Inheritance
Ch 6

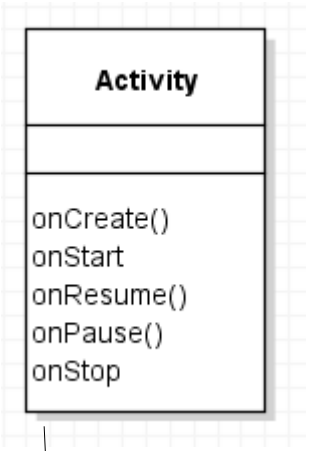
Topics

1) How can Java work with class inheritance?
    1) Creating subclasses
    2) Overriding methods
    3) Flexible Classes
    4) Visibility

Using Inheritance for Subclasses

Android Activities Intro

- An Android Activity
  - A screen in an Android app
- Activity class
  - Android framework provides an Activity base class to manage much of the Activity's work
  - Functions implement default behaviour for many event such as pausing, or showing a menu.



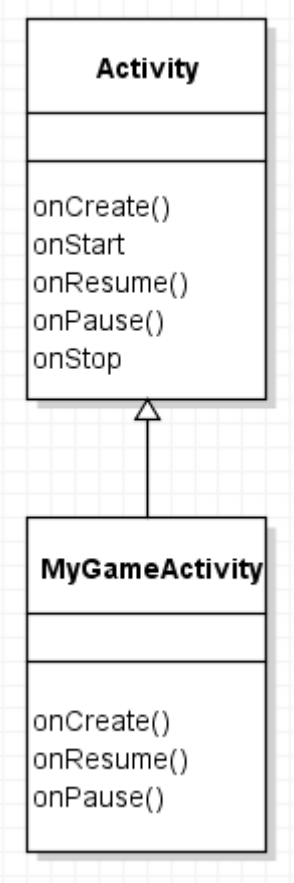
onPause(): Save data & stop animations

onResume():
Start animations

Inheritance

- Inheritance:
  creates the “is-a” relationship between classes
  - Ex: MyGameActivity is-an Activity

    MyGameActivity inherits from Activity
    (*subclass*) (*superclass*)
    (*derived*) (*base*)

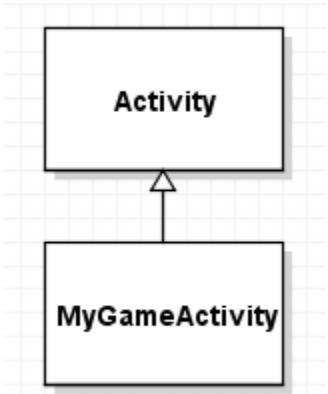


- Motivation:
  - API & implementation of the base are inherited by the derived.
  - Reuse code from base class in derived class.
  - ..allows polymorphism between objects

Notes on Inheritance

- Instantiating MyGameActivity..
  does not also instantiate an Activity object

  but base class constructor will be called first before derived class constructor

  - MyGameActivity object has all members from:
    - the Activity class (its superclass), and
    - the MyGameActivity class

- Access:

  when instantiating a derived object, its constructors may call the base class constructor first then initialize itself (only base class constructors were called), so the base object was not instantiated

  - Subclass may call/access.. of super class.

    non private members

    Ex: MyGameActivity code can call protected and public functions in Activity.

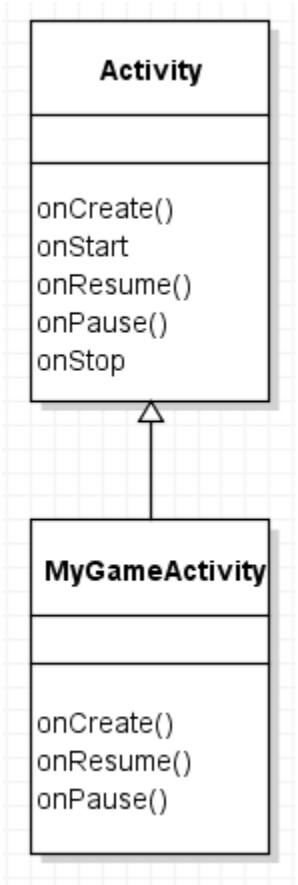
  - Base class cannot access members of derived class.

