



Enumerations

The History of Enums in Java

Enumerations

- Java has never had an enumerated type
 - Something regarded as essential and straightforward in other programming languages (even very old ones)
 - The reason for this is to do with type safety
- Enumerations tend to degenerate into integers
 - Which in C/C++ etc... can lead to horrible code
 - Implicit conversions are particularly nasty...

```
//This wont work in Java...
Enumeration Colors = {RED, GREEN, BLUE};
Color c = (Color)2;
if(c == GREEN) {
    print("Enums are assigned from 1");
} else if(c == BLUE) {
    print("Enums are assigned from 0");
}
```



Simulating Enumerations in Java

- Java developers ‘hack’ enums in two ways
 - By specifying constant values as static fields
 - By applying the typesafe enumeration pattern
- The typesafe enum pattern is the better choice
 - But it was invented late and is not widely used
 - Constant values are widely used in the core libraries
 - Occasionally strings are used rather than integers

```
public class MyShape {  
    public static final int CIRCLE = 1;  
    public static final int SQUARE = 2;  
    public static final int DIAMOND = 3;  
    public static final int OVAL = 4;  
}
```



The Typesafe Enumeration Pattern

- The typesafe enum pattern works as follows:
 - Declare a class to represent the enumeration
 - Make the default constructor private
 - For each value in the enum create a static final field
 - These are the only instances that can ever exist
 - This provides a limited set of values with type safety

```
public class MyShape {
    private MyShape() {
        super();
    }
    public static final MyShape CIRCLE = new MyShape();
    public static final MyShape SQUARE = new MyShape();
    public static final MyShape DIAMOND = new MyShape();
    public static final MyShape OVAL = new MyShape();
}
```



Enumerations in Java 1.5

- Java 1.5 integrates the typesafe pattern
 - The compiler applies it for you whenever you declare a new type using the 'enum' keyword
 - Hence 'enum Shape {CIRCLE, SQUARE}' declares a new class called 'Shape' with two instances
 - A reference of type 'Shape' can only be assigned to 'Shape.CIRCLE', 'Shape.SQUARE' or 'null'
 - For convenience a comma is allowed after the final constant e.g. 'enum Shape {CIRCLE, SQUARE, }'
- An enumeration class is still just a class
 - You can add extra members if required
 - In which case a semi-colon follows the last constant



Enumerations in Java 1.5

```
public enum Direction {
    NORTH,
    SOUTH,
    EAST,
    WEST;

    public void print() {
        if(this == WEST) {
            System.out.println("W<--");
        } else if(this == EAST) {
            System.out.println("E-->");
        } else if(this == NORTH) {
            System.out.println("N ^");
        } else {
            System.out.println("S \");
        }
    }
}
```

```
Direction d1 = Direction.NORTH;
Direction d2 = Direction.WEST;
Direction d3 = Direction.EAST;
Direction d4 = Direction.SOUTH;

d1.print();
d2.print();
d3.print();
d4.print();
```



Enumerated Types in Depth

- An 'enum' is a special kind of class declaration
 - You cannot create any further instances via 'new'
 - You cannot derive another class or enum from it
 - You are not allowed to override 'finalize'
- The compiler adds two static utility methods
 - The 'values' method returns an array containing each of the constant values declared in the enum
 - Hence the length of the array gives the number of constants
 - The 'valueOf' method converts a string to a constant
 - The match is case-sensitive so capitalization matters
 - 'IllegalArgumentException' is thrown if there is no match



Enumerated Types in Depth

```
private static void listEnumValues() {  
    System.out.println("\nValues of Direction are:");  
    for(Direction d : Direction.values()) {  
        System.out.println("\t" + d);  
    }  
    System.out.println("\nValues of Shape are:");  
    for(Shape s : Shape.values()) {  
        System.out.println("\t" + s);  
    }  
}
```

```
private static void showStringConversion() throws IOException {  
    BufferedReader console = new BufferedReader(new InputStreamReader(System.in));  
    System.out.println("\nEnter a direction:");  
    String input = console.readLine();  
    Direction d = Direction.valueOf(input);  
    d.print();  
}
```



Enumerated Types in Depth

- All enum classes extend 'java.lang.Enum' which:
 - Overrides the canonical methods of 'java.lang.Object'
 - Declares an 'ordinal' method which returns a number for the constant value it is called on
 - The first value in the enum is zero, the next one etc...

```
private static void listEnumValuesWithOrdinals() {
    System.out.println("\nValues of Direction are:");
    for(Direction d : Direction.values()) {
        System.out.println("\t" + d + " (" + d.ordinal() + ")");
    }
    System.out.println("\nValues of Shape are:");
    for(Shape s : Shape.values()) {
        System.out.println("\t" + s + " (" + s.ordinal() + ")");
    }
}
```



Enumerated Types In Depth

- Enums can have user defined constructors
 - These are automatically set as private members
- Enums can also have constant specific behaviour
 - By adding blocks of code prefixed with the constant name
 - In this case the constants are best thought of as anonymous inner classes that extend the base enum
- These features should be used sparingly
 - Most enums should be used simply as markers rather than as full objects with their own state and behaviour



User Defined Constructors

```
public enum GameType {
    SHOOTER("A 'just blast em!' type game"),
    STRATEGY("A game that involves thought"),
    SIMULATION("A game that mimics driving or flying");

    GameType(String description) {
        this.description = description;
    }
    public String description() {
        return description;
    }
    private String description;
}
```

```
private static void showUserDefinedConstructor() {
    System.out.println("\nDescriptions of game types are:");
    for(GameType g : GameType.values()) {
        System.out.println("\t" + g + "-->" + g.description());
    }
}
```



Constant Specific Behaviour

```
public enum MovieType {
    SCIFI {
        public String description() { return "A movie set in the future"; }
    },
    COMEDY {
        public String description() { return "A movie that makes you laugh"; }
    },
    HEIST {
        public String description() { return "A movie based around a crime"; }
    };
    public abstract String description();
}
```

```
private static void showConstantSpecificBehaviour() {
    System.out.println("\nDescriptions of movie types are:");
    for(MovieType m : MovieType.values()) {
        System.out.println("\t" + m + "-->" + m.description());
    }
}
```



