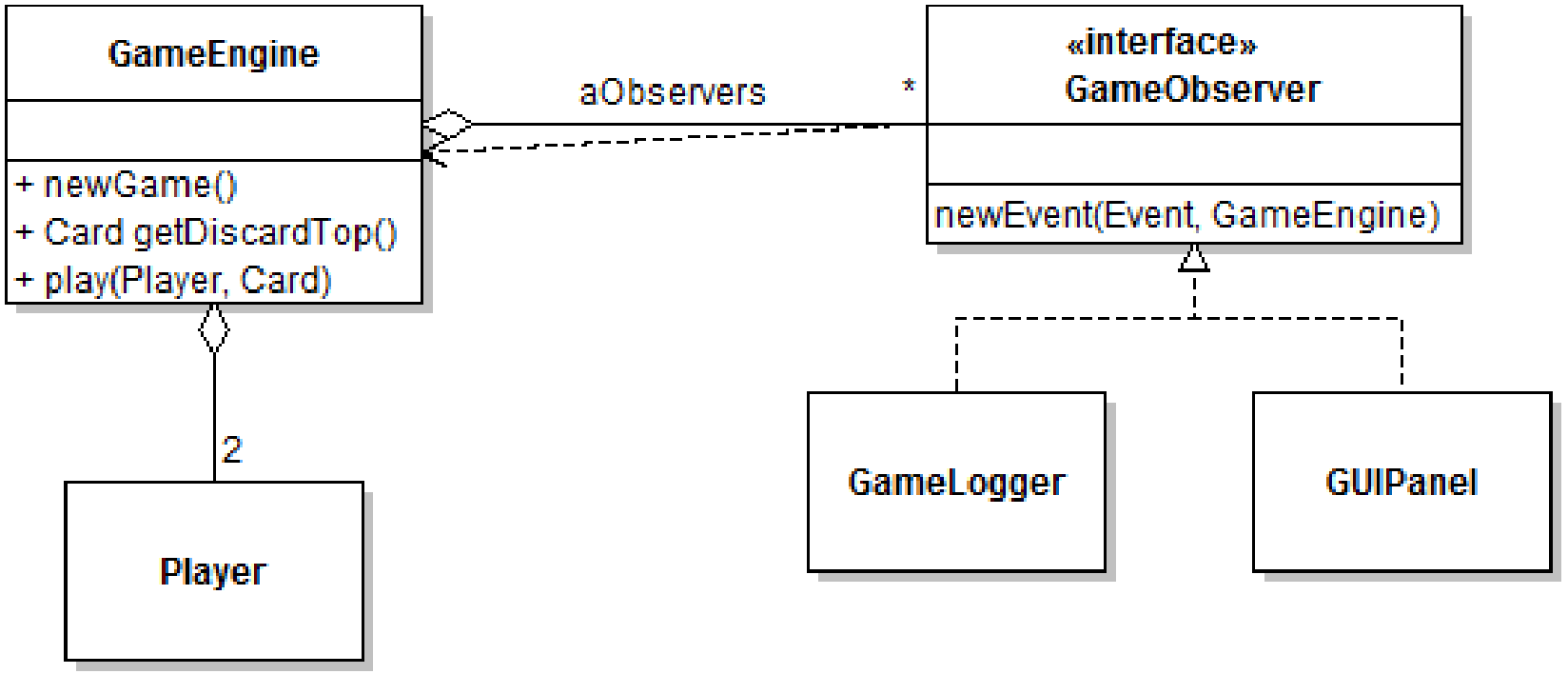
Today: Inheritance-Based Reuse

1. Quick review of the Observer Design Pattern
2. A review of inheritance in Java
3. Reasoning about method dispatch
4. Problems with Inheritance
   - Creeping exposure
   - Liskov Substitutability Principle violations
   - Identity crises
   - Compositional inheritance

Observer Review

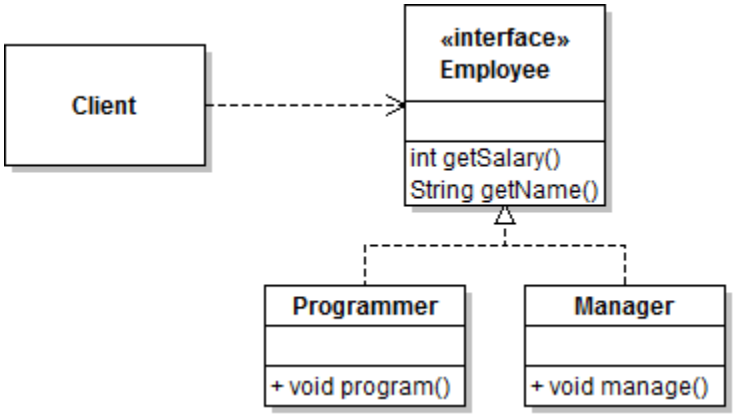


- What class is the model/subject?
- What class is the abstract observer?
- What classes are concrete observers?
- What method is the callback?
- Does this design use the push or pull data-flow technique?
- Is the ISP at play here?

