Inheritance

Lecture 4-2

"Knowing Is Not Enough; We Must Apply. Wishing Is Not Enough; We Must Do." - Johann Wolfgang Von Goethe

Overview

- Inheritance basics
- Types of inheritance
- Overriding
- Casting

Inheritance: Motivation

- Almost always, new software expands on previous developments; the best way to create it is by imitation, refinement and combination.

- Inheritance makes specialization and extension from existing modules / systems flexible and convenient.

Motivation: Code without Inheritance

Class Definition

```
class Lecture {
  boolean hasTA = true;
  boolean hasExams = true;
  boolean hasAssignments = true;
}
class CSLecture {
  boolean hasTA = true;              // Repeated code
  boolean hasExams = true;           // Repeated code
  boolean hasAssignments = true;     // Repeated code
  boolean isHard = true;
}
class CPLecture {
  boolean hasTA = true;              // Repeated code
  boolean hasExams = true;           // Repeated code
  boolean hasAssignments = false;
  boolean isHard = true;             // Repeated code
  boolean isExciting = true;
}
```

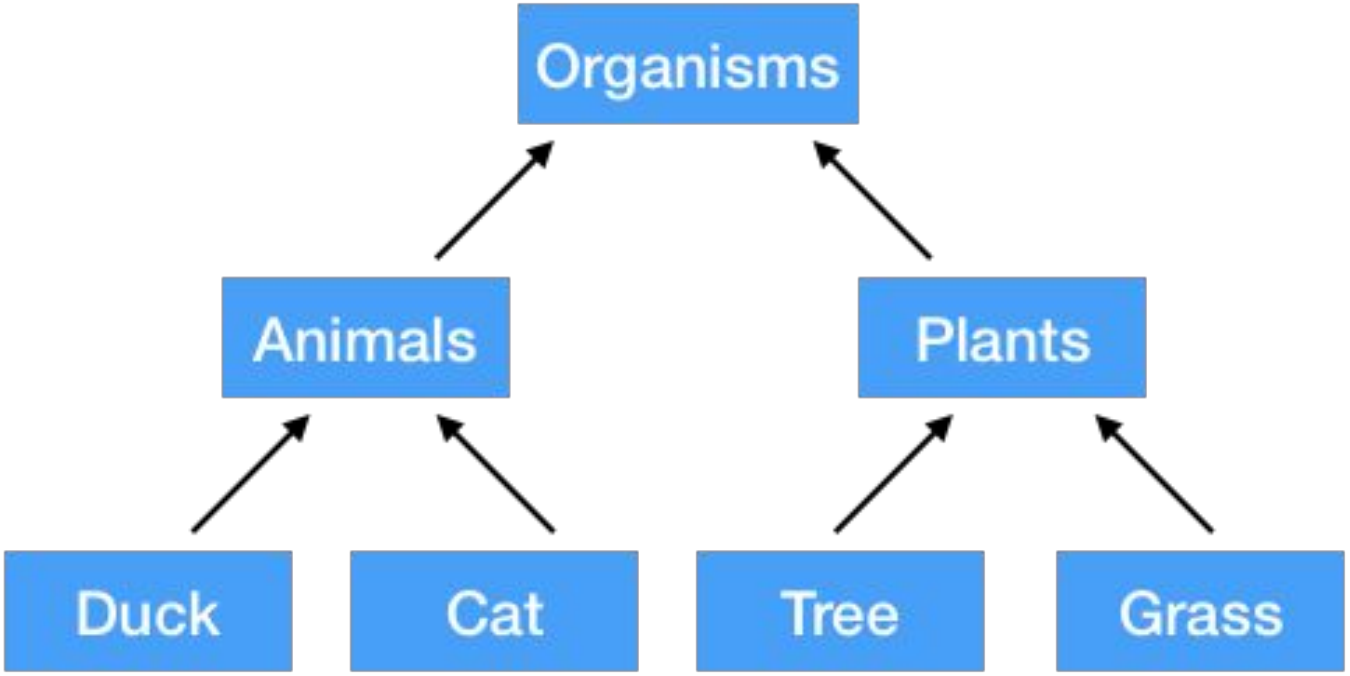
Motivation: Code with Inheritance

```
class Lecture {
  boolean hasTA = true;
  boolean hasExams = true;
  boolean hasAssignments = true;
}
class CSLecture {
  boolean hasTA = true;
  boolean hasExams = true;
  boolean hasAssignments = true;
  boolean isHard = true;
}
class CPLecture {
  boolean hasTA = true;
  boolean hasExams = true;
  boolean hasAssignments = false;
  boolean isHard = true;
  boolean isExciting = true;
}
```

```
class Lecture {
  boolean hasTA = true;
  boolean hasExams = true;
  boolean hasAssignments = true;
}
class CSLecture extends Lecture {
  boolean isHard = true;
}
class CPLecture extends CSLecture {
  boolean hasAssignments = false;
  boolean isExciting = true;
}
```

What is Inheritance?

- Inheritance is a mechanism in which one class is implemented based upon another class.