Objects In Java Pt 2

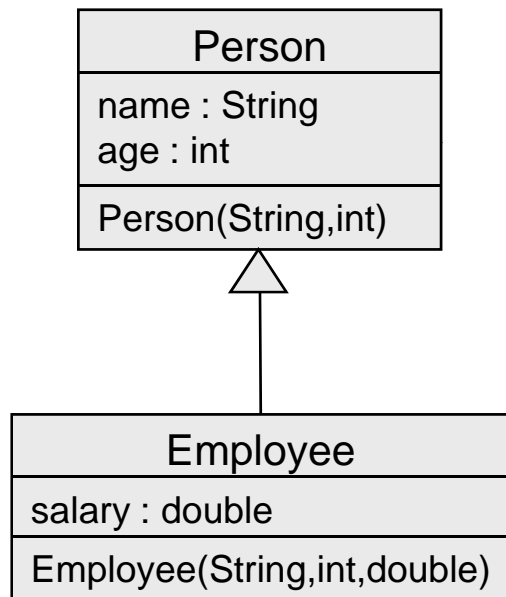
Inheritance and Polymorphism



Inheritance

- Java uses the 'extends' keyword for inheritance
 - A class can only inherit from a single base class
 - But classes can also implement interfaces...
- All classes in Java are derived classes
 - If no base is specified you extend 'java.lang.Object'
 - 'class MyClass' is equivalent to 'class MyClass extends Object'
- All fields and methods are inherited
 - Although any private methods or fields you inherit cannot be called or used by methods of the derived class
- Constructors are not inherited
 - Each class needs to have its own set of constructors
 - However a derived constructor can call a base constructor

Representing Inheritance In UML

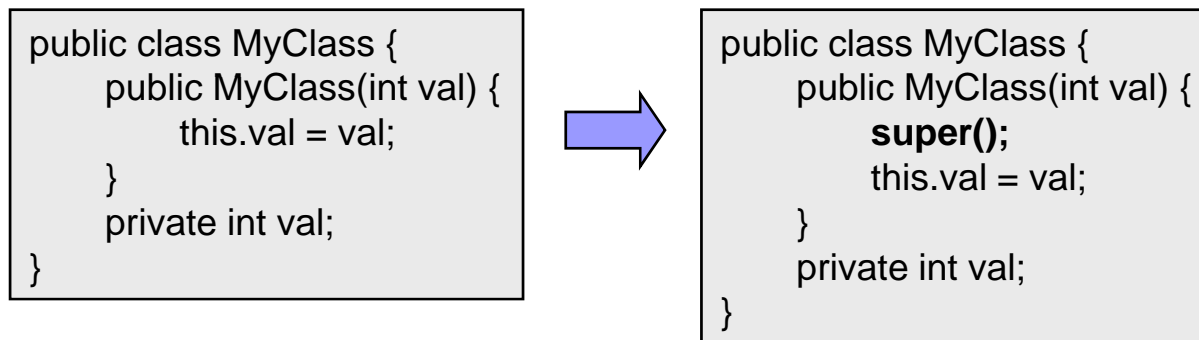


```
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```
public class Employee extends Person {
    private double salary;
    public Employee(String name, int age, double salary) {
        super(name, age);
        this.salary = salary;
    }
}
```

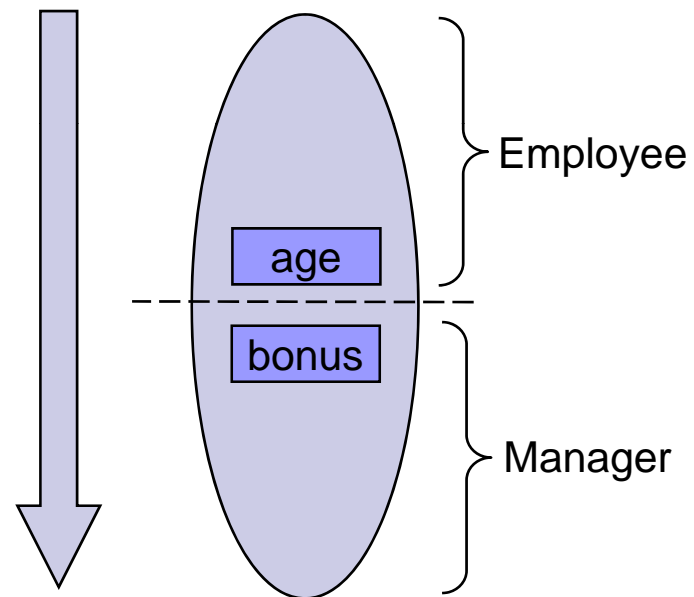
Inheritance and Constructors

- A base class constructor can be called via 'super'
 - It is good practice to let a base layer initialize itself
- Inherited fields must be initialized before derived fields
 - The first line in a constructor must call another constructor
 - Either in the current class via 'this' or in the base class via 'super'
 - If this is not the case then a call to 'super()' is added
 - This ensures initialization proceeds from the top down



Inheritance and Constructors

```
class Employee {
    Employee(int age) {
        this.age = age;
    }
    protected int age;
}
class Manager extends Employee {
    Manager(int p_age) {
        super(p_age);
        if(age > 40) {
            bonus = 2000;
        } else {
            bonus = 1000;
        }
    }
    private double bonus;
}
```

