- Solution: leave the **`getDescription`** method abstract
  - Hence leave the **`Person`** class abstract

```
abstract class Person
{  public Person(String n)
   {   name = n;}

   public abstract String getDescription();

   public String getName()
   { return name;}
   private String name;
}
```

Abstract Classes

- An abstract method is a method that
  - Cannot be specified in the current class.
  - Must be implemented in non-abstract subclasses.
- An abstract class is a class that may contain one or more abstract methods
- Notes:
  - An abstract class does not necessarily have abstract method
  - Subclass of a non-abstract class can be abstract.

Abstract Classes

- Cannot create objects of an abstract class:

```
New Person("Micky Mouse")   // illegal
```

- An abstract class must be extended before use.

```
class Student extends Person
{  public Student(String n, String m)
   {  super(n);  major = m;}

   public String getDescription()
   { return "a student majoring in " + major; }
   private String major;
}
```

Abstract Classes

```
class Employee extends Person
{   …
    public String getDescription()
    {
        NumberFormat formatter
            = NumberFormat.getCurrencyInstance();
        return "an employee with a salary of "
            + formatter.format(salary);
    }
    …
    private double salary;
}
```

Open-Closed Principle

- **In object-oriented programming, the open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"**

- That is, once a class is written it should not be required to be rewritten unless its specification changes!!!!!

Open-Close

- The Open Close Principle states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

?????

- If you write code that requires class B to be updated if class A changes….
- YOU GOT IT WORNG – YOUR SOLUTION NEEDS SCRAPPED!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
- IT IS COMPLETELY UNECONOMICAL!!!!!!!!!!

Bad Code

```
class GraphicEditor {

public void drawShape(Shape s) {
if (s.m_type==1)
drawRectangle(s);
else if (s.m_type==2)
drawCircle(s);
}
public void drawCircle(Circle r)
{....}
public void
drawRectangle(Rectangle r) {....}
}
```

```
class Shape {
int m_type;
}

class Rectangle extends Shape {
Rectangle() {
super.m_type=1;
}
}

class Circle extends Shape {
Circle() {
super.m_type=2;
}
}
```

Good Code

```
class GraphicEditor {
public void drawShape(Shape
s) {
        s.draw();
}
}
```

```
abstract class Shape {
        abstract void draw();
 }

 class Rectangle extends Shape
{
        public void draw() {….
        }
 }
```

Dependency Inversion Principle

- Low level classes, the classes which implement basic and primary operations
- high level classes, the classes which encapsulate complex logic
- A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes.
- What happens if we need to replace a low level class?

Example

- Let's take the classical example of a copy module which reads characters from the keyboard and writes them to the printer device.
- The high level class containing the logic is the Copy class.
- The low level classes are KeyboardReader and PrinterWriter.

Going Wrong

- In a bad design the high level class uses directly and depends heavily on the low level classes.
- In such a case if we want to change the design to direct the output to a new FileWriter class
- we have to make changes in the Copy class.

Back on path

- In order to avoid such problems we can introduce an abstraction layer between high level classes and low level classes.
- Since the high level modules contain the complex logic they should not depend on the low level modules
- so the new abstraction layer should not be created based on low level modules.

Low level modules are to be created based on the abstraction layer.

from high level modules to the low level modules:

High Level Classes -->

Abstraction Layer -->

Low Level Classes

Use it!

Can we have a real example

- We have the manager class which is a high level class, and the low level class called Worker. We need to add a new module to our application to model the changes in the company structure determined by the employment of new specialized workers. We created a new class SuperWorker for this.
- Let's assume the Manager class is quite complex, containing very complex logic. And now we have to change it in order to introduce the new SuperWorker. Let's see the disadvantages:
- we have to change the Manager class (remember it is a complex one and this will involve time and effort to make the changes).

And …

- some of the current functionality from the manager class might be affected.
- the unit testing should be redone.
- All those problems could take a lot of time to be solved and they might induce new errors in the old functionality.
- The situation would be different if the application had been designed following the Dependency Inversion Principle.
- It means we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface.
- When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it. No additional changes in the existing classes.

Bad Code

```
class Manager {
Worker worker;

public void setWorker(Worker
w) {
worker = w;
}

public void manage() {
worker.work();
}
}
```

```
class Worker {
public void work() {
 ....
}
}
```

```
class SuperWorker {
public void work() {
....
}
}
```

Good Code

```
class Manager {
IWorker worker;

public void setWorker(IWorker
w) {
worker = w;
}

public void manage() {
worker.work();
}
}
```

```
abstract class IWorker {
abstract void work();
}

class Worker extends IWorker{
public void work() {….
}}

class SuperWorker extends
IWorker{
public void work() {….
}}
```

Object class

- **The** `Object` **class** (`java.lang.Object`) is the mother of all classes

- Never write `class Employee extends Object {…}` because `Object` is taken for granted if no explicitly superclass:

```
class Employee {…}
```

That is