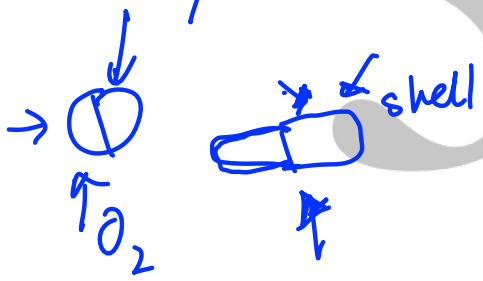
Encapsulation and Subclassing



Encapsulation

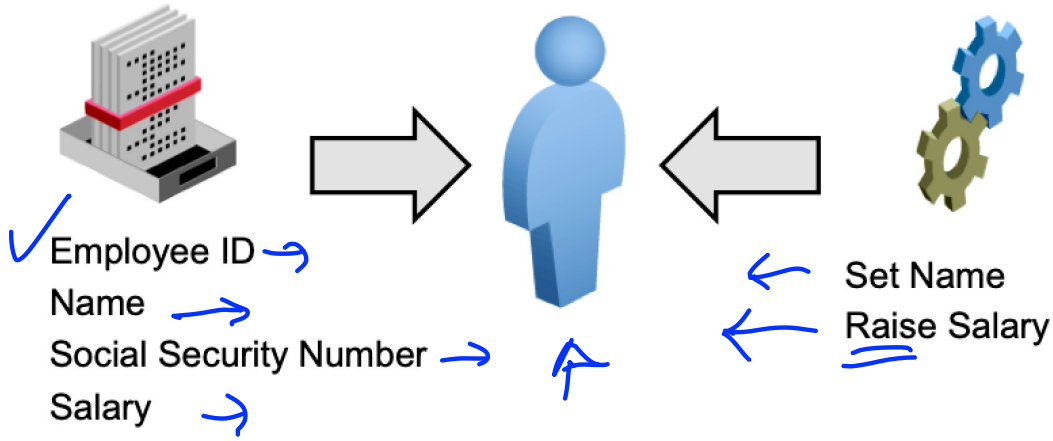
The term *encapsulation* means to enclose in a capsule, or to wrap something around an object to cover it. In object-oriented programming, encapsulation covers, or wraps, the internal workings of a Java object.

- Data variables, or fields, are hidden from the user of the object.
- Methods, the functions in Java, provide an explicit service to the user of the object but hide the implementation.
- As long as the services do not change, the implementation can be modified without impacting the user.

Access Modifier
public, private

Encapsulation: Example

What data and operations would you encapsulate in an object that represents an employee?



Employee

ID

Name

SSN

Salary

setters

getters

Encapsulation: Private Data, Public Methods

One way to hide implementation details is to declare all of the fields `private`.

```
1 public class CheckingAccount {
2     private int custID;
3     private String name;
4     private double amount;
5     public CheckingAccount (
6     }
7     public void setAmount (double amount) {
8         this.amount = amount;
9     }
10     public double getAmount () {
11         return amount;
12     }
13     //... other public accessor and mutator methods
14 }
```

Declaring fields `private` prevents direct access to this data from a class instance.

private

Public and Private Access Modifiers

- The `public` keyword, applied to fields and methods, allows any class in any package to access the field or method.
- The `private` keyword, applied to fields and methods, allows access only to other methods within the class itself.

```
CheckingAccount chk = new CheckingAccount ();
chk.amount = 200; // Compiler error - amount is a private field
chk.setAmount (200); // OK
```

- The `private` keyword can also be applied to a method to hide an implementation detail.

```
// Called when a withdrawal exceeds the available funds
private void applyOverdraftFee () {
    amount += fee;
}
```

Revisiting Employee

The `Employee` class currently uses `public` access for all of its fields. To encapsulate the data, make the fields `private`.

```
package come.example.model;
public class Employee {
    private int empId;
    private String name;
    private String ssn;
    private double salary;
    //... constructor and methods
}
```

Encapsulation step 1: Hide the data (fields).

Method Naming: Best Practices

Although the fields are now hidden using `private` access, there are some issues with the current `Employee` class.