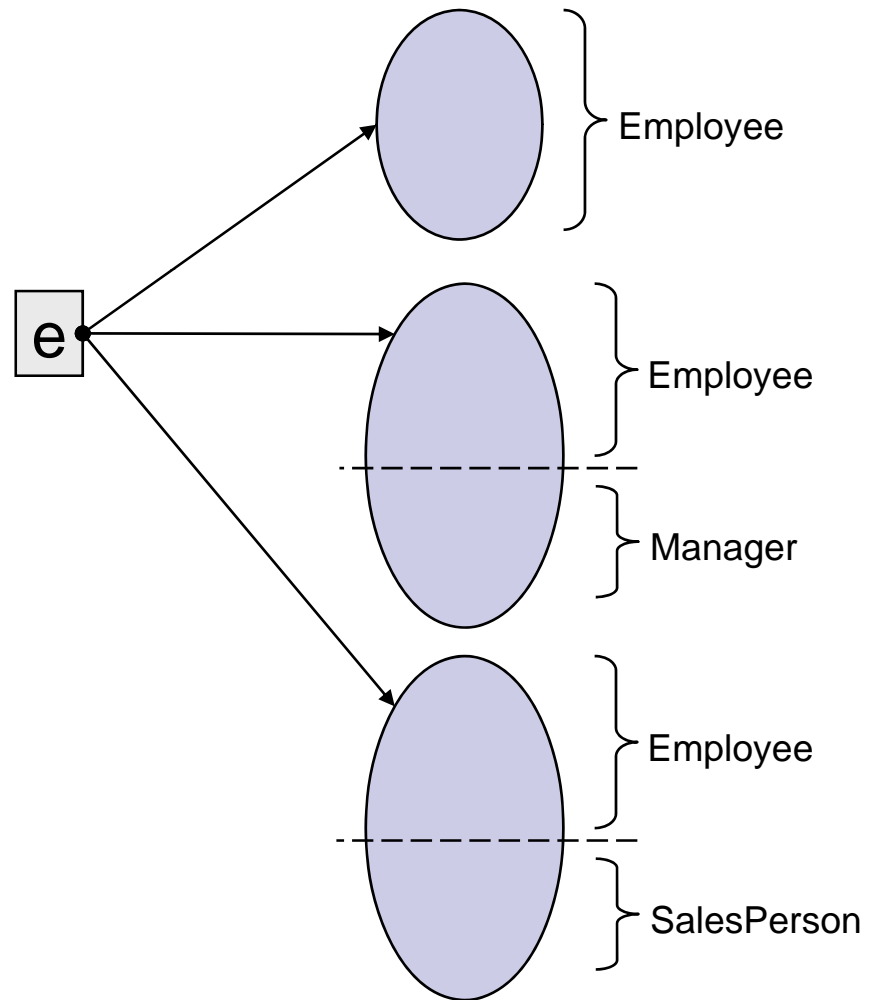
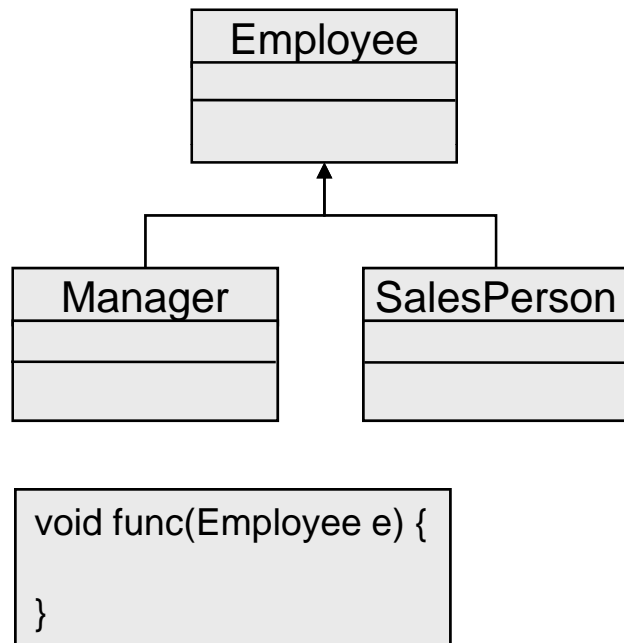




Inheritance and References

- Base class references can refer to derived objects
 - E.g. you can write 'Employee e = new Manager();'
- What you see is not always what you have
 - The type of the reference is the *apparent* type
 - The type of the object is the *inherent* or *actual* type
- This is useful when you consider parameters
 - A method that takes an 'Employee' parameter can be passed a 'Manager' or a 'Developer' etc...
 - The compiler limits your access to the slots declared on the layer corresponding to the apparent type (and above)

Inheritance and References





Overriding Methods

- A class may override inherited methods
 - The best metaphor is that the slot created by the base class is *rewired* into the derived implementation
- Overriding is a very powerful concept
 - It allows clients to work with groups of objects without caring about their actual type (via lists of references of the base type)
 - E.g. A 'Window' object can contain an array of 'Widget' references and call 'display' on each one as required
 - All GUI classes (buttons, textboxes etc...) inherit from 'Widget' and override methods like 'display' to render themselves appropriately
 - This is in essence how most OO user interface libraries work