Polymorphism via Class Inheritance

- Polymorphic references can refer to an object of its class, or any derived class:

```
Activity a = new MyGameActivity();
a.onCreate();

// Reference to derived class
a = new MySettingsActivity();
a.onCreate();
```



In Android, you never call onCreate();
the Android Framework does it for you.

*watch video again

Overriding Methods

(Not over**loading**, over**riding**)

overloading: methods with same name but different type/# of parameters

super & this

- super: refers to.. superclass (not an object)
- this: refers to current object, not superclass.

Overriding

- Subclass can override a method of superclass if same signature as base:
  - Same name
  - Same argument # and types

```
public static void main(String[] args) {
        Fruit apple = new Fruit("Apple");
        System.out.println(apple.getType());

        Fruit deluxe = new DeluxeFruit("Apple");
        System.out.println(deluxe.getType());
}
```

```
Apple
Deluxe Apple
```

```
public class Fruit {
        private String type;

        public Fruit(String type) {
                this.type = type;
        }
        public String getType() {
                return type;
        }
}

public class DeluxeFruit extends Fruit {
        public DeluxeFruit(String type) {
                super(type);
        }
```

constructor chaining
derived class must call base class's constructor

```
        @Override
        public String getType() {
                return "Deluxe " + super.getType();
        }
}
```

Overriding Details

- To override a method, derived class's method must:
  - Have identical signature
  - Not throw any extra checked exceptions (more later)
  - ..not reduce visibility of overriden method
    - Ex: Can go from protected to public, but not public to protected/private.
  - Cannot override a private, a static, or a final method.
  - Not change return type of method.
    - But you can return a subtype of original return type

Base Class Constructor Chaining

- Subclass's constructor can “call” superclass constructor:

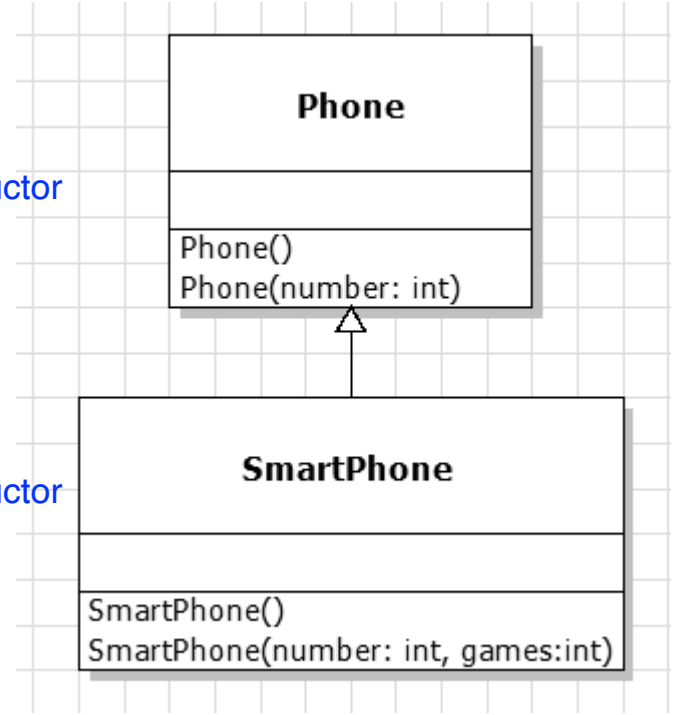
```
public class SmartPhone extends Phone {
    int numGames = 0;

    public SmartPhone () {
        super();
    }
    public SmartPhone (int number, int games) {
        super(number);
        numGames = games;
    }
}
```

call 0 arg. superclass constructor

call 1 arg. superclass constructor

if remove this line,
java automatically calls the 0 arg constructor for us



- super() must be the.. first line of the constructor
  - If missing, super(); automatically added as first line (unless using constructor chaining via this(...))

??????