- The setter methods (currently `public` access) allow any other class to change the ID, SSN, and salary (up or down).
- The current class does not really represent the operations defined in the original `Employee` class design.
- Two best practices for methods:
  - Hide as many of the implementation details as possible.
  - Name the method in a way that clearly identifies its use or functionality.
- The original model for the `Employee` class had a Change Name and Increase Salary operation.

Employee Class Refined

```
1 package com.example.domain;
2 public class Employee {
3     // private fields ...
4     public Employee () {
5     }
6     // Remove all of the other setters
7     public void setName(String newName) {
8        if (newName != null) {
9              this.name = newName;
10          }
11     }
12
13     public void raiseSalary(double increase) {
14         this.salary += increase;
15     }
16 }
```

Encapsulation step 2: These method names make sense in the context of an Employee.

ORACLE

Make Classes as Immutable as Possible

```
1 package com.example.domain;
2 public class Employee {
3     // private fields ...
4     // Create an employee object
5     public Employee (int empId, String name,
6                      String ssn, double salary) {
7        this.empId = empId;
8        this.name = name;
9        this.ssn = ssn;
10       this.salary = salary;
11    }
12
13    public void setName(String newName) { ... }
14
15    public void raiseSalary(double increase) { ... }
16 }
```

Encapsulation step 3: Replace the no-arg constructor with a constructor to set the value of all fields.

Creating Subclasses

You created a Java class to model the data and operations of an `Employee`. Now suppose you wanted to specialize the data and operations to describe a `Manager`.

```
1  package com.example.domain;
2  public class Manager {
3      private int empId;
4      private String name;
5      private String ssn;
6      private double salary;
7      private String deptName;
8      public Manager () { }
9      // access and mutator methods...
10 }
```

wait a minute...
this code looks very familiar....

forget

getters

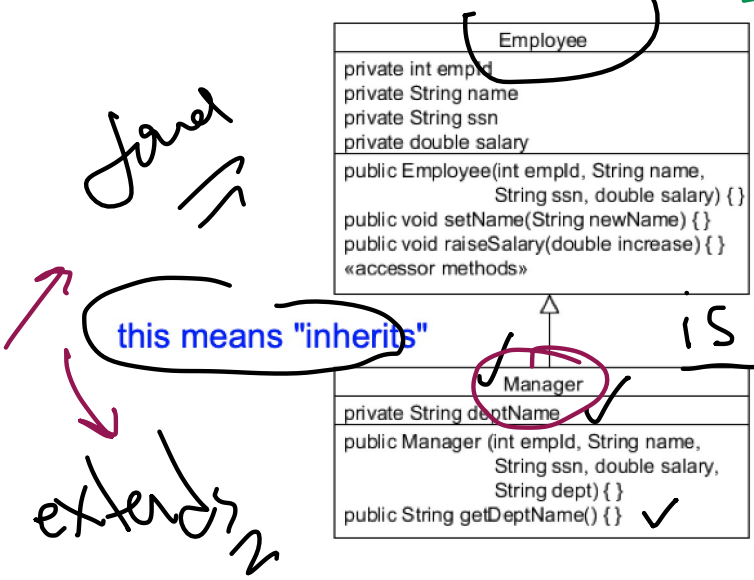
setters

twice? NO

inheritance

Subclassing

In an object-oriented language like Java, subclassing is used to define a new class in terms of an existing one.



UML

Java

this means "inherits"

is a

extends

superclass: `Employee` ("parent" class)

subclass: `Manager`, is an `Employee` ("child" class)

SuperClass

SubClass

```
1 package com.example.domain;
2 public class Manager extends Employee {
3     private String deptName;
4     public Manager (int empId, String name,
5                     String ssn, double salary, String dept) {
6       super (empId, name, ssn, salary);
7         this.deptName = dept;
8     }
9
10      public String getDeptName () {
11          return deptName;
12      }
13      // Manager also gets all of Employee's public methods!
14 }
```

The `super` keyword is used to call the constructor of the parent class. It must be the first statement in the constructor.

Object Oriented Programming

Java

Everything is an Object

Object?