Inheritance and Overriding

```
class Employee {
    void downsize() {
        System.out.println("Employee fired");
    }
}
class Manager extends Employee {
    void downsize() {
        System.out.println("Manager fired");
    }
}
class SalesPerson extends Employee {
    void downsize() {
        System.out.println("Salesman fired");
    }
}
```

```
public class TestOverriding {

    public static void main(String[] args) {

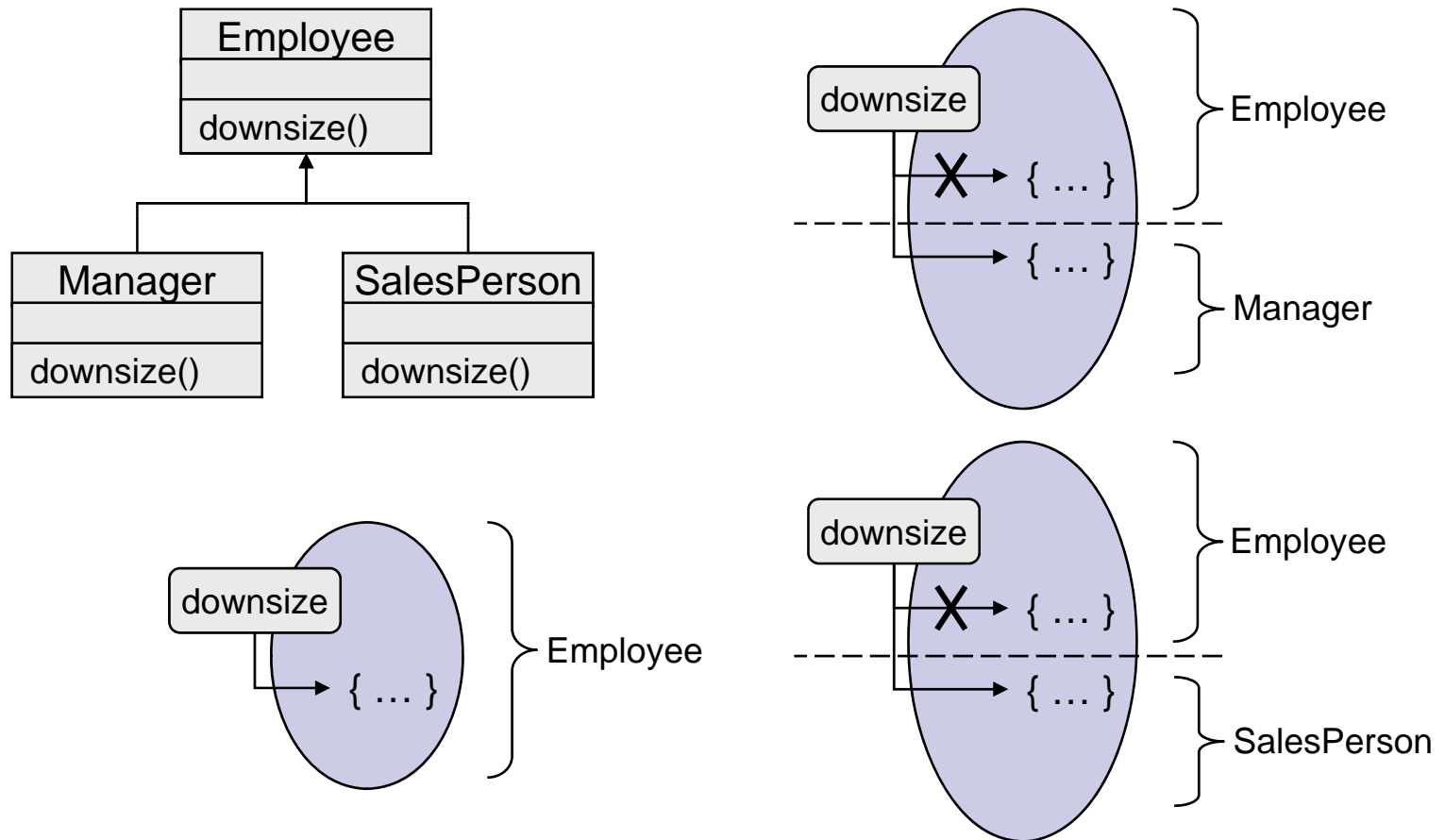
        Employee e1 = new Employee();
        Employee e2 = new Manager();
        Employee e3 = new SalesPerson();

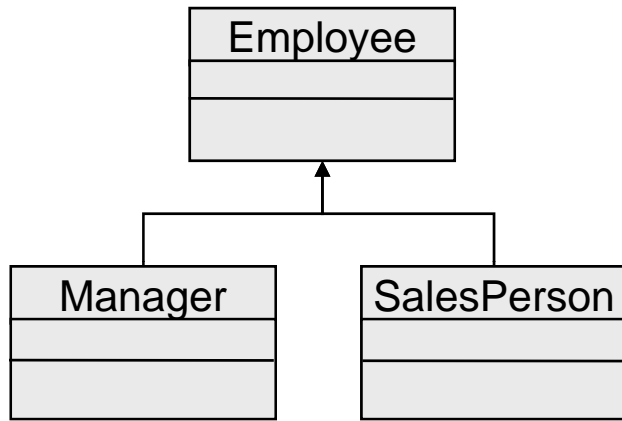
        //Calls Employee.downsize()
        e1.downsize();
        //Calls Manager.downsize()
        e2.downsize();
        //Calls SalesPerson.downsize()
        e3.downsize();

    }

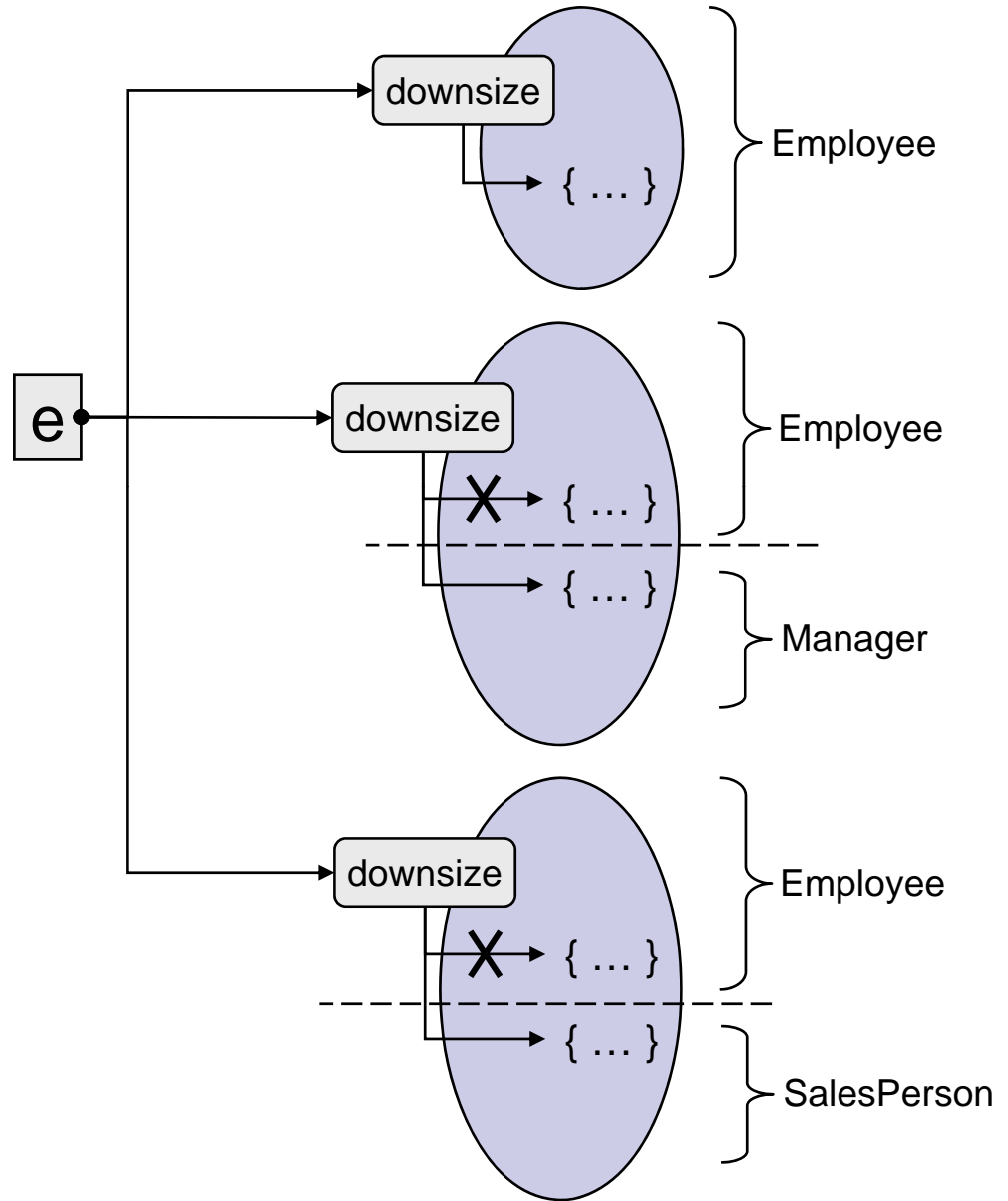
}
```