* watch video again

Chaining Constructors

- How does each of these constructors work?

```
public class Base {
  private int count = 0;

  public Base() {
    this(5);
    // Do anything...
  }

  public Base(int count) {
    this.count = count;
    // Do anything...
  }
}
```

```
public class Derived extends Base {
  private final double DEFAULT = 42.0;
  private double other;

  public Derived(int count) {
    this(count, DEFAULT);
    // Do anything...
  }

  public Derived(int count, double other){
    super(count);
    this.other = other;
    // Do anything...
  }
}
```

???????

final vs Overriding

- final method:.. cannot be overriden

```
class MCHammer {
    final String getSaying() {
        return "Can't touch this!";
    }
}

class MCWho extends MCHammer{
    @Override
    String getSaying() {
        return "Who's MC Hammer?";
    }
}
```

error

- final class:.. cannot be extended

Shadow Variables - a Bad Idea

- Shadow Variables:
  - Subclass declares a variable of the same name as the superclass

```
class Pet {
        private String name;
        // ...
}

class PetRock extends Pet {
        private String name;
        // ...
}
```

- this does not change the base class

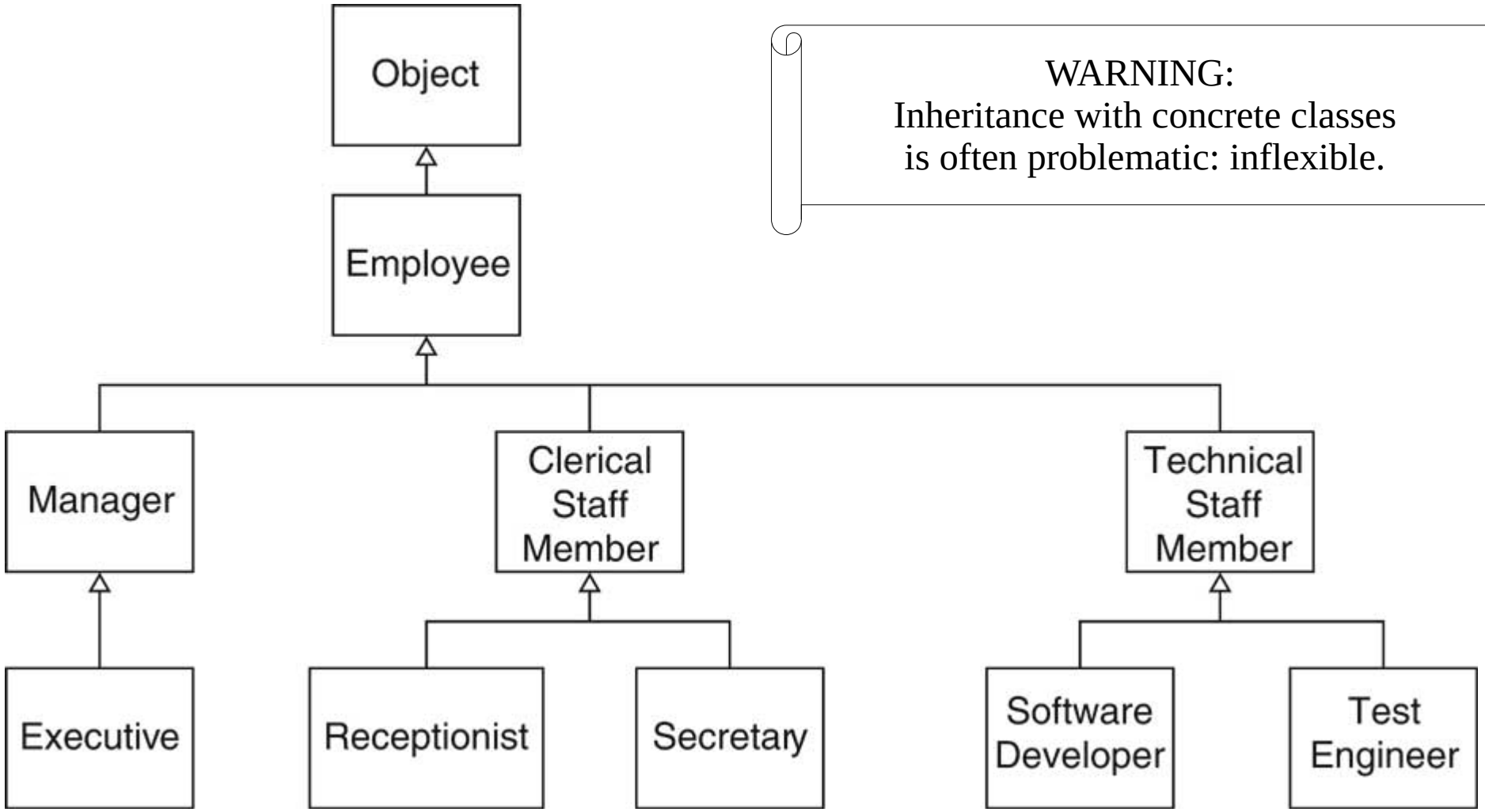
  only creates confusion for programmers!
  - No good reason to use a shadow variable.
  - Pick good, unique names!