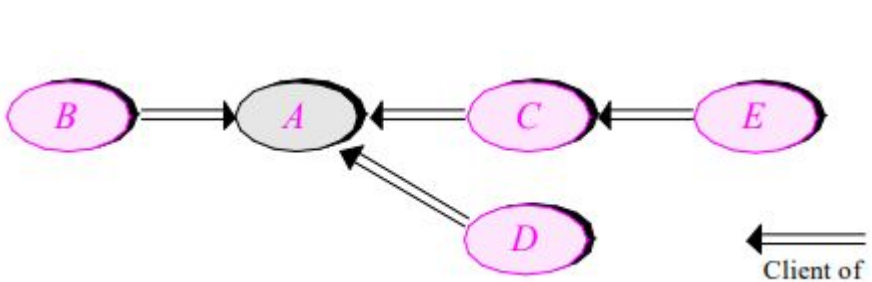


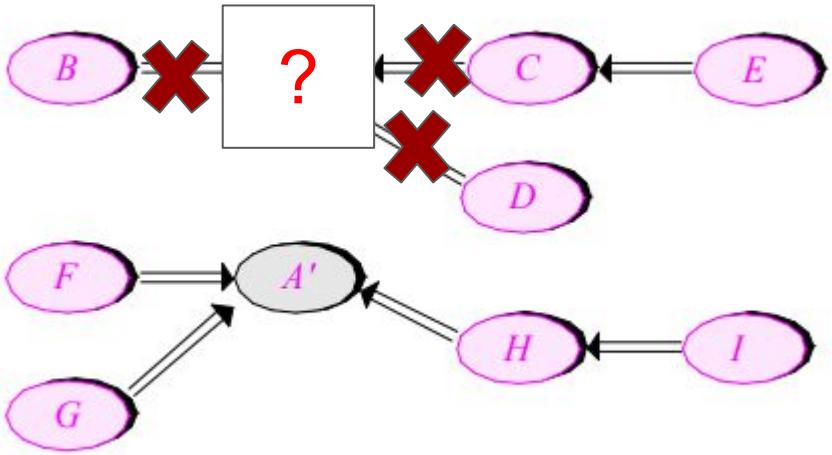
Why Inheritance?

- Inheritance is one of the central concepts of the object-oriented programming.
- Software development involves a large number of classes; many are variants of others.
- Control the resulting potential complexity by a classification mechanism for those classes, known as inheritance.

Advantage: Modularity

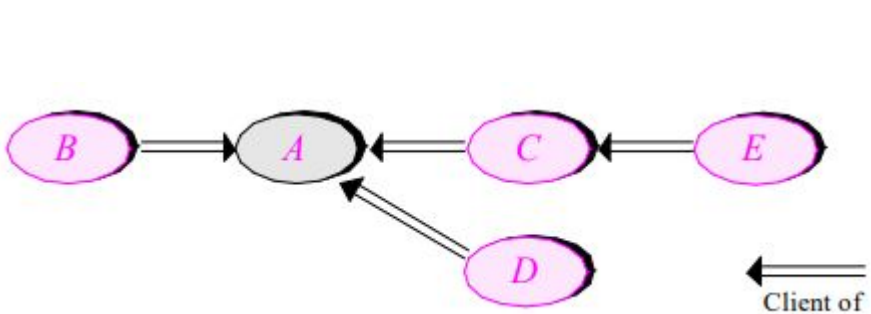
- Let's say you want to extend A to offer the extended or modified functionality (A') without inheritance.
- What you can do is to modify A itself to A'.
  - Old methods may not be able to use A.

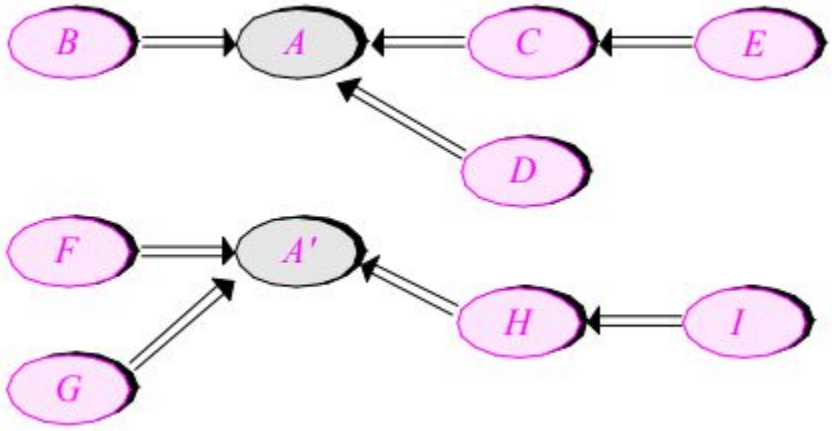




Advantage: Modularity

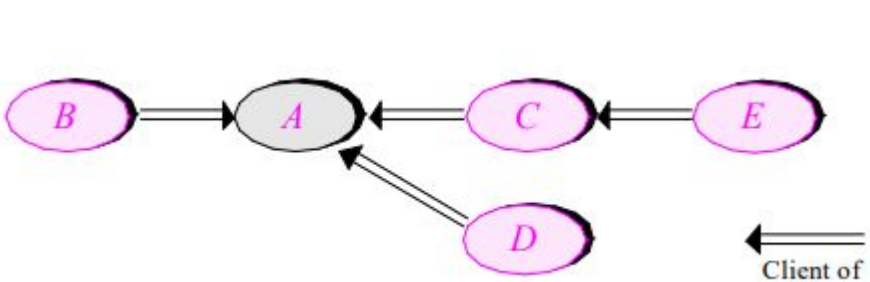
- Another method is to make a copy A', and perform all the necessary adaptations on the new module with no further connection to A.
  - An explosion of similar-but-not-identical variants of the original module.