Overriding Methods

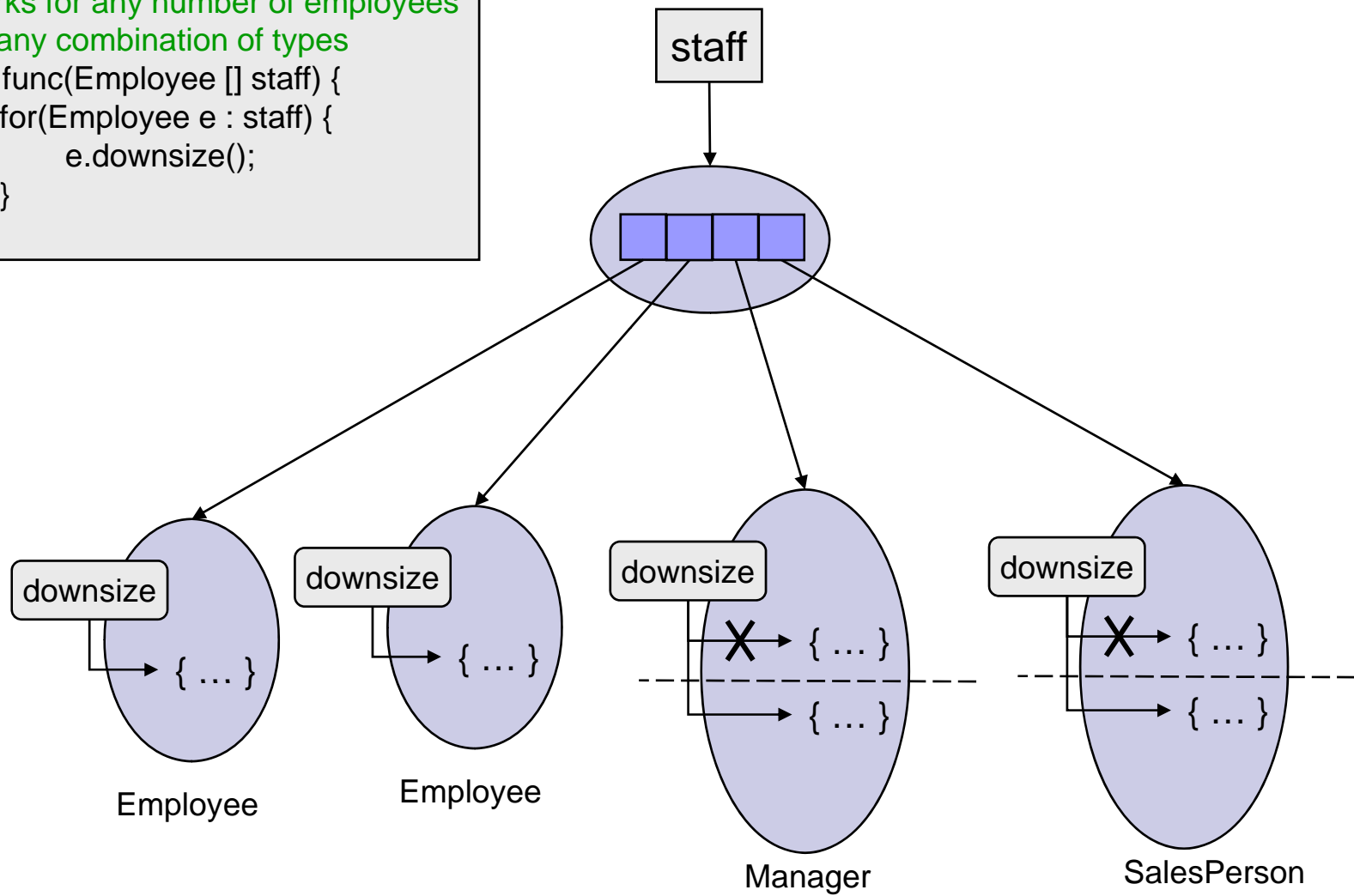




```
void func(Employee e) {
    e.downsize();
}
```



```
//works for any number of employees
// in any combination of types
void func(Employee [] staff) {
  for(Employee e : staff) {
    e.downsize();
  }
}
```





Using 'instanceof'

- The 'instanceof' keyword defines an operator
 - A reference to an object goes on the lhs
 - The name of a class is placed on the rhs
 - The operator returns true if the reference refers to an instance of the class, or a class that inherits from the specified class
- Always use 'instanceof' before a downcast
 - Casting a base reference to a derived class type
 - Unless you are sure of the actual type (e.g. Collections)
- Downcasting usually indicates a problem in your design
 - Using polymorphism should solve the problem