Today: Inheritance-Based Reuse

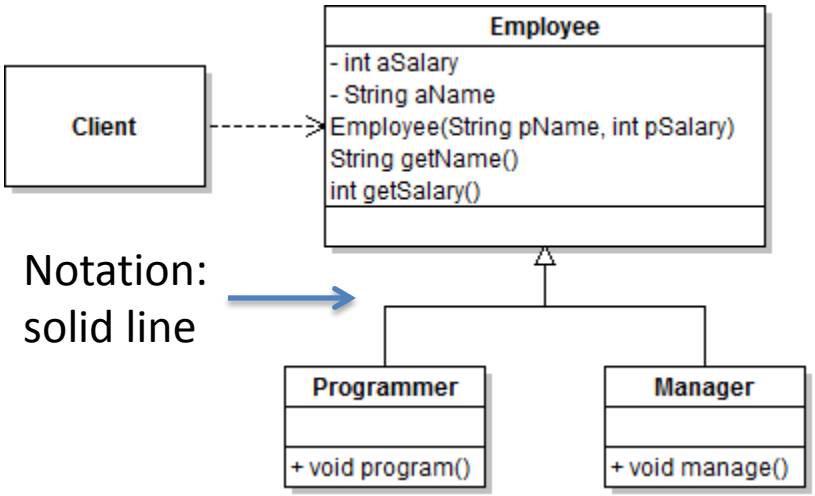
1. Quick review of the Observer Design Pattern
2. A review of inheritance in Java
3. Reasoning about method dispatch
4. Problems with Inheritance
   - Creeping exposure
   - Liskov Substitutability Principle violations
   - Identity crises
   - Compositional inheritance

Inheritance:
Subtype relation + implementation reuse




```java
public class Client
{
    // Some stuff

    public static void printName(Employee pEmployee)
    {
        System.out.println(pEmployee.getName());
    }
}
```

Inheritance:
Subtype relation + implementation reuse



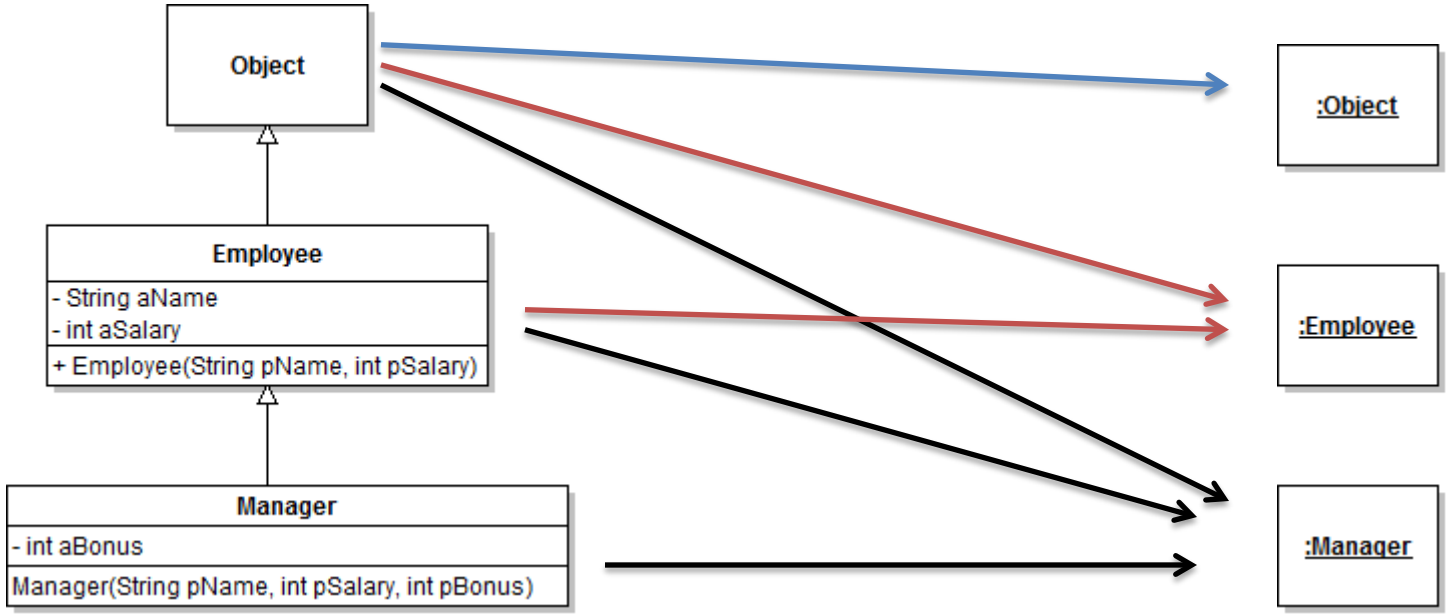
Notation:
solid line

```
class Manager extends Employee
{
```

