Lesson - 5: Inheritance

Wholeness of the Lesson

Java supports inheritance between classes in support of the OO concepts of inherited types and polymorphism. Interfaces play a role similar to abstract classes, and provide a safe alternative to multiple inheritance. Likewise, relationships of any kind that are grounded on the deeper values at the source of the individuals involved result in fuller creativity of expression with fewer mistakes.

Inheritance

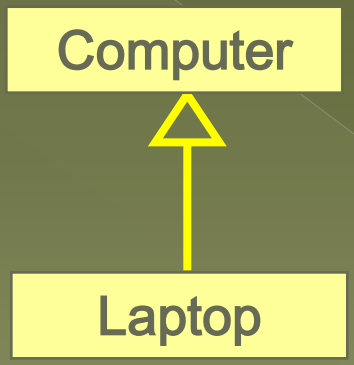
- Another fundamental object-oriented technique is inheritance, used to organize and create reusable classes
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined for the parent class

Inheritance

- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones.
- *Software reuse* is the heart of inheritance.

Inheritance

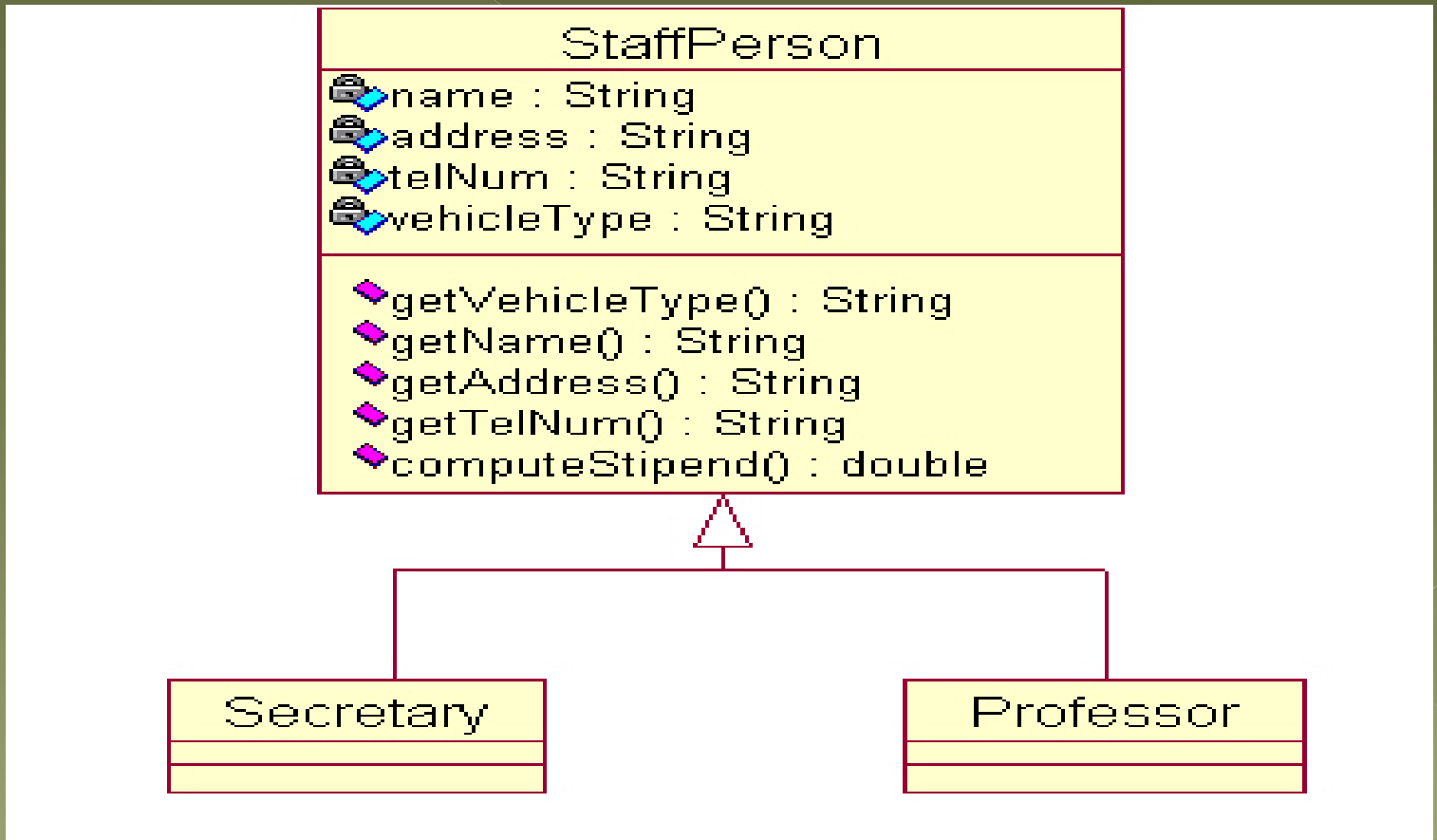
- Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead pointing to the parent class



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

Inheritance

Strategy: Create a *generalization* of Secretary and Professor from which both of these classes *inherit.* A StaffPerson class can be defined having all four fields and related methods, and Secretary and Professor can be defined so they are *subclasses* of StaffPerson.