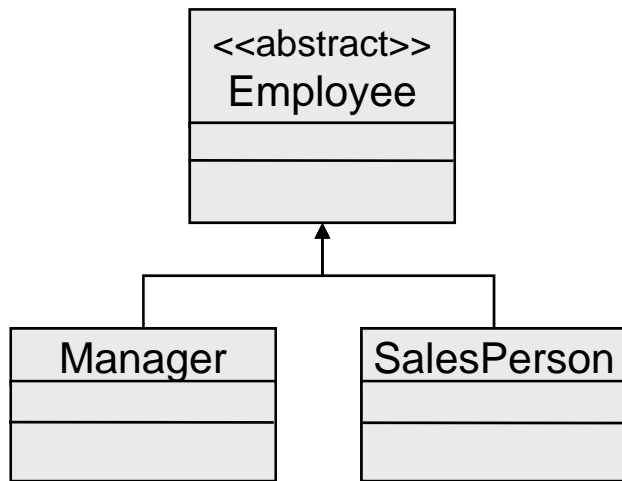
Objects in Java Pt 3

Special Class Types



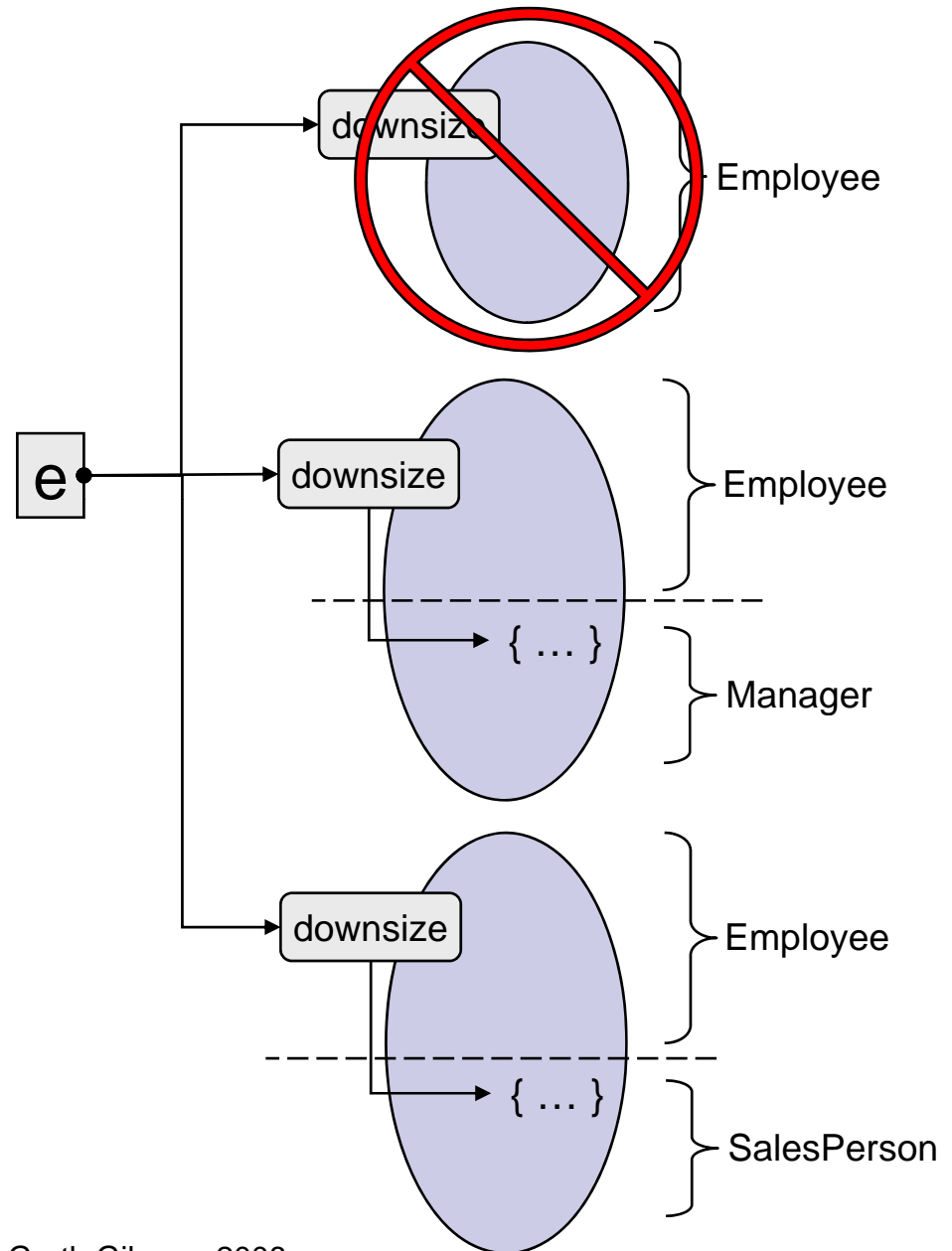
An Abstract Class

- A class may be declared as 'abstract'
 - Abstract classes cannot be instantiated
 - They are used as base classes in situations where no instance of the base type ever exists in the problem domain
 - For example there will be no 'Employee' objects in a system where all employees are managers, developers or salespersons
- Abstract classes usually have abstract methods
 - These are declared but not defined
 - For example 'public abstract void promote();'
 - Abstract methods must be overridden in derived classes
 - Unless the derived class is itself declared abstract



```

void func(Employee e) {
    e.downsize();
}
  
```





Declaring Symbols As Final

- Java uses the ‘final’ keyword to limit inheritance:
 - A final class cannot be inherited from
 - This is usually done for security reasons
 - E.g. to prevent clients accessing protected fields
 - A final method cannot be overridden
 - This is where the base definition is always correct
 - You cannot declare a method with the same signature in the derived class (C++ allows this and creates a new slot...)
 - A final field cannot be changed
 - Its value must be set when the object is created
 - A field which can be final should be final



Interfaces

- C++ has the concept of ‘virtual’ classes
 - Classes which contain only pure virtual methods
- Java formalizes this idea with interfaces
 - A separate type containing method declarations and constants
 - Interface methods are automatically public and abstract
 - Interface fields are automatically public static and final
- An interface describes a set of services a class can offer
 - A class can implement any number of interfaces
- Interface fields don't have to have hardcoded values
 - They can be initialized by an expression or function call when the interface is loaded, as happens with static class fields