Static vs. Dynamic

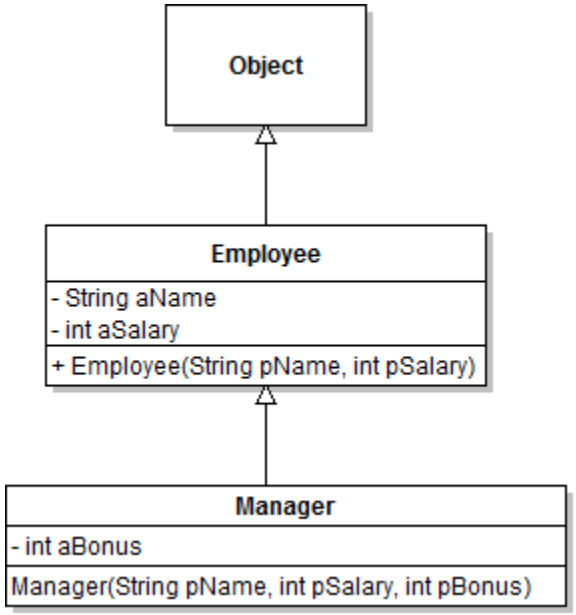
Static: A class is a template for creating objects

Dynamic: An object is built from all (transitive) super-classes

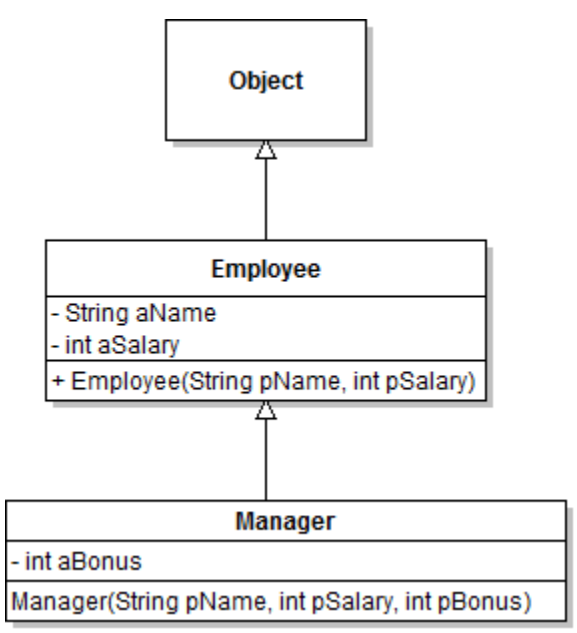


"instantiates"

Object Construction



Object Construction



Order of calls

Order of object construction

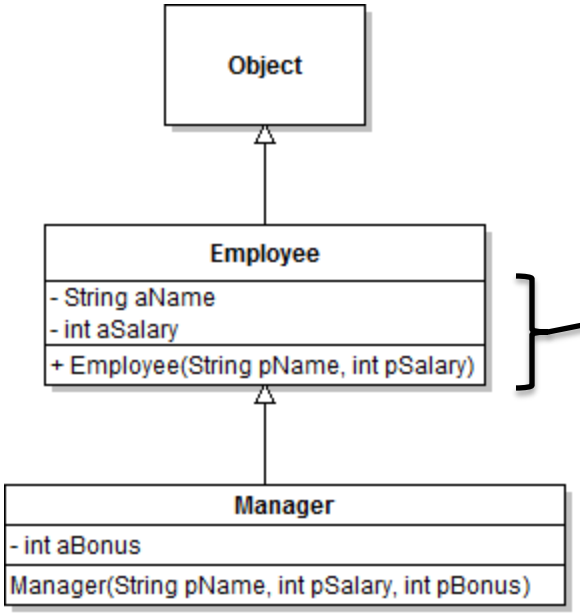
Data-flow in Object Construction

```
class Manager extends Employee
{
    private int aBonus;

    public Manager()
    {
        System.out.println("Manager default constructor");
    }

    public Manager(String pName, int pSalary, int pBonus)
    {
        super(pName, pSalary);
        aBonus = pBonus;
    }
```