Class Hierarchies
and
Flexible Classes

Multiple Inheritance

- Single Inheritance:
  A class may inherit from.. at most one superclass
  - Ex: A Car is a Vehicle.
  - Java uses this approach.
- Multiple Inheritance:
  A class may inherit from many superclasses.
  - Ex: A TA is both a Student and a Teacher.
    - TA multiply inherits Student and Teacher
  - Impossible in Java (specifically forbidden).
- Use.. interfaces to get some benefits of multiple inheritance using only single inheritance.

Inheritance Hierarchy



WARNING:
Inheritance with concrete classes
is often problematic: inflexible.

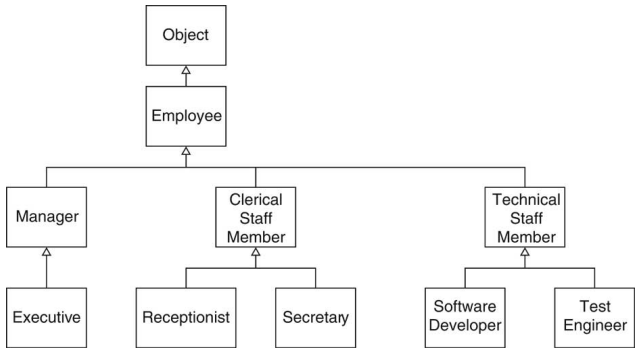
Flexible Objects

- Once instantiated.. an object's type never changes

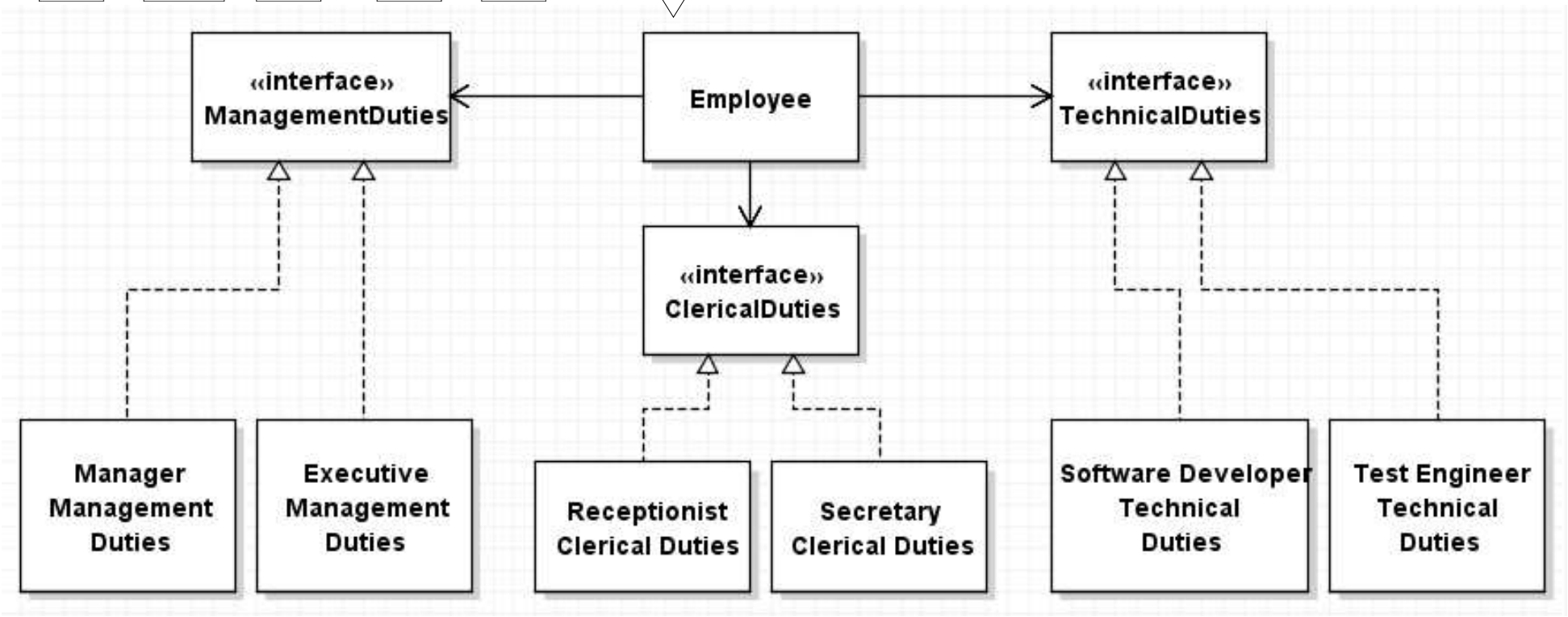
- **Design Principle:**
  Program to an interface, not an implementation
  - Flexibility to reference a different concrete class later