Interfaces

```
package javax.transaction;

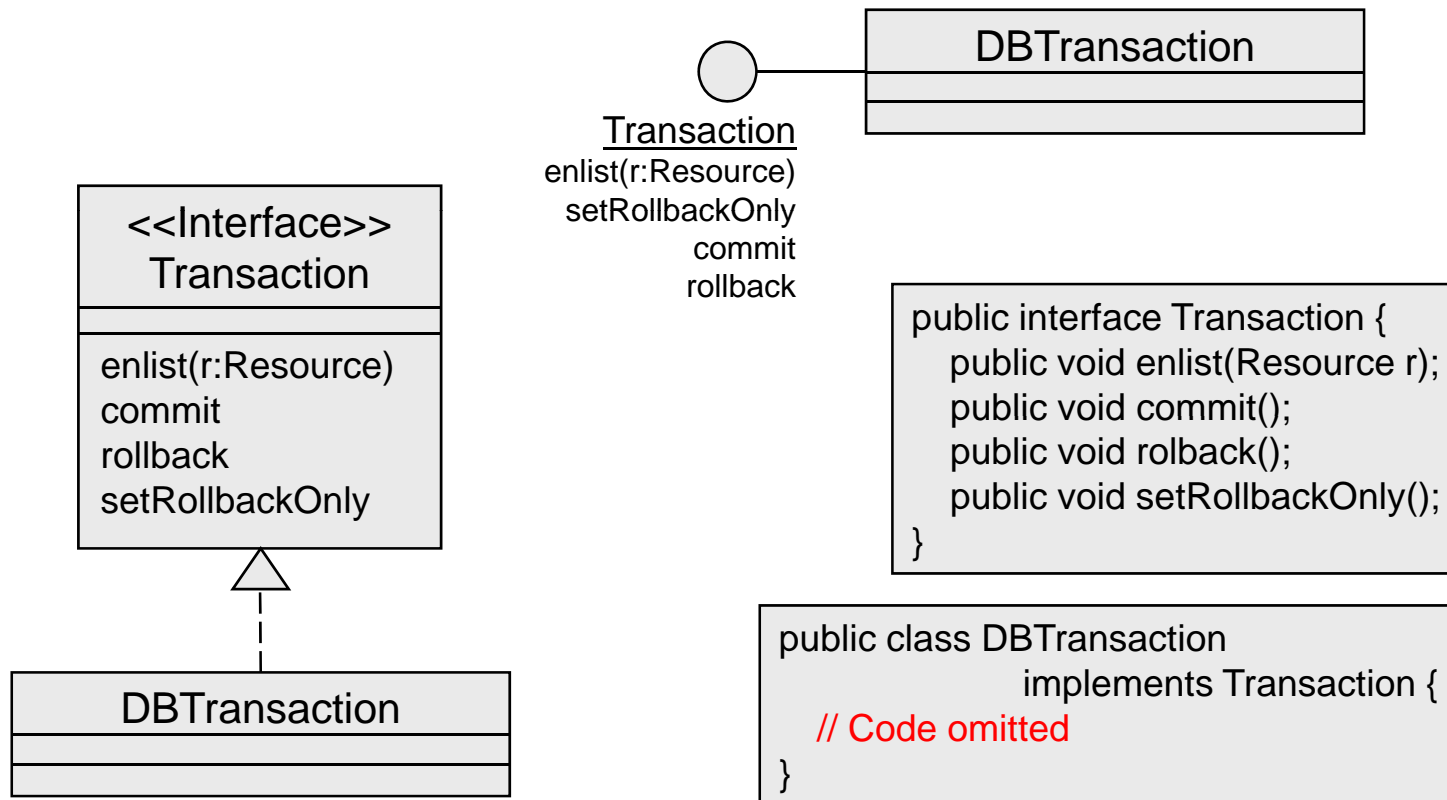
//A sample interface used in J2EE applications
public interface Transaction {
    void commit();
    boolean delistResource(XAResource xaRes, int flag);
    boolean enlistResource(XAResource xaRes);
    int getStatus();
    void registerSynchronization(Synchronization sync);
    void rollback();
    void setRollbackOnly();
}
```



Interfaces

- Interfaces are heavily used by library designers
 - They are one of the secrets of Java's success
- Most J2EE standards are made up of:
 - A large number of interfaces
 - To be implemented appropriately by different vendors
 - In a manner transparent to client code
 - A small number of concrete classes
 - Typically utility classes whose implementation remains the same
 - A few exception classes
 - To model error types specific to the standard
- Interfaces are more flexible than abstract classes
 - If in doubt prefer to use an interface rather than a base class


Representing An Interface In UML





Valid Types in Interfaces

- Methods are the most common declarations in interfaces
 - They are automatically public and abstract
- Field declarations are often added
 - These are automatically public, final and static
 - The intention is that to make the methods easier to use
 - Interface fields don't have to be compile-time values
 - Which many people find surprising
- Very rarely inner classes are used
 - Classes placed in an interface are automatically public and static
 - Hence they are best described as nested classes
 - We will meet these in detail in the next chapter



```
public interface Shop {
    class Item {
        public Item(double price, String catalogId) {
            this.price = price;
            this.catalogId = catalogId;
        }
        public double getPrice() {
            return price;
        }
        public String getCatalogId() {
            return catalogId;
        }
        private double price;
        private String catalogId;
    }

    double MIN_DISCOUNT_AMOUNT = 75.50;
    double MIN_PURCHASE_AMOUNT = ShopUtilities.minAmount();

    boolean makePurchase(Item [] items, CreditCard card);
    boolean makePurchase(Item [] items, Customer customer);
}
```



The For-Each Loop and Interfaces

- A for-each loop can be used with any type that implements the new 'Iterable' interface
 - 'Iterable' defines a single method called 'iterator' which returns an implementation of the standard 'Iterator' interface

```
public class CustomList implements Iterable {
    public CustomList() { values = new int[] {101,202,303,404}; }
    public Iterator iterator() { return new Customlterator(); }

    private class Customlterator implements Iterator {
        public boolean hasNext() { return position < values.length; }
        public Object next() { return values[position++]; }
        public void remove() { //DO NOTHING... }
        private int position;
    }
    private int [] values;
}
```