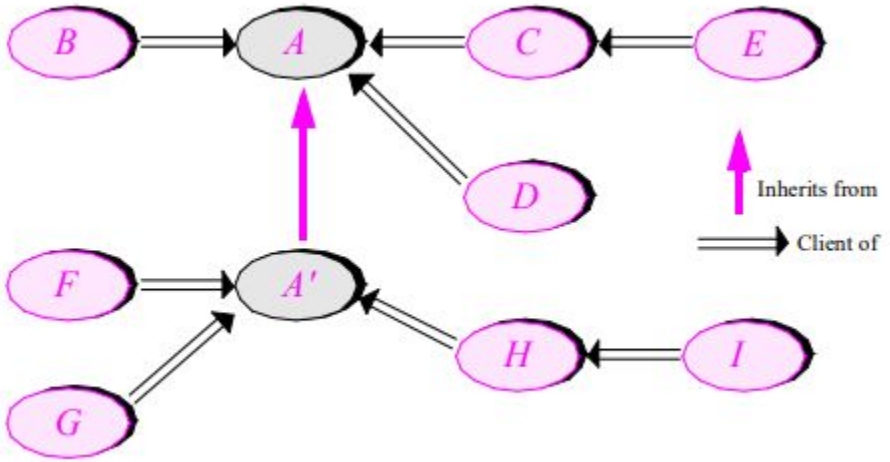




Advantage: Modularity

- Thanks to inheritance, developers can adopt a much more incremental approach to software development





Advantage: Reusability

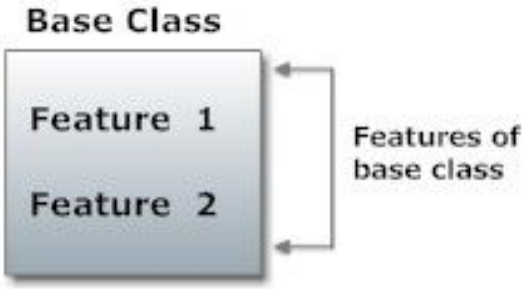
- Create new class from existing class instead of building it entirely from the scratch
  - Decreased maintenance effort
  - Timeliness: speed of bringing projects to completion and products to market
  - Reliability: the expectation that developers will have applied all the required care, including extensive testing & validations
  - Consistency: a strict emphasis on regular, coherent design.

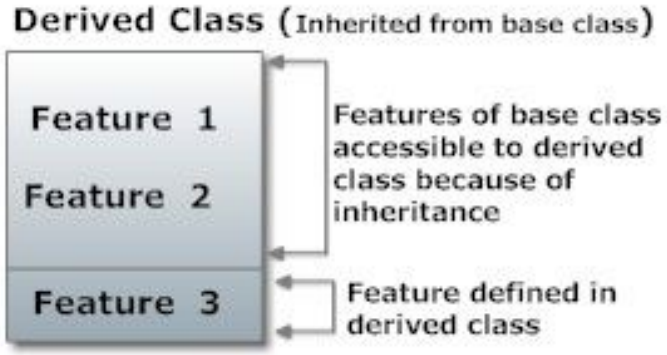
Inheritance in Java

- A class C inherits from its superclass (static and instance) members, m:
  - m is a variable or a method of the superclass of C
  - m is public, protected

Real-world
A child inherits the traits of his/her parents

Object-Oriented Programming
A derived class reuse the fields and methods of the base class





Inheritance Syntax

- Java class can inherit methods and variables from an existing class with “extends”.

```
// Suppose BaseClass is already defined
class DerivedClass extends BaseClass
{
    // fields and methods
 }
```

Inheritance Syntax

- Java class can inherit methods and variables from an existing class with “extends”.

Class Definition

```
class Parent {
  int var = 123;
  void func() { System.out.println("Parent"); }
}
class Child extends Parent { }
```

main() Method

```
Parent parent = new Parent();
Child child = new Child();
parent.func();
child.func();
System.out.println(parent.var);
System.out.println(child.var);
```

Output

```
Parent
Parent
123
123
```

Inheritance of Class Members

```
public class Point {
  int x, y;
  public void move(int dx, int dy) { x += dx; y += dy; }
}
public class Point3d extends Point {
  int z;
  public void move(int dx, int dy, int dz) {
    x += dx; y += dy; z += dz;
  }
}
```

Point3d inherits fields int x,y of Point

Super

- Superclass members can be accessed by using the keyword *super*.
- Overridden methods can be also accessed with super.
- Casting to a superclass type is not effective in attempting to access an overridden method.

Superclass Method Invocation

Class Definition

```
class Parent {
  void printName() { System.out.println("Parent"); }
}

class Child extends Parent {
  @Override
  void printName() { System.out.println("Child"); }
  void printParentName() { super.printName(); }
}
```

main() Method

```
Child child = new Child();
child.printName();
child.printParentName();
```

Output

```
Child
Parent
```

Superclass Method Invocation

Class Definition

```
class T1 { String s() { return "1"; } }
class T2 extends T1 { String s() { return "2"; } }
class T3 extends T2 { String s() { return "3"; }
  void test() {
   System.out.println("s()=\t\t" + s());
   System.out.println("super.s()=\t" + super.s());
   System.out.println("((T2)this).s()=\t" + ((T2)this).s());
   System.out.println("((T1)this).s()=\t" + ((T1)this).s());
  }
}
```

main() Method

```
T3 t3 = new T3();
t3.test();
```

Output

```
s()= 3
super.s()= 2
((T2)this).s()= 3
((T1)this).s()= 3
```

Superclass Constructor Call

- Call the parent class constructor with `super()`

```
class Parent {
  Parent() { System.out.println("Parent"); }
}
class Child extends Parent {
  Child() {
    super();
    System.out.println("Child");
  }
}
```

Class Definition

main() Method

```
Child child = new Child();
```

```
Parent
Child
```

Output

Superclass Constructor Call

- Even without explicit superclass constructor call with super(), the default constructor of superclass is implicitly called.

Superclass

```
class Geometry{
    private int dimension;
    Geometry(int dimension){
     this.dimension = dimension;
    }
    Geometry(){ dimension = 3; }
}
```

Implicit call to the default constructor of superclass

Subclass

```
class Point extends Geometry{
    Point(){}
}
```

Equivalent

Subclass

```
class Point extends Geometry{
    Point(){ super(); }
}
```

Constructor Chaining

- A class constructor calls superclass constructor
- Superclass constructor also calls its superclass constructor. It **chains up** to the class *Object*.