Deriving Subclasses

- In Java, we use the reserved word extends to establish an inheritance relationship

```
class Secretary extends StaffPerson
      {
         // class contents
      }
class Professor extends StaffPerson
      {
         // class contents
      }
```

Example Code

```
// Parent Class

class square {
    int length, breadth;
    public void get(int x, int y)
    {
        length=x;
        breadth=y;
    }
    int area()
    {
        return(length*breadth);
    }

}
```

```
// Child class
 class cube extends square
{
    int height;
    public void getdata(int x,int
     y,int z)
    {
        get(x,y);
        height=z;
    }
    int volume()
    {
     return(length*breadth*height);
    }

}
```

```
// Main Class
public class SimpleInherit ance{
    public static void main(String a[])
   {
       cube C=new cube();
       C.getdata(10,20,30);

       int b1=C.area();
       System.out.println("Area of Square: "+b1);

       int b2=C.volume();
       System.out.println("Volume of Cube: "+b2);
   }

}
```

Inheritance and Access

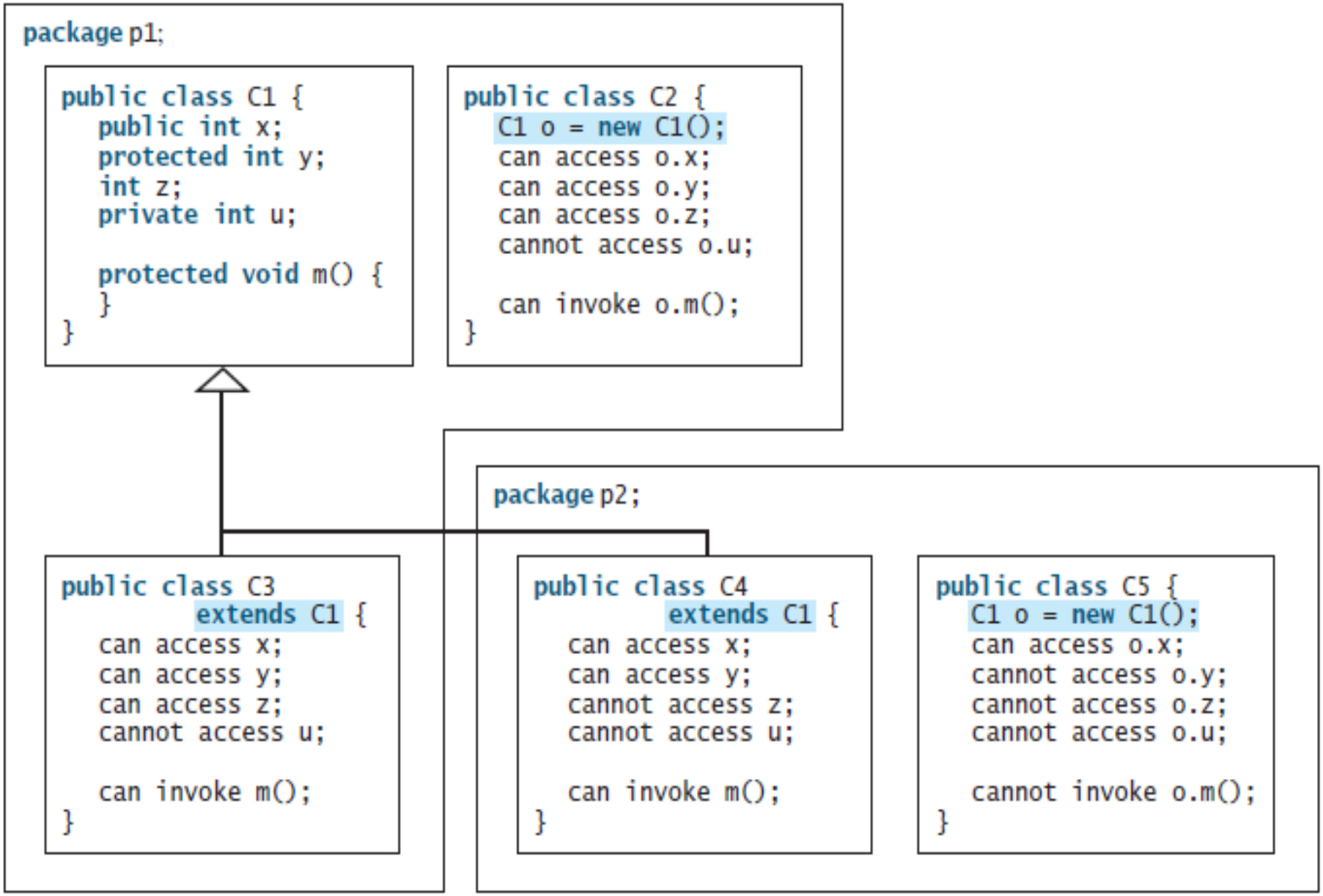
Base class access	Accessibility in derived class
`public`	`Yes`
`protected`	`Yes`
`private`	Inaccessible
Unspecified (package access-default)	`Yes`