- **Design Principle:**
  Prefer composition over inheritance
  - Composition allows.. runtime flexibility to change
    (reference a new object)
  - Reduces rigid coupling from static inheritance hierarchy

Use Composition Instead



Composition is more flexible.
Can change object an runtime.
Can have multiple duties.



Abstract Class

Abstract Classes

- Abstract class: (basic idea)

  a class with some unimplemented methods

  - abstract method: Un-implemented method.
    Concrete derived classes must.. override (implement) all abstract methods

  - Classes with abstract methods must be abstract.
  - Abstract class cannot be instantiated:
    it's incomplete; not concrete.

- Make a class abstract:
  public abstract class Plant { ... }

- Make a method abstract:
  public abstract void doSomethingAmazing();

Abstract Class Example

```
abstract class GraphicObject {
    int x, y;                                   ..
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();                     ..
}

class Circle extends GraphicObject {
   @Override
    void draw() {
        ...
    }                                           ..
   @Override
    void resize() {
        ...
    }
}
```

Abstract class...

can have instance data and method implementations

Abstract method has no implementation.

draw() and resize() must be..

Abstract Class vs Interface

Similarities

Abstract class: Java interfaces:

- Force derived concrete class to.. implement methods
- Supports constants

Differences

- implemented some methods (non-abstract)
- include instance data (non-constant fields)
- Extend classes

abstract class can inherit from another abstract class

- Class can implement..

multiple interfaces

- In UML, abstract classes shown in *italics*.
  - Sometimes decorated with {abstract}

In Java 8, interfaces can have default (“defender”) methods, but these can only call other methods of the interface.

Abstract Questions

- Can a method be both abstract and final?
  - no, abstract methods must be overriden, but final methods cannot
- Can an abstract class have a static method?
  - yes
- Can a method be both abstract and static?
  - no: would have no definition (abstract)
    so could not be invoked directly on the class (static)
- Can a class be both final and abstract?
  - no, final only for “complete” class
    but abstract classes are incomplete

Math is final with a private constructor.

Visibility

protected only use when needed

- protected
  - allows.derived classes to access a member
    Crates a “protected” interface.
  - unrelated classes cannot access the protected members.
- Not a great idea:
  it exposes implementation details to derived classes
  - breaks encapsulation:

    you have no control over which classes extend your class in the future.
  - Create a “protected” interface to expose just those things that only derived classes will need (“template method”)
    Often better to use public interface.

Class Member Visibility

- Visibility Modifies and member accessibility:
  - public: anywhere
  - protected: in the class, package, and derived classes
  - default: in the class, and package
    - default is without any modifiers; called package-private
  - private: only in the class

	Inside Own Class	Inside Same Package	Inside Inherited Classes	Rest of the world
public	Visible	Visible	Visible	Visible
protected	Visible	Visible	Visible	
"default" no modifier	Visible	Visible		
private	Visible			

Summary

- Inheritance (is-a) used to create subclasses
  - Supports polymorphism
  - Child overrides methods of parents to change behaviour
  - Child uses super in constructor
- Composition is more flexible than inheritance
- Visibility modifiers affect inheritance