Static call to (needs to be first statement)

```
public class Employee
{
    private String aName;
    private int aSalary;

    public String getName()
    {
        return aName;
    }

    public Employee()
    {
        System.out.println("Employee default constructor");
    }

    public Employee(String pName, int pSalary)
    { aName = pName; aSalary = pSalary; }
```

Access Rules



public: all classes

default: classes in the same package

private: only the class itself

protected: classes in the same package and any subclass

Today: Inheritance-Based Reuse

1. Quick review of the Observer Design Pattern
2. A review of inheritance in Java
3. Reasoning about method dispatch
4. Problems with Inheritance
   - Creeping exposure
   - Liskov Substitutability Principle violations
   - Identity crises
   - Compositional inheritance

Overloading

```
public class Overloader
{
    public static void main(String[] args)
    {
        System.out.println(new Overloader().doit("Foo"));
        String foo = "Foo";
        new Overloader().redirector(foo);
    }

    public void redirector(Object p)
    {
        System.out.println(new Overloader().doit(p));
    }

    public void doit(int p) {  }
    public String doit(long p) { return "long"; }
    public String doit(double p) { return "double"; }
    public String doit(Employee p) { return "Employee"; }
    public String doit(String p) { return "String"; }
    public String doit(Object p) { return "Object"; }
}
```

Explicit parameter

Selection Algorithm:

- Select all applicable methods based on static type;
- Choose the most specific

What will this print?

Pros and cons of overloading?

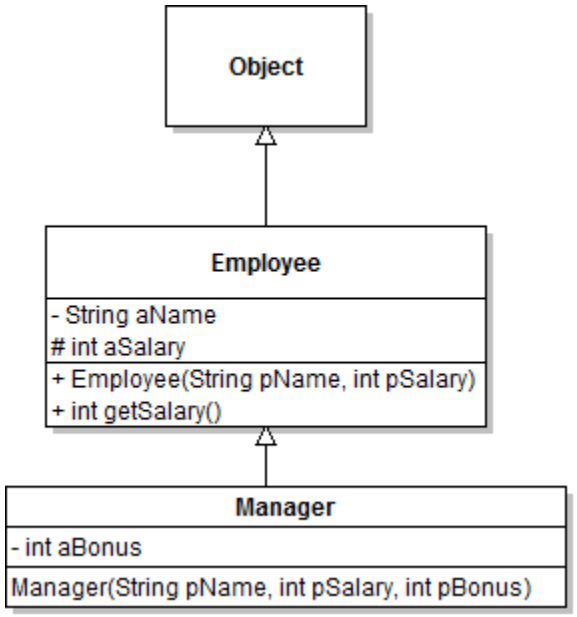
Quick Check

```
public class Confusing
{
    public Confusing( Object p )
    {
        System.out.println("Object");
    }

    public Confusing( double[] p )
    {
        System.out.println("Double array");
    }

    public static void main(String[] args)
    {
        new Confusing(null);
    }
}
```

Selection Algorithm:

- Select all applicable methods based on static type;
- Choose the most specific

Overriding

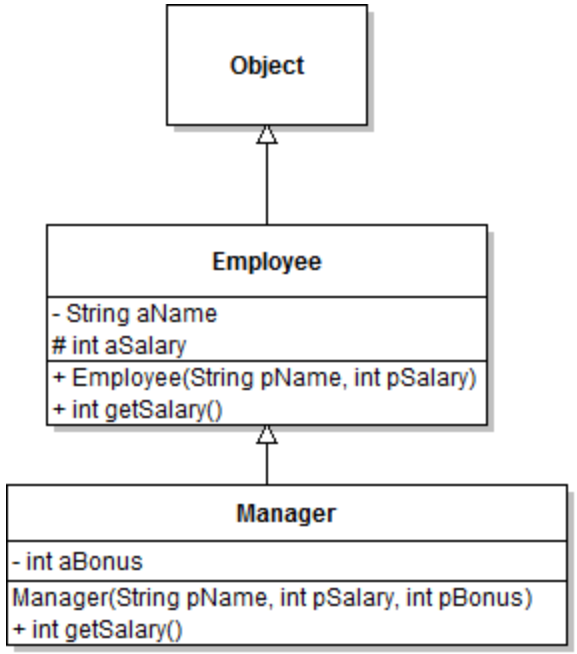



```
public class Client
{
    public static void main(String[] args)
    {
        Employee bob = new Programmer("Bob", 1000);
        Employee alice = new Manager("Alice", 2000, 500);

        printSalary(bob);
        printSalary(alice);
    }

    private static void printSalary( Employee e )
    {
        System.out.println( e.getName() + ":" + e.getSalary() );
    }
}
```