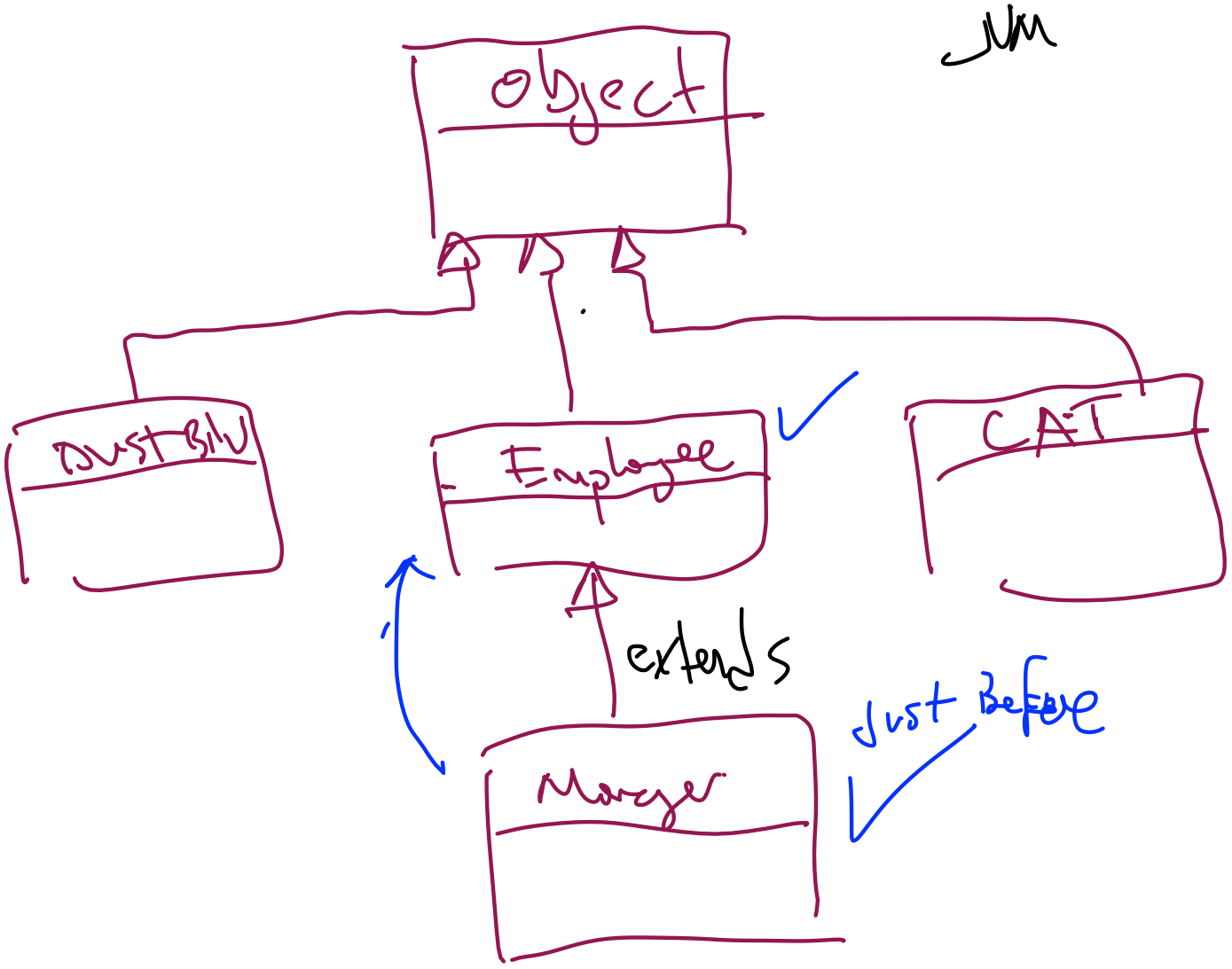
CAR

Employee

Manager

DustBin

CAR


Object
Employee
CAT
extends
Just Before
Merger

```java
package test;

public class Driver {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Employee emp= new Employee("EMP1", 45, 8000);


        Manager man= new Manager("Boss", 65, 50000, "Finance");


        man.getInfo();


        emp.getInfo();


    }

}
```

```java
package test;
public class Employee {

    private String Name;
    private int Age;
    private  double  salary;
    private  final double TAX=0.2;
    private double taxAmount;


    public Employee() {

    }

public Employee(String Name, int Age, double salary) {
    setAge(Age);
    setName(Name);
    setSalary(salary);
    System.out.println("Name, Age and Salary is assigned in EMPLOYEE Constructor.");
}

 public void setAge( int Age) {

        if(Age>0)
         this.Age=Age;
        else
            System.out.println("Age can not be negative");

    }

public String getName() {
    return Name;
}

public void setName(String name) {
    if(name!=null)
        Name = name;
}

public double getSalary() {
    return salary;
```

```
50  public double getTaxAmount() {
51      return taxAmount;
52  }
53
54      public int getAge() {
55
56          return Age;
57
58      }
59
60
61
62  private void calculateTaxAmount() {
63
64      taxAmount=getSalary()*TAX;
65
66  }
67
68
69  public void getInfo() {
70
71      System.out.println("------Employee Information-------");
72      System.out.println("Name: "+ getName());
73      System.out.println("Age :"+ getAge());
74      System.out.println("Salary:" + getSalary());
75      System.out.println(getTaxAmount()+ " $ the tax amount");
76
77
78  }
79
80  }
81
```

```
1  package test;
2
3  public class Manager extends Employee {
4
5      private String dept;
6
7      public Manager(String Name, int Age, double salary, String dept) {
8          super(Name, Age, salary);
9          //super();
10         // TODO Auto-generated constructor stub