Class Definition

```
class Point {
  int x, y;
  Point() { x = 1; y = 1; }
}
class ColoredPoint extends Point {
  int color = 0xFF00FF;
}
```

main() Method

```
  ColoredPoint cp = new ColoredPoint();
  System.out.println(cp.color);
```

Output

```
0xFF00FF
```

What happened?

Constructor Chaining

- Instance variable initiation comes after the superclass constructor call.

```
//Order of execution
ColoredPoint() { super(); }
Point() { super(); }
Object() { }
x = 1; y = 1;
int color = 0xFF00FF;
```

Accessing Grandparent's Member

- Directly accessing grandparent's member in Java using super is prohibited.

```
class Grandparent {
    public void Print() { System.out.print("Grand"); }
}
class Parent extends Grandparent { }
class Child extends Parent {
    public void Print() { super.super.Print();
        System.out.println("Child");
    }
}
```

main() Method

```
Child child = new Child();
child.Print();
```

Output

```
Compile Error
```

Accessing Grandparent's Member

- Can only access grandparent's member only through the parent class.

```
class Grandparent {
  public void Print(){ System.out.println("Grand");}
}
class Parent extends Grandparent {
  public void Print(){ super.Print();}
}
class Child extends Parent {
  public void Print() {
    super.Print(); System.out.println("Child");
  }
}
```

“Is-a” Rule of Inheritance

- Make a class B inherit from a class A only if every instance of B **is an** instance of A
  - To prevent overkill and support modular extension
- Consider CarOwner, Person, and Car classes. What should CarOwner inherit from?
  - Inherit from Person! “every CarOwner is a Person”
  - But, not from a Car! Instead, include Car as a member variable
    - “Every CarOwner **has a** Car”

“Is-a” vs. “Has-a”

- “Is-a” indicates inheritance.
  - Is-a relation: “ B is a A”
  - More specifically, every B has a component and attribute of A.
  - ex) Every CarOwner is a Person.
- “Has-a” indicates composition.
  - Has-a relation: “Every B has a A”
  - B contains A as a member variable
  - ex) Every CarOwner has a Car

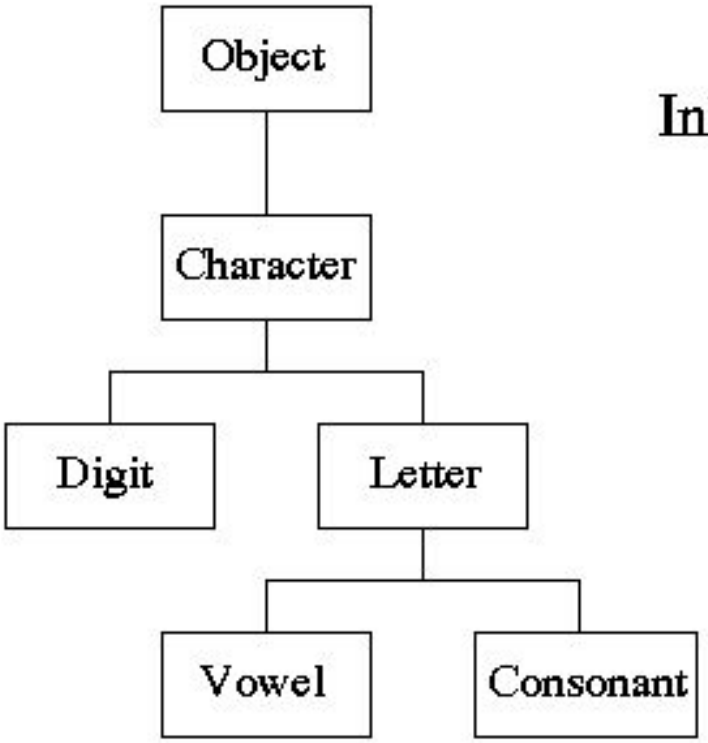
*Circle-Ellipse Problem

- Circle **is an** Ellipse, but if Circle.stretchX or stretchY, the circle instance actually loses its characteristic as a circle.
- Criticism on OOP

```
class Ellipse{
 float axisX, axisY;
 Ellipse(float x, float y){axisX = x; axisY = y;}
 public void stretchX(float scaleX){ axisX *= scaleX; }
 public void stretchY(float scaleY){ axisY *= scaleY; }
}
class Circle extends Ellipse{
  Circle(float radius){
    super(radius,radius);
  }
}
```

Class Hierarchy

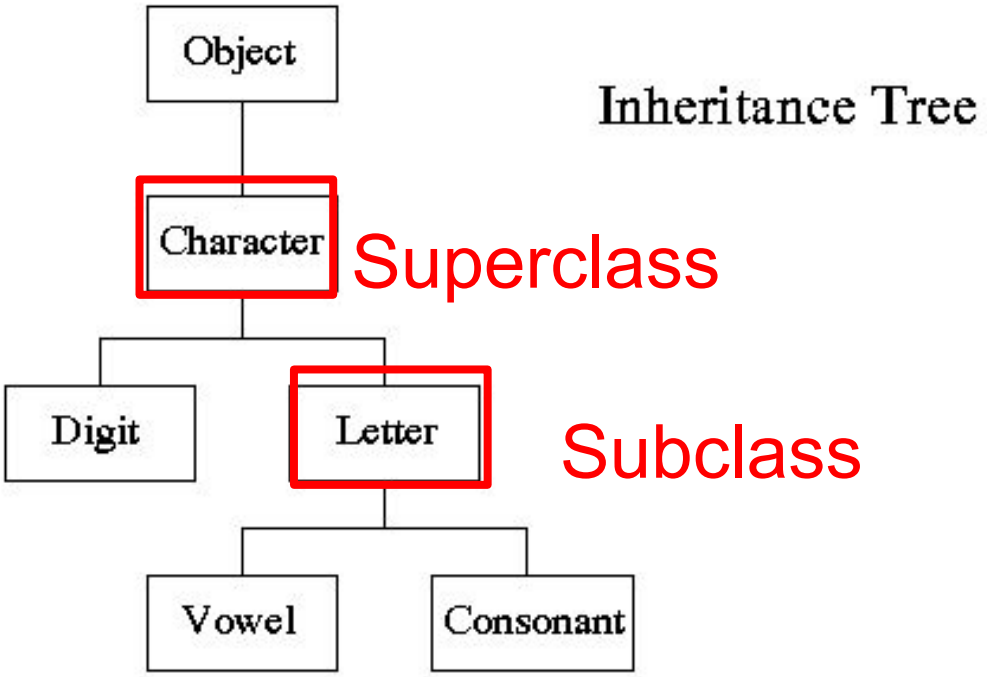
- Class hierarchy (inheritance hierarchy) is very similar to the classifications of species.