Implicit parameter

inherited

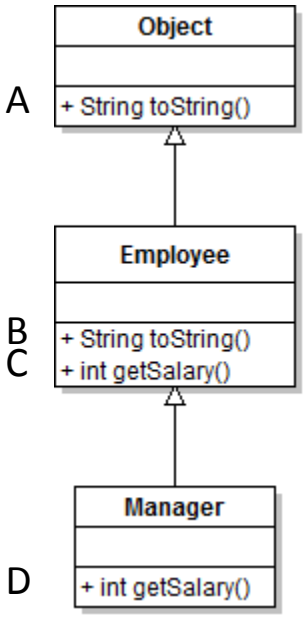
Overriding




```
public class Client
{
    public static void main(String[] args)
    {
        Employee bob = new Programmer("Bob", 1000);
        Employee alice = new Manager("Alice", 2000, 500);

        printSalary(bob);
        printSalary(alice);
    }

    private static void printSalary( Employee e )
    {
        System.out.println( e.getName() + ":" + e.getSalary() );
    }
}
```

redefined

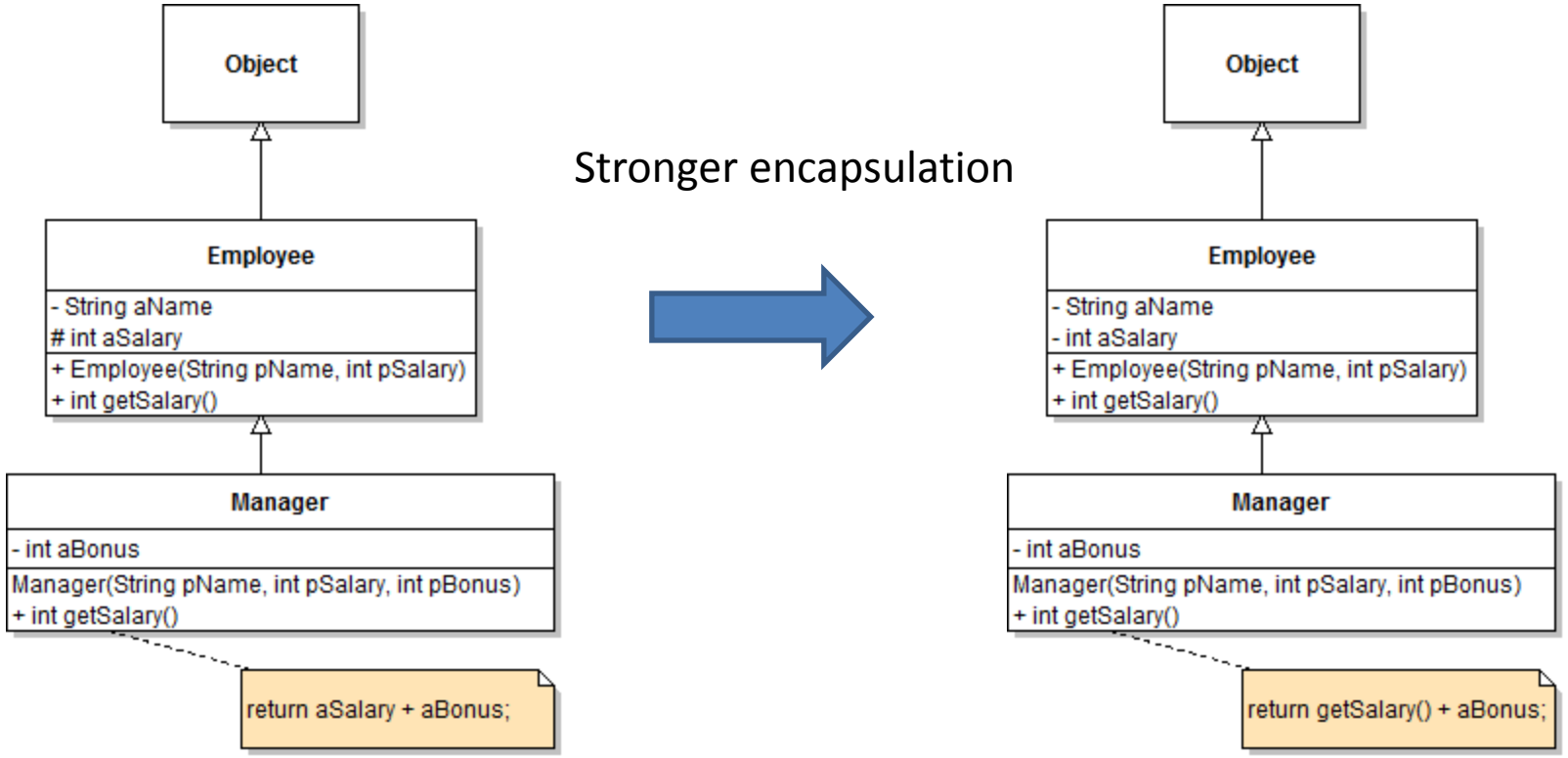
Dynamic Dispatch



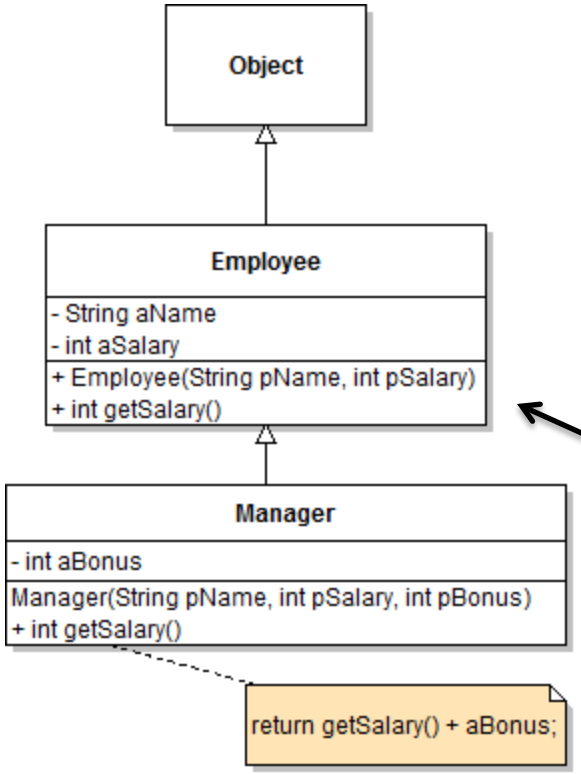
What gets called?

```
public void doit(Employee e)
{
   e.toString();
   e.getSalary();
}
```

Super Calls


Object
Employee
- String aName
int aSalary
+ Employee(String pName, int pSalary)
+ int getSalary()
Manager
- int aBonus
Manager(String pName, int pSalary, int pBonus)
+ int getSalary()