Access Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The protected modifier allows a member of a base class to be inherited into a child
- Protected visibility provides more encapsulation than public visibility does
- However, protected visibility is not as tightly encapsulated as private visibility
- Try the implementation of next slide to understand modifiers.



Visibility modifiers are used to control how data and methods are accessed.

Inheritance and Constructors, super Keyword

- *The keyword* **super** *refers to the superclass and can be used to invoke the superclass's data members, methods and constructors.*

- Unlike properties and methods, the constructors of a superclass are not inherited by a subclass. They can only be invoked from the constructors of the subclasses using the keyword **super**.

The super Keyword

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the super keyword to call the parent's constructor

What is the output ?

```
class First{
First(){
System.out.println("Super Class Constructor");
}
}
class Second extends First{
Second()
{
System.out.println("Sub Class Constructor");
}
}
public class TestClass{
public static void main(String[] args) {
Second s = new Second();
}
```

Output

Super Class Constructor

Sub Class Constructor

Super demo

```
// Using super to overcome name
      hiding.
class A {
              int i;
   A(int a){
      i = a;
}
           }

class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super(a);
i = b; // i in B
}
void show() {
System.out.println("i in superclass: " +
      super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

Overriding Methods

- Redefining an instance method in a class, which is inherited from the super class is called method overriding.
- A child class can *override* the definition of an inherited method in favor of its own.
- The child class method must have the same signature as the parent's method, but can have a different body.
- The type of the object executing the method determines which version of the method is invoked

Overriding

- A parent method can be invoked explicitly using the **super** keyword
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code
- @Override
- This annotation denotes that the annotated method is required to override a method in the superclass.

Is it overriding?

```
public class S{
public void print(){
  System.out.println(“S”);
        }
}
public class T extends S{
public void print(String msg){
  System.out.println(msg);
        }
}
```

Example

```
class Loan{
int getRateOfInterest(){return 0;}
}

class Midwest extends  Loan{
int getRateOfInterest(){return 8;}
}

class ICICI  extends  Loan{
int getRateOfInterest(){return 7;}
}
class AXIS extends  Loan{
int getRateOfInterest(){return 9;}
}
```