Inheritance Tree

Terminology in Inheritance

- Letter class is **derived/inherited from** Character class.
- Character class is **super/parent/base class** of Letter class.
- Letter class is **child/subclass** of Character class.



The Class *Object*

- Default Superclass: In the absence of any other explicit superclass, every class is implicitly a subclass of the *Object* class.
  - All Java classes can be upcast to Object class
- Every class has one and only one direct superclass except the *Object* class.

The Class *Object*

- All class inherit the methods of class *Object*.
  - *toString*(): returns a String representation of the object.
  - *getClass()*: returns the Class object that represents the class of the object.
  - *equals*(): defines a notion of value-based object equality.
  - *finalize(), clone(), hashCode(), wait(), notify(), notifyAll()*
- Classes can override these methods to apply their own.

Check “Is-a” in Java

- Syntax: `<instance> instanceof <Class>`
- True iff the class of `<instance>` inherits from `<Class>`

```
class Parent { }
class Child extends Parent { }
```

main() Method

```
Parent p = new Parent();
Child c = new Child();
System.out.println(p instanceof Parent);
System.out.println(p instanceof Child);
System.out.println(c instanceof Child);
System.out.println(c instanceof Parent);
```

Output

```
true
false
true
true
```

Check “Is-a” with the *Object* Class

- To check that all classes are descendants of the class *Object*

```
class MyClass { }
```

main() Method

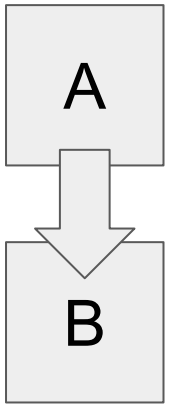
```
MyClass myClass = new MyClass();
System.out.println(myClass instanceof MyClass);
System.out.println(myClass instanceof Object);
System.out.println(String instanceof Object);
System.out.println(Integer instanceof Object);
System.out.println(Exception instanceof Object);
System.out.println(File instanceof Object);
```

Output

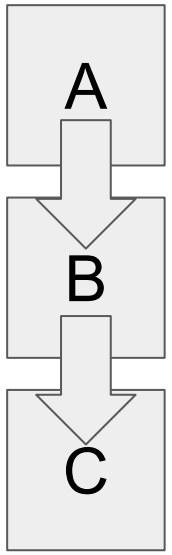
```
true
true
true
true
true
true
```

Types of Inheritance

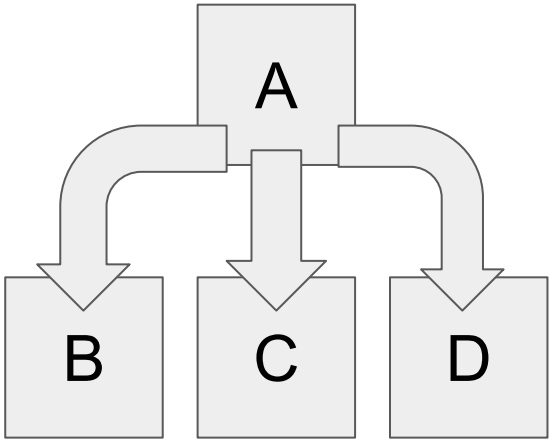
- Single Inheritance
  - Inheriting features of only one superclass
- Multi-level Inheritance
  - Derived class also act as the base class to others
- Hierarchical Inheritance
  - A class serves as a superclass for more than one subclasses



Single Inheritance



Multi-level Inheritance

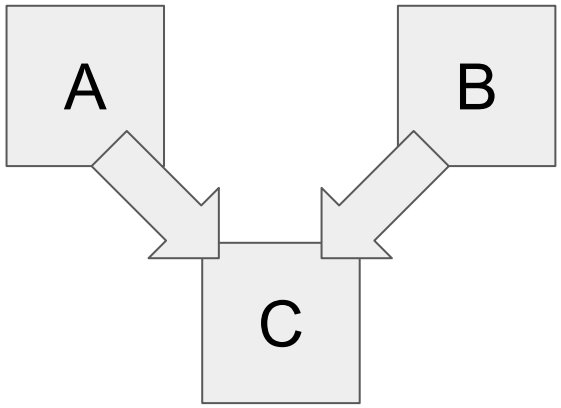


Hierarchical Inheritance

Multiple Inheritance

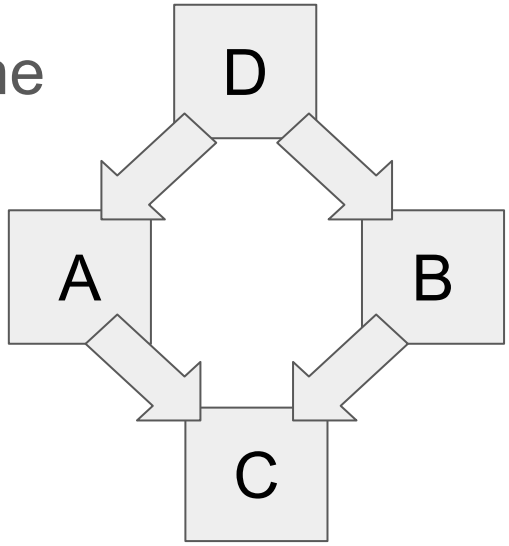
- Inheriting features from multiple parent classes.
- Java **does not** support inheritance from multiple **classes.**
- Java support inheritance from several **interfaces.**