return aSalary + aBonus;
Stronger encapsulation
Object
Employee
- String aName
- int aSalary
+ Employee(String pName, int pSalary)
+ int getSalary()
Manager
- int aBonus
Manager(String pName, int pSalary, int pBonus)
+ int getSalary()
return getSalary() + aBonus;

Super Calls




```
class Manager extends Employee
{
    private int aBonus;

    public Manager(String pName, int pSalary, int pBonus)
    {
        super(pName, pSalary);
        aBonus = pBonus;
    }

    public int getSalary()
    {
        return super.getSalary() + aBonus;
    }
}
```

Static method selection

Static vs. Dynamic Binding Cheatsheet

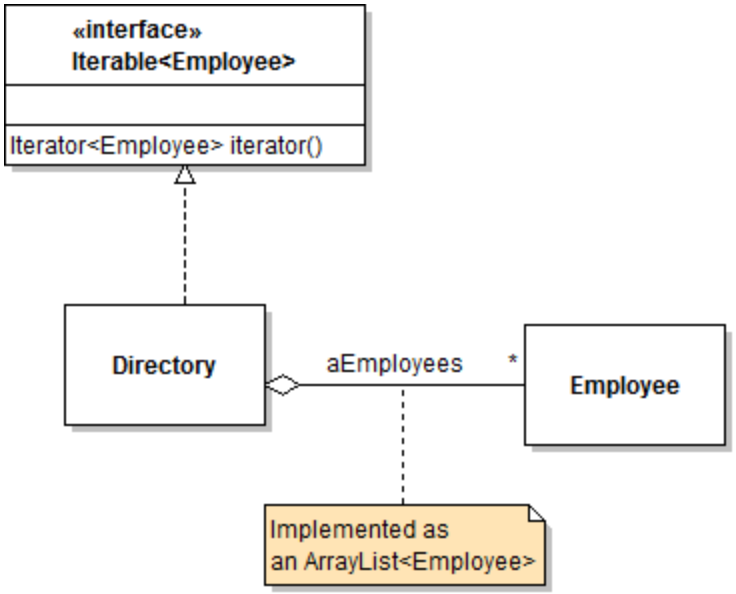
- Dynamic binding (run-time type)
  - Overriden methods
  - Interface implementations
- Static binding (static type)
  - All constructor calls
  - All static methods
  - Choice between overloaded variants
  - Super calls