```
class Main{
public static void main(String args[]){
Midwest  m=new  Midwest();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println(“Midwest Rate of Intere
      st: "+m.getRateOfInterest() + “%”);
System.out.println("ICICI Rate of Interest: "
      +i.getRateOfInterest() + “%”);
System.out.println("AXIS Rate of Interest: "+
      a.getRateOfInterest() + “%”);
}
}
```

Rules for overriding

1. The method must be apply to an instance method. Overriding does not apply to static method.
2. The overriding method must have the same name as the overridden method.
3. The overriding method must have the same number of parameters of the same type in the same order as the overridden method
4. Return type of the overriding and overridden methods must be the same
5. The access level of the overriding method must be at least the same or more relaxed than that of the overridden method.

Overridden(Parent)	**Overriding Access level (Child)**
public	public
protected	public, protected
package-level	public, protected, package-level

Overloading vs. Overriding

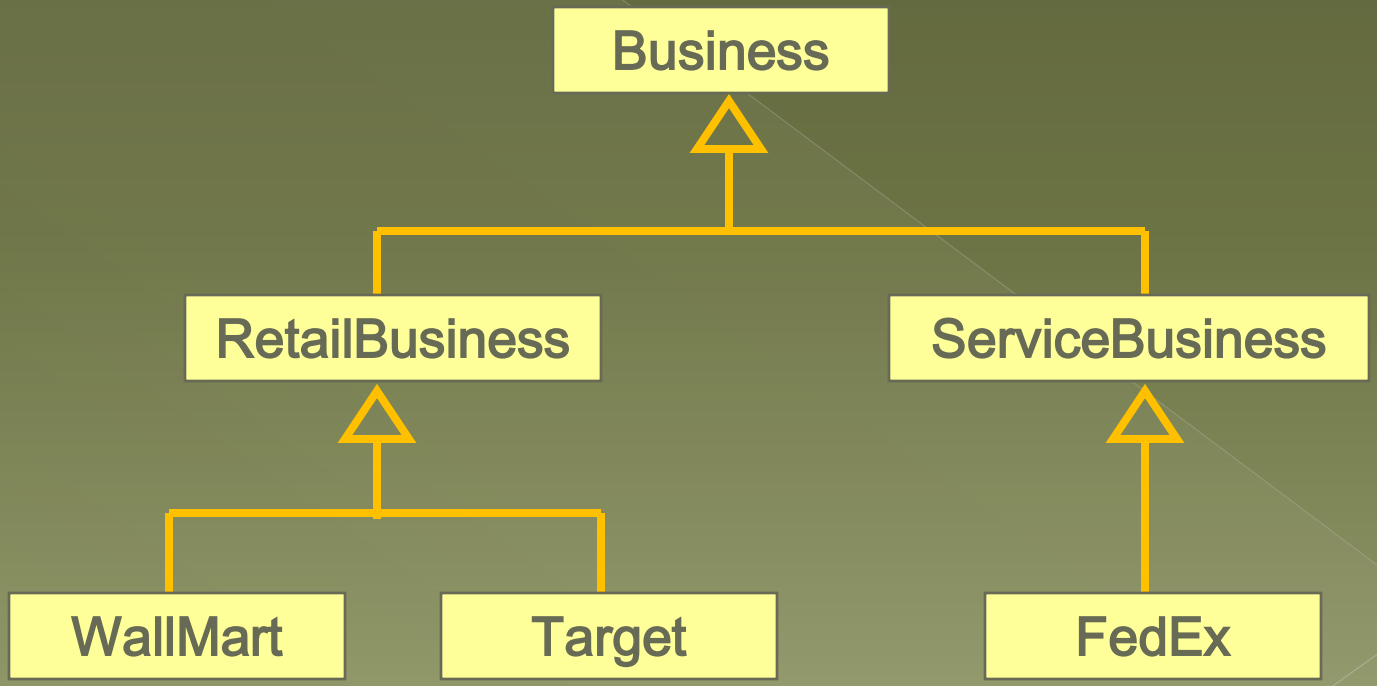
- Overridden methods are in different classes related by inheritance; overloaded methods can be in the same class.
- Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list, return types do not play any roles in overloading.
- Overriding applies only to instance methods; Any method(static/non-static) can be overloaded.
- Overloading is determined during the compile time but overriding determined during the runtime.

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Polymorphism

- Polymorphism refers to the ability of an object to take on many forms.
- *A variable of a supertype can refer to a subtype object.*
- **Compile time Polymorhism(Early/static Binding) :**
  - **Method overloading**
- `obj.doIt(); obj.doIt(10); Obj.doIt(10,20);`
- This line of code might execute different methods at different times, if the object that `obj` points to changes.
- **Runtime Polymorhisrm(Late/dynamic binding) :**
  - **Method Overriding** :Polymorphic references are resolved at run time this is called *dynamic binding.*

Main Point

One class (the *subclass)* inherits from another class (the *superclass*) if all proteced and public data and methods in the superclass are automatically accessible to the subclass, even though the subclass may have additional methods and data not found in the superclass. Java supports this notion of inheritance.

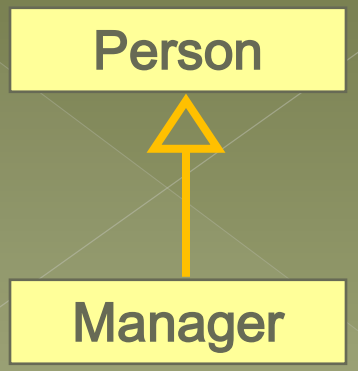
In Java syntax, a class is declared to be a subclass of another by using the *extends* keyword.

Likewise, individual intelligence "inherits from" cosmic intelligence, though each "implementation" is unique.

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the Person class is used to derive a child class called Manager, then a Person reference could be used to point to a Manager object

```
Person p = new Person();
Manager Jon = new Manager();
Person obj =p;
obj.display();
Obj = Jon;
Obj.display();
```



Right use of inheritance

- Manager IS-A Employee – it's not just that