Multiple Inheritance

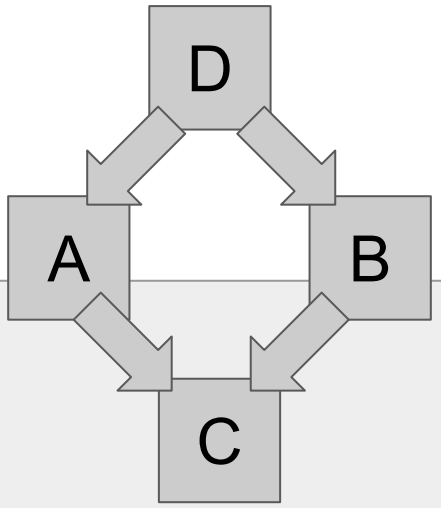


Repeated Inheritance

- A class inherits from another through two or more paths
- Diamond problem
  - If different superclasses have same formatted method, which one should the subclass execute?
  - A main reason JAVA doesn't support multiple inheritances in classes.



Repeated Inheritance




```
Class D {
    public int var = 0;
    public void f(){ var = 1;}
}
Class A extends D{ public void f(){ var = 2;} }
Class B extends D{ public void f(){ var = 3;} }
Class C extends A,B{ public void g(){ f();} }
```

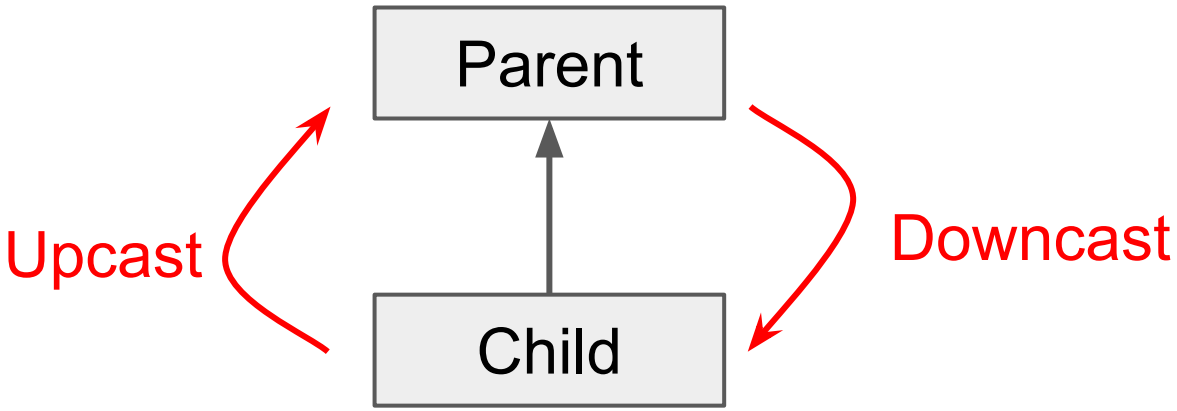
Javac cannot determine whether to execute : f() from A or f() from B

Casting

- Changing a data type of expression from one to another
- Casting doesn’t change **the instance itself**, only the **explicit data type** of the instance
  - ⇒ Only changes the type of the reference

Types of Casting

- There are two types of casting related to inheritance in Java.
  - **Upcast:** Casting a class to its superclass
    - To generalize and utilize type abstraction
  - **Downcast:** Casting a class to its subclass
    - To use extended features of subclass



Upcasting

- There is no requisite for upcasting.
- Both **intrinsic and explicit casting** are available for upcasting.
- **Intrinsic casting**: automatic casting
- **Explicit casting**: Casting with exact class name

```
class Parent { }
class Child extends Parent { }
```

main() Method

```
Parent parent1 = new Child(); // Intrinsic Casting
Parent parent2 = (Parent) (new Child()); // Explicit
Casting
```

Upcasting

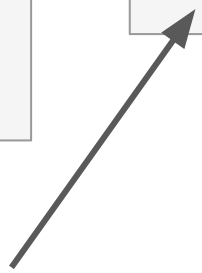
```
class Parent {
  void printName() { System.out.println("Parent"); }
}
class Child extends Parent {
  @Override
  void printName() { System.out.println("Child"); }
}
```

main() Method

```
Child child = new Child();
child.printName();
Parent parent = (Parent) child;
parent.printName();
```

Output

```
Child
Child
```



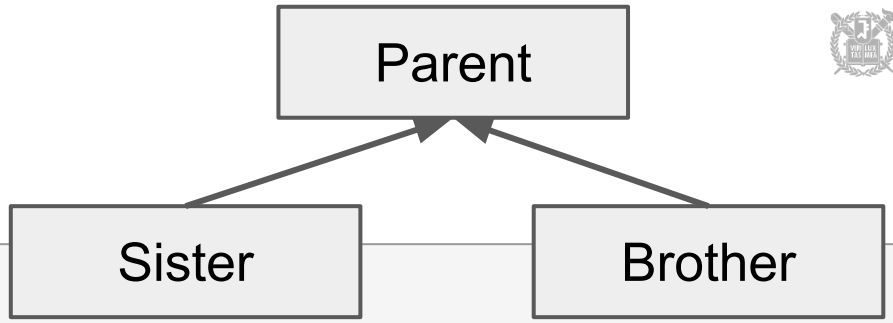
"parent" is still Child class

Casting does NOT change the class/type of the instance itself

Downcasting

- Should explicitly cast.
- Run-time type check.
  - If the actual class of the referenced object is not equal or is not a subclass of the target type
  - ClassCastException will be raised in runtime
  - Not checked in compile time

Downcasting




```
class Parent {
  void print() { System.out.println("Parent"); }
}
class Sister extends Parent {
  void print() { System.out.println("Sister");}
}
class Brother extends Parent {
  void print() { System.out.println("Brother"); }
}
```

main() Method

```
Parent parent = (Parent)
(new Sister());
parent.printName();
Brother sister = (Brother)
parent;
```

Output

```
Sister
Exception in thread “main”
java.lang.ClassCastException:
class Sister cannot be cast
to class Brother
```

Method Overriding

- To redefine inherited instance methods of a subclass when it needs to change or extend the behavior of the method.
- A subclass cannot override constructor.

Overriding in Java

An **instance method** m_C declared in or inherited by class C, overrides another method m_A from C declared in class A, iff all of the following are true:

- C is a subclass of A.
- C implements m_c different from m_A.
- Function name, argument types & orders (signature) of m_C is equal to m_A.
- m_A is public or protected.

https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Overriding Example

```
class Point {
  int x = 0, y = 0;
  void move(int dx, int dy) { x += dx; y += dy; }
}
class SlowPoint extends Point {
  int xLimit = 10, yLimit = 10;
  void move(int dx, int dy) {
    super.move(limit(dx, xLimit), limit(dy, yLimit));
  }
  static int limit(int d, int limit) {
    return d > limit ? limit : d < -limit ? -limit : d;
  }
}
```

main() Method

```
Point p = new SlowPoint(); p.move(32,-32);
System.out.println("(x,y)=("+p.x+","+p.y+")");
```

Output

```
(10,-10)
```

`@Override` Annotation

- Write `@Override` on a method declaration.
- Informs the compiler that the method is meant to override a method declared in a superclass.
  - Compilation error if there is no method with the same format in the superclass.
  - Optional, but strongly recommended to avoid mistakes.

@Override Annotation

```
class Parent {
  void printName() { System.out.println("Parent"); }
}

class Child extends Parent {
  @Override // It raise an error if there is no
"printName" method in "Parent" class
  void printName() { System.out.println("Child"); }
}
```

main() Method

```
Parent parent = new Parent();
Child child = new Child();
parent.printName()
child.printName();
```

Output

Parent

Child

Overriding vs. Hiding

- Hiding (on **static methods)**
  - To redefine a **static method** m
    - m hides superclass method m' of same format (signature)
  - Overriding: on **instance** methods
  - Hiding: on **static** methods
- Static methods called based on type, while instance methods called based on actual instance.

Invocation of Hidden Static Methods

```
class Super {
  static String greeting() { return "Goodnight"; }
  String name() { return "Richard"; }
}
class Sub extends Super {
  static String greeting() { return "Hello"; }
  String name() { return "Henry"; }
}
```

main() Method

```
Super s = new Sub();
System.out.println(s.greeting() + ", " + s.name());
```

Output

```
Goodnight, Henry
```

Hiding Static Variables

- Redeclaration of static variable in a subclass makes the subclass not inherit the variable in the superclass.

```
class Point { static int x = 2; }
class Test extends Point {
  static double x = 4.7;
  void printX() {
    System.out.println(
      x + " " + super.x + " " + ((Point)this).x);
  }
}
```

main() Method

```
Test test = new Test(); test.printX();
```

Output

```
4.7 2 2
```

Hiding Instance Variables

- A child class can declare a variable with the same name as an inherited variable from its parent class, thus hiding the inherited variable.

```
class Parent {
  int var = 123;
}

class Child {
  int var = 456;
}
```

main() Method

```
Parent parent = new Parent();
Child child = new Child();
System.out.println(parent.var);
System.out.println(child.var);
```

Output

```
123
456
```

Accessing Instance Variables

- Access parent class instance variables with `super`, even the variable is hidden (redefined).

```
class Parent { int var = 123; }
class Child extends Parent {
  int var = 456;
  int getParentVar() { return super.var; }
}
```

main() Method

```
Parent parent = new Parent();
Child child = new Child();
System.out.println(parent.var);
System.out.println(child.var);
System.out.println(child.getParentVar());
```

Output

```
123
456
123
```

Accessing Instance Variable

- Unlike overriding, hidden instance variable could be accessed with casting.

```
class Point { int x = 2; }
class Test extends Point {
  double x = 4.7;
  void printX() {
    System.out.println(
      x + " " + super.x + " " + ((Point)this).x);
  }
}
```

main() Method

```
Test test = new Test(); test.printX();
```

Output

```
4.7 2 2
```

Access Modifier

- *protected* members can be accessed from
  - the class itself,
  - subclasses of the class
  - all other classes in the same package of the class

```
public class Shape {
   protected double height, width;
   public void setValues(double height, double width)
     {  this.height = height;  this.width = width; }
}
public class Rectangle extends Shape {
   public double getArea() { return height * width; }
}
```

Access Modifier

- The access modifier of an overriding or hiding method must provide at least as much access as the overridden or hidden method.
  - When overriding a *public* method, then overriding method must be *public.*
  - When overriding a *protected* method, then overriding method must be *protected / public.*
- Private or static methods cannot be overridden either because they are implicitly final.

Public and Protected Inheritance

- Public and protected members could be inherited and overridden/hidden from subclasses.

```
public class Point {
  public int x, y; protected int useCount = 0;
  static protected int totalUseCount = 0;
  public void move(int dx, int dy) {
    x += dx; y += dy; useCount++; totalUseCount++;
  }
}
class PointBack extends Point {
  public void moveBack(int dx, int dy) {
    x -= dx; y -= dy; useCount++; totalUseCount++;
  }
}
```

Default vs. Protected

- Default members cannot be used from other packages, but Protected members can be used from other packages when inherited
- This can be used to manage interactions between packages.

Default vs. Protected

```
package ComplexAlgebra;
public class C {
    protected double real;
    double imag;
    public C(double real, double imaginary) {
        this.real = real; this.imag = imaginary;
    }
    protected C add(C op2) {
        return new C(real + op2.real, imag + op2.imag);
    }
    protected C multiply(C op2){
        return new C(real * op2.real - imag * op2.imag,
                     Imag * op2.real + real * op2.imag);
    }
    double angle() { return Math.atan2(imag, real); }
    double radius() { return Math.sqrt(imag * imag + real * real); }
    @Override
    public String toString() {
        return String.format("(%f)+(%f)j", real, imag);
    }
}
```

Default vs. Protected

```
package RealAlgebra;
import ComplexAlgebra.C;
public class R extends C {
    public R(double value) { super(value, 0); }
    @Override
    public String toString() {
        return "("+Double.toString(real)+")" ;
    }
}
```

protected : add(), multiply()
Can be used outside ComplexAlgebra, like for RealAlgebra
default : angle(), radius()
Cannot be accessed from the outside of ComplexAlgebra

Default vs. Protected

```
C comp = new C(8,8);
R real = new R(7);
System.out.println(
   "(" + comp.toString() + "+" + real.toString() +
   ")*"+ comp.toString() +"="+
   comp.add(real).multiply(comp).toString());
//System.out.println("Radius : "+real.radius()+
      " | Angle : " +real.angle());
System.out.println("Radius of R: " +
      ((C)real).radius() + " | Angle of R:
   "+((C)real).angle());
```

Output

```
((8.0)+(8.0)j+(7.0))*(8.0)+(8.0)j=(56.0)+(184.0)j
Radius of C : 11.31 | Angle of C : 0.78
Radius of R: 7.0 | Angle of R: 0.0
```

Private Member Inheritance

- Subclass does not inherit the private members of its parent class (More precisely, it is invisible).

```
class Point {
  int x, y;
  void move(int dx, int dy)
    { x += dx; y += dy; totalMoves++; }
  private static int totalMoves;
}
class Point3d extends Point { int z;
  void move(int dx, int dy, int dz) {
    super.move(dx, dy); z += dz;
    totalMoves++; // error
  }
}
```

Output

Error: java: totalMoves has private access in Point

Private Member Inheritance

- A private field of a superclass might be accessible to the subclass with a circumvent way.
  - ex) The superclass has public or protected methods for accessing its private fields.
- A private method is cannot be overridden or hidden. In other words, there is no relationship between a superclass and the subclass methods which have a same signature as a private method.

Private Field Inheritance

```
class Point {
  private int x, y;
  void move(int dx, int dy) {
    x += dx;
    y += dy;
  }
}
class Point3d extends Point {
  private int z;
  void move(int dx, int dy, int dz) {
    // Access private superclass fields, x and y, with an
accessible superclass method "move".
    super.move(dx, dy);
    z += dz;
  }
}
```

Private Method Inheritance

```
/* There is no relation between Parent.makeMoney and
Child.makeMoney. They are just different private methods.
*/

class Parent {
  private int makeMoney() {
    return 100;
  }
}

class Child extends Parent {
  private void makeMoney() {
    return 0;
  }
}
```

Final Class and Method

- Final class cannot be inherited.
  - Declare a class final if its definition is complete and no subclasses are desired.
- Final method prevents subclasses from overriding/hiding.
  - But final method can still be inherited.
- Final variable
  - Final variable has nothing to do with inheritance.
  - After the constructor call, that variable cannot be reassigned.

Final Class

- Final class cannot be inherited.

```
class Point { int x, y; }
final class ColoredPoint extends Point { int color; }
class Colored3DPoint extends ColoredPoint { int z; }
```

Output

```
Error: java: cannot inherit from final ColoredPoint
```

Final Method Syntax

- Final method prevents subclasses from overriding/hiding.

```
class Parent {
  final void method() {  System.out.println("out"); }
}

class Child extends Parent {
  @Override
  void method() { super.out(); }
}
```

Output

```
Error: java: method() in Child cannot override method() in
Parent
 overridden method is final
```

Final Variables

- The class Point declares a final class variable origin. The value of the variable Point.origin can never change.

```
class Point {
  int x, y; int useCount;
  Point(int x, int y) { this.x = x; this.y = y; }
  static final Point origin = new Point(0, 0);
}
```

main() Method

```
Point p = new Point();
p.origin = new Point(-1,-2);
```

Output

Error:java: cannot assign a value to final variable origin

Summary

- Inheritance is used in OOP for modularity and reusability.
- Inheritance in Java is done by the keyword ***extends***.
- Multiple Inheritance is not supported in Java.
- To access superclass members from the subclass, use the keyword ***super***.
- Overriding / Hiding of superclass members.
- Effects of inheritance on Casted class instances.
- Access Modifier on Inheritance.

References

- Java SE 12 Language Specifications : https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf
- https://www.researchgate.net/publication/268288994_A_BRIEF_STUDY_ON_INHERITANCE
- Meyer, B. (1988). *Object-oriented software construction* (Vol. 2, pp. 331-410). New York: Prentice hall.
- Kazimir Majorinc, Ellipse-Circle Dilemma and Inverse Inheritance, ITI 98, Proceedings of the 20th International Conference of Information Technology Interfaces, Pula, 1998