Today: Inheritance-Based Reuse

1. Quick review of the Observer Design Pattern
2. A review of inheritance in Java
3. Reasoning about method dispatch
4. Problems with Inheritance
   - Creeping exposure
   - Liskov Substitutability Principle violations
   - Identity crises
   - Compositional inheritance

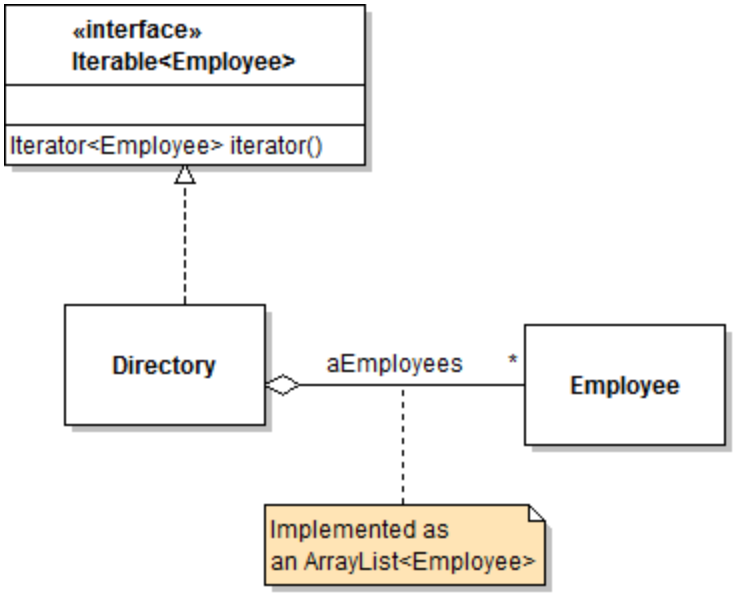
Creeping Exposure

Good Encapsulation



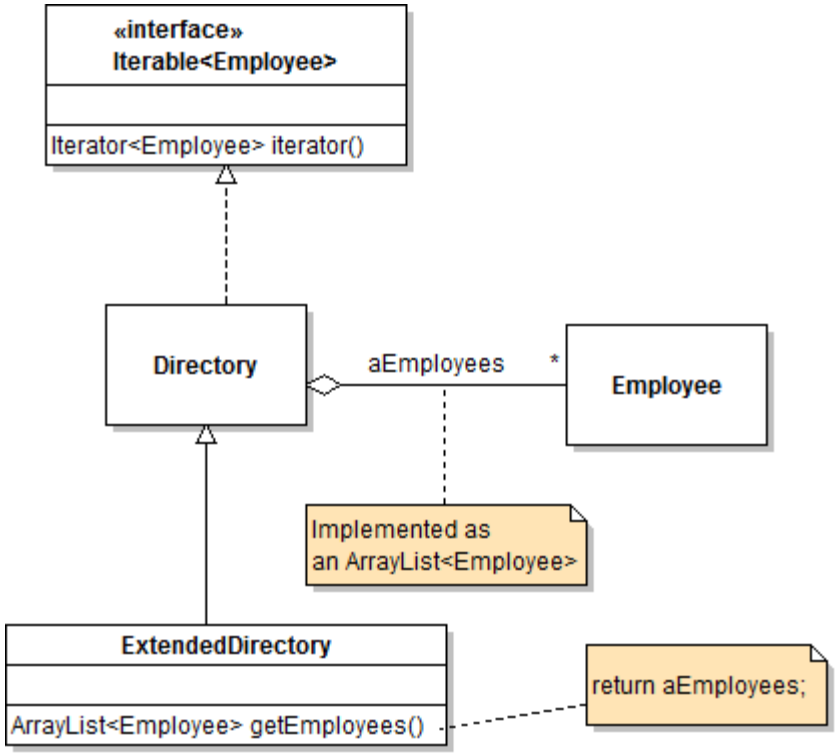
Creeping Exposure

Good Encapsulation



Solutions?