Inner Classes

- A class with multiple jobs to perform can encapsulate the secondary tasks in nested classes
 - Introduced in Java 1.1 to simplify event handling in GUI's
 - Inner classes count as a separate class
 - They compile to OuterClass\$InnerClass.class
- Inner classes have some special features
 - They may have any accessibility (mostly private)
 - They can access any fields or methods of the enclosing object
 - They always contain a reference to the enclosing object
 - You can access this via the syntax 'OuterClassName.this'

```
public class LinkedList implements List {
    private class MyIterator implements Iterator {
        MyIterator() {
            position = first;
        }
        public boolean hasNext() {
            return position != null;
        }
        public Object next() {
            if(position == null) {
                throw new NoSuchElementException();
            }
            Object item = position.getItem();
            position = position.getNext();
            return item;
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
        private Node position;
    }
    public Iterator iterator() { return new MyIterator(); }
    private Node first;
}
```

```
void printList(List list) {
    Iterator iter = list.iterator();
    while(iter.hasNext()) {
        System.out.println(iter.next());
    }
}
```



Nested (Static Inner) Classes

- An inner class can be declared as static
 - In which case it is referred to as a nested class
- Nested classes don't have an enclosing object
 - Hence they can be created directly in static methods
 - They still have permission to access all the members of the enclosing class, but only static fields are available
 - Unless the nested object obtains an instance of the enclosing class
- Nested classes can be used in interfaces
 - An inner class declared in an interface is automatically static
 - These are rarely used and hence frequently overlooked



Nested (Static Inner) Classes

```
class LinkedList implements List {
    private static class Node {
        Node(Object value, Node other) {
            this.payload = value;
            previous = other;
        }

        //get and set methods omitted ...

        private Node next;
        private Node previous;
        private Object payload;
    }
    private int size;
    private Node first;
}
```



Anonymous Inner Classes

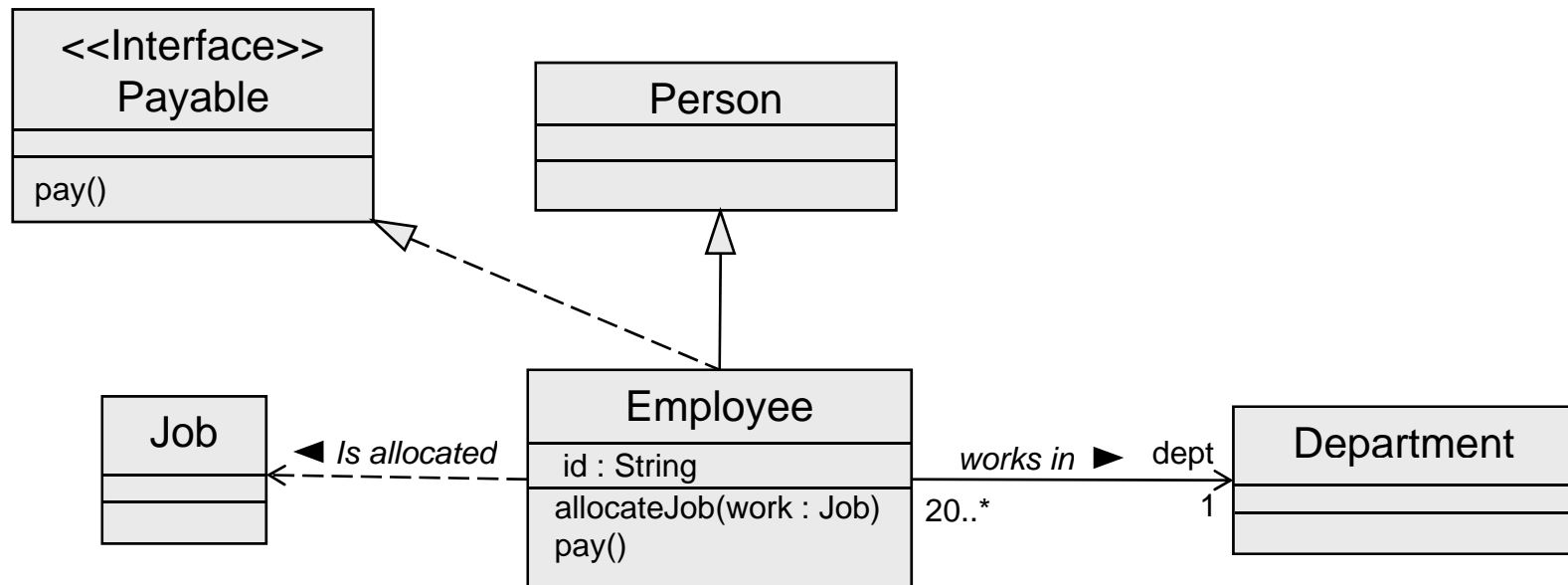
- Anonymous classes represent a ‘one-off’ inner class
 - They are used where only one instance of an inner class would be required, usually at a single point in the code
 - The declaration and instantiation of a new class are combined
 - The syntax of anonymous classes takes time to appreciate
- Anonymous classes are just a special type of inner class
 - Mostly used to quickly generate event handlers and loggers
 - They have the same special features and access rights
 - Their class files are named Outer\$1.class Outer\$2.class etc...
- Initialization blocks were created for anonymous classes
 - As an alternative to the default constructor



Anonymous Inner Classes

```
// Create an anonymous inner class that extends java.lang.Thread
// Note the use of the superclass constructor and initializer block
Thread obj = new Thread("MyThread") {
    {
        setPriority(MIN_PRIORITY);
    }
    public void run() {
        // Code to run in separate
        // thread goes here
    }
    public String toString() {
        return "Anonymous thread";
    }
};
```

Detailed Relationships Summary



```
public class Employee extends Person implements Payable {  
    private Department dept;  
    private String id;  
    public void allocateJob(Job work) { ... }  
    public void pay() { ... }  
}
```