11         this.dept=dept;
12         System.out.println("Department is assined in MANAGER Constructor.");
13     }
14
15
16     public String getDept() {
17         return dept;
18     }
19
20     @Override
21     public void getInfo() {
22         System.out.println(" -----Manager Indormation------");
23         System.out.println("Department :"+ getDept());
24         super.getInfo();
25
26
27     }
28
29 }
30
```

Constructors Are Not Inherited

Although a subclass inherits all of the methods and fields from a parent class, it does not inherit constructors. There are two ways to gain a constructor:

- Write your own constructor.
- Use the default constructor.
  - If you do not declare a constructor, a default no-argument constructor is provided for you.
  - If you declare your own constructor, the default constructor is no longer provided.

Using `super` in Constructors

To construct an instance of a subclass, it is often easiest to call the constructor of the parent class.

- In its constructor, `Manager` calls the constructor of `Employee`.

```
super(empId, name, ssn, salary);
```

- The `super` keyword is used to call a parent's constructor.
- It must be the first statement of the constructor.
- If it is not provided, a default call to `super()` is inserted for you.
- The `super` keyword may also be used to invoke a parent's method or to access a parent's (non-private) field.

Constructing a Manager Object

Creating a `Manager` object is the same as creating an `Employee` object:

```
Manager mgr = new Manager (102, "Barbara Jones",
                           "107-99-9078", 109345.67, "Marketing");
```

- All of the `Employee` methods are available to `Manager`:

```
mgr.raiseSalary (10000.00);
```

- The `Manager` class defines a new method to get the Department Name:

```
String dept = mgr.getDeptName();
```

Encapsulation, Inheritance, Polymorphism

What Is Polymorphism?

The word *polymorphism*, strictly defined, means "many forms."

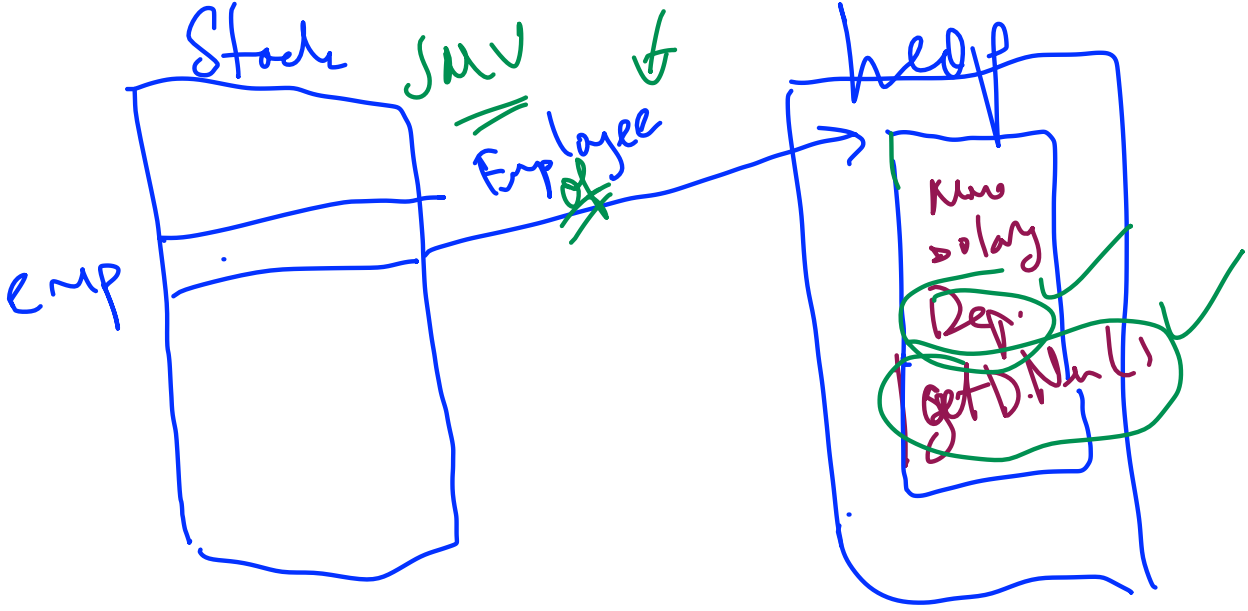
```
Employee emp = new Manager();
```

- This assignment is perfectly legal. An employee can be a manager.
- However, the following does not compile:

```
emp.setDeptName ("Marketing"); // compiler error!
```

- The Java compiler recognizes the `emp` variable only as an `Employee` object. Because the `Employee` class does not have a `setDeptName` method, it shows an error.

emp.setDeptName



Overloading Methods

Your design may call for several methods in the same class with the same name but with different arguments.

```
public void print (int i)
public void print (float f)
public void print (String s)
```

- Java permits you to reuse a method name for more than one method.
- Two rules apply to overloaded methods:
  - Argument lists *must* differ.
  - Return types *can* be different.
- Therefore, the following is not legal:

```
public void print (int i)
public String print (int i)
```

Methods Using Variable Arguments

A variation of method overloading is when you need a method that takes any number of arguments of the same type:

```
public class Statistics {
    public float average (int x1, int x2) {}
    public float average (int x1, int x2, int x3) {}
    public float average (int x1, int x2, int x3, int x4) {}
}
```

- These three overloaded methods share the same functionality. It would be nice to collapse these methods into one method.

```
Statistics stats = new Statistics ();
float avg1 = stats.average(100, 200);
float avg2 = stats.average(100, 200, 300);
float avg3 = stats.average(100, 200, 300, 400);
```

Methods Using Variable Arguments

- Java provides a feature called *varargs* or *variable arguments*.

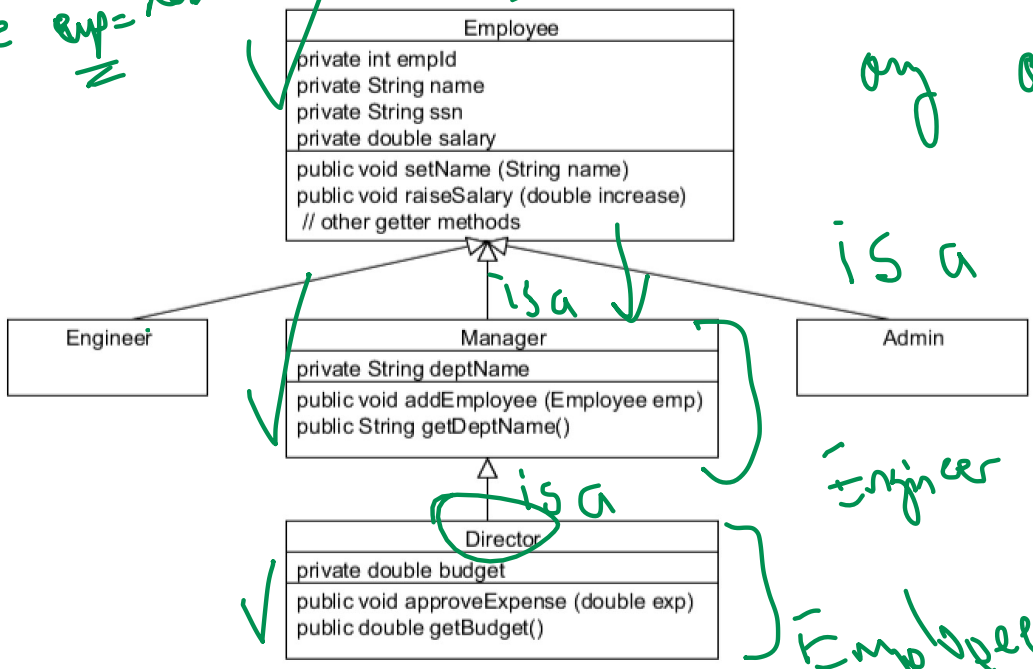
```
1 public class Statistics {
2     public float average(int... nums) {
3         int sum = 0;
4         for (int x : nums) {  // iterate int array nums
5             sum += x;
6         }
7         return ((float) sum / nums.length);
8     }
9 }
```

The varargs notation treats the `nums` parameter as an array.

- Note that the `nums` argument is actually an array object of type `int[]`. This permits the method to iterate over and allow any number of elements.

Single Inheritance

The Java programming language permits a class to extend only one other class. This is called *single inheritance*.



Using Access Control

You have seen the keywords `public` and `private`. There are four access levels that can be applied to data fields and methods. The following table illustrates access to a field or method marked with the access modifier in the left column.

Modifier (keyword)	Same Class	Same Package	Subclass in Another Package	Universe
`private`	Yes			
`default`	Yes	Yes		
`protected`	Yes	Yes	Yes *	
`public`	Yes	Yes	Yes	Yes

Classes can be default (no modifier) or `public`.

Protected Access Control: Example

```
1 package demo;
2 public class Foo {
3     protected int result = 20;
4     int other = 25;
5 }
```

subclass-friendly declaration

```
6 package test;
7 import demo.Foo;
8 public class Bar extends Foo {
9       private int sum = 10;
10       public void reportSum () {
11            sum += result;
12            sum += other;
13       }
14 }
```

compiler error

Field Shadowing: Example

```
1 package demo;
2 public class Foo2 {
3     protected int result = 20;
4 }
```

```
5 package test;
6 import demo.Foo2;
7 public class Bar2 extends Foo2 {
8     private int sum = 10;
9     private int result = 30;
10     public void reportSum() {
11         sum += result;
12     }
13 }
```

`result` field shadows the parent's field.

Access Control: Good Practice

A good practice when working with fields is to make fields as inaccessible as possible, and provide clear intent for the use of fields through methods.

```
1 package demo;
2 public class Foo3 {
3     private int result = 20;
4     protected int getResult() { return result; }
5 }
```

```
6 package test;
7 import demo.Foo3;
8 public class Bar3 extends Foo3 {
9     private int sum = 10;
10     public void reportSum() {
11         sum += getResult();
12     }
13 }
```


Overriding Methods

optional @Override

Consider a requirement to provide a String that represents some details about the `Employee` class fields.

```
1 public class Employee {
2     private int empId;
3     private String name;
4     // ... other fields and methods
5     public String getDetails () {
6         return "Employee id: " + empId +
7                " Employee name:" + name;
8     }
9 }
```

Overriding Methods

In the `Manager` class, by creating a method with the same signature as the method in the `Employee` class, you are *overriding* the `getDetails` method:

```
1 public class Manager extends Employee {
2     private String deptName;
3     // ... other fields and methods
4     public String getDetails () {
5         return super.getDetails () +
6                 " Department: " + deptName;
7     }
8 }
```

A subclass can invoke a parent method by using the `super` keyword.

Invoking an Overridden Method

- Using the previous examples of `Employee` and `Manager`:

```
Employee e = new Employee (101, "Jim Smith", "011-12-2345",
100_000.00);
Manager m = new Manager (102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
System.out.println (e.getDetails());
System.out.println (m.getDetails());
```

- The correct `getDetails` method of each class is called:

```
Employee id: 101 Employee name: Jim Smith
Employee id: 102 Employee name: Joan Kern Department: Marketing
```

Virtual Method Invocation

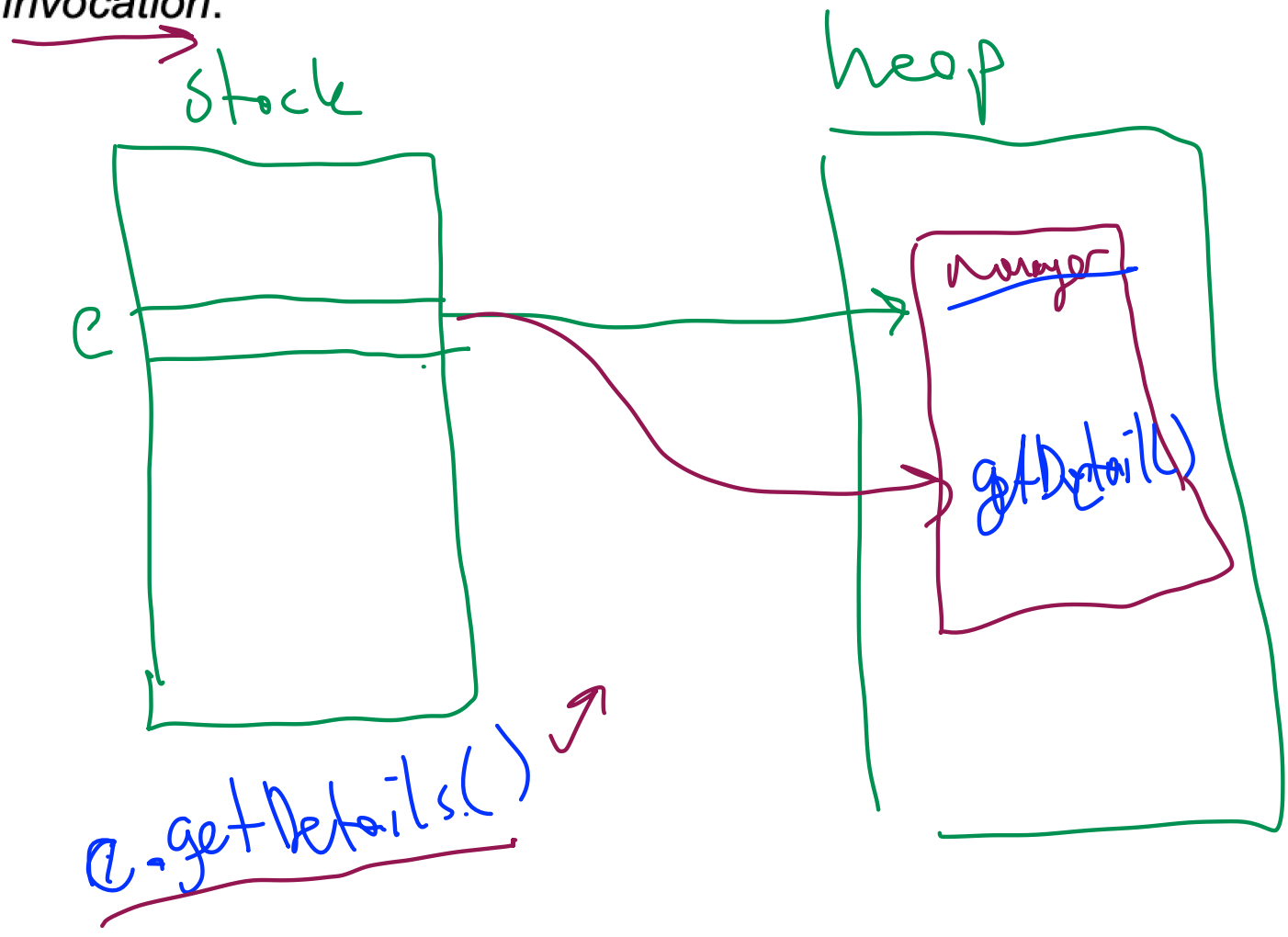
- What happens if you have the following?

```
Employee e = new Manager (102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
System.out.println (e.getDetails());
```

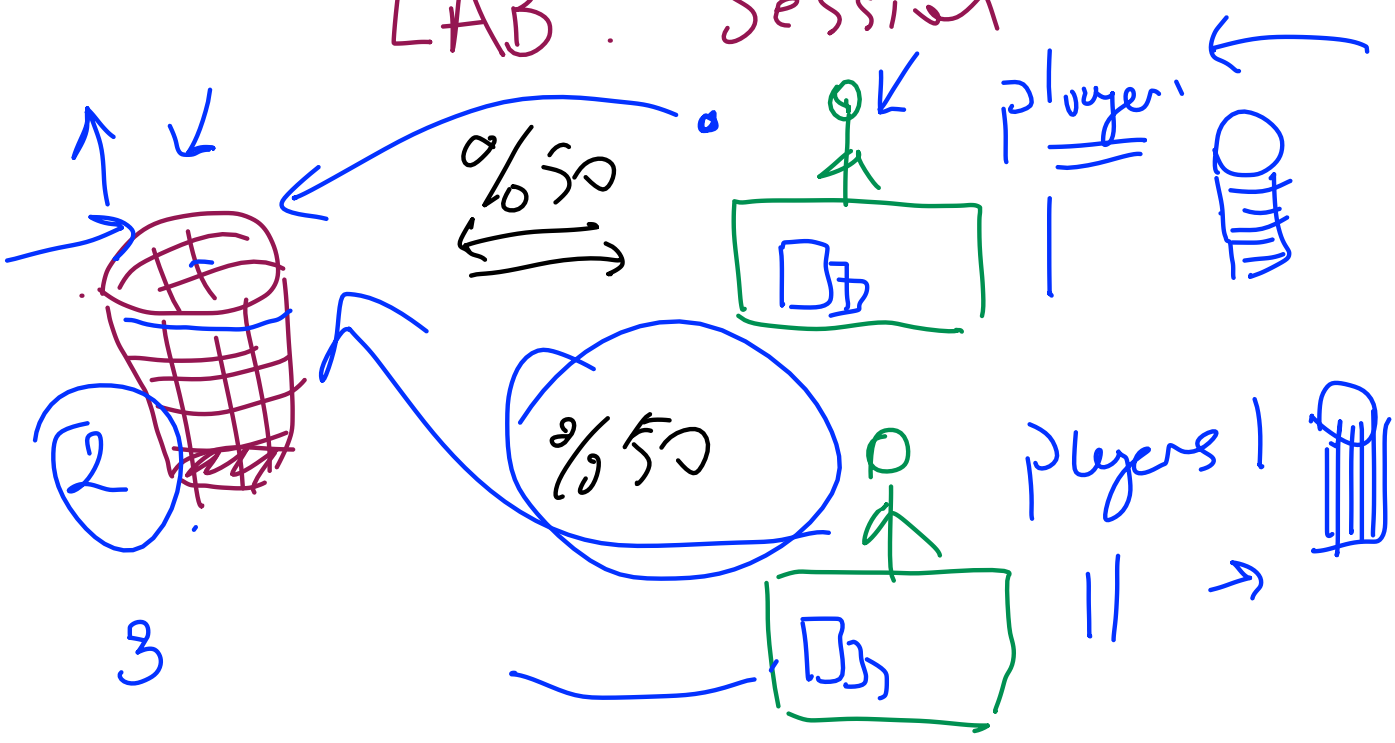
- During execution, the object's runtime type is determined to be a `Manager` object:

```
Employee id: 102 Employee name: Joan Kern Department: Marketing
```

- The compiler is satisfied because the `Employee` class has a `getDetails` method, and at run time the method that is executed is referenced from a `Manager` object.
- This is an aspect of polymorphism called *virtual method invocation*.



LAB. Session



(+) Capacity Bin (10) -this should be / can be assinged in constructor time

(+) We can add garbage

* players has none

(+) players can shoot at basket

(+) players can keep track of their scores

(f) players has %50 percent of chance of scoring. Random

(f)

2 class , 1 driver

Player

DustBin

+Capacity : int
+CurrentAmount:int

addGarbage()
emptyBin()