Good Grief Encapsulation



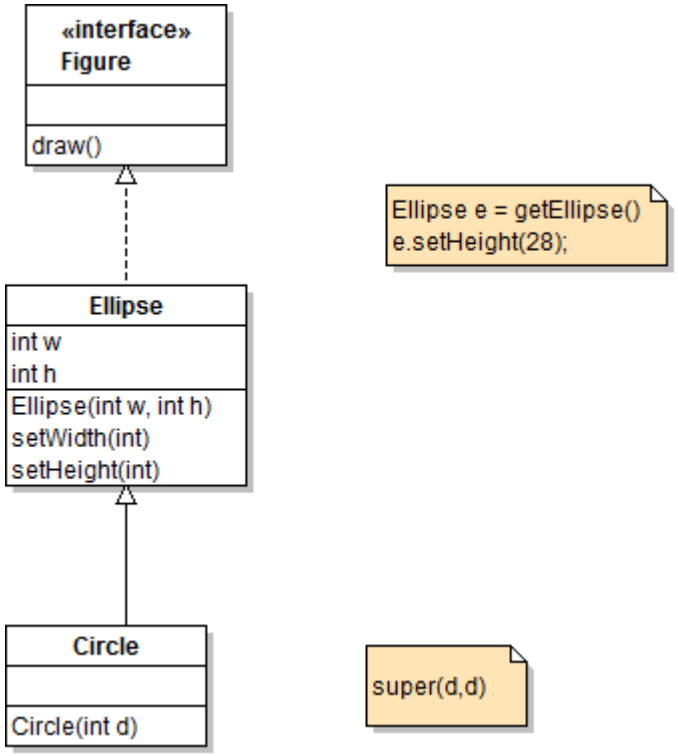
Liskov Substitutability Principle

```java
public class Employee
{
    private String aName;
    protected int aSalary;

    /**
     * @pre pSalary >= 100000
     */
    public Employee( String pName, int pSalary )
    {
        aName = pName;
        aSalary = pSalary;
    }

    public int getSalary()
    {
        return aSalary;
    }

    public int baselineIncrements()
    {
        return aSalary / 100000;
    }
}
```

```java
public class Client
{
    public static void main(String[] args)
    {
        Employee bob = new Employee("Bob", 100000);
        System.out.println(bonus(bob));
    }

    private static double bonus(Employee pEmployee)
    {
        return 10000/pEmployee.baselineIncrements();
    }
}
```

```java
public class Manager extends Employee
{
    public Manager(String pName, int pSalary)
    {
        super(pName, pSalary);
    }

    public void reduceSalary()
    {
        aSalary = aSalary - aSalary / 10;
    }

}
```

The Famous Ellipse Example



The Liskov Substitutability Principle

A method in a subclass overriding a method in a superclass **can never restrict** what the clients of the *superclass* can do. The method:

- Can't have stricter preconditions
- Can't take more specific types as parameters
- Can't make the method less accessible (e.g., public -> protected)
- Can't throw more checked exceptions
- Cant have less strict post-conditions
- Cant have a less specific return type

Identity Crisis

```java
public class Employee implements Cloneable
{
    protected String aName;

    public Employee( String pName )
    { aName = pName; }

    public String getName()
    { return aName; }

    @Override
    public boolean equals(Object pObject)
    {
        if( pObject == null ) return false;
        if( pObject == this ) return true;
        if( !(pObject instanceof Employee )) return false;
        return aName.equals(((Employee)pObject).getName());
    }

    @Override
    public int hashCode()
    {
        return aName.hashCode();
    }

    @Override
    public Employee clone()
    {
        return new Employee(getName());
    }
}
```

```java
public class Client
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee("Alice");
        Employee e2 = new Employee("Alice");
        compareThem(e1,e2);
    }

    public static void compareThem(Employee e1, Employee e2)
    {
        System.out.println(e1.equals(e2));
        System.out.println(e2.equals(e1));
        System.out.println(e2.equals(e2.clone()));
    }
}
```

What does this print?

Identity Crisis

```java
public class Employee implements Cloneable
{
    protected String aName;

    public Employee( String pName )
    { aName = pName; }

    public String getName()
    { return aName; }

    @Override
    public boolean equals(Object pObject)
    {
        if( pObject == null ) return false;
        if( pObject == this ) return true;
        if( !(pObject instanceof Employee )) return false;
        return aName.equals(((Employee)pObject).getName());
    }

    @Override
    public int hashCode()
    {
        return aName.hashCode();
    }

    @Override
    public Employee clone()
    {
        return new Employee(getName());
    }
}
```

```java
public class Client
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee("Alice");
        Employee e2 = new Employee("Alice");
        compareThem(e1,e2);
    }

    public static void compareThem(Employee e1, Employee e2)
    {
        System.out.println(e1.equals(e2));
        System.out.println(e2.equals(e1));
        System.out.println(e2.equals(e2.clone()));
    }
}
```

Manager

```java
class Manager extends Employee
{
    public Manager(String pManager)
    {
        super(pManager);
    }

    @Override
    public boolean equals(Object pObject)
    {
        if( pObject == null ) return false;
        if( pObject == this ) return true;
        if( !(pObject instanceof Manager )) return false;
        return aName.equals(((Employee)pObject).getName());
    }
}
```

Compositional Inheritance

See code Lecture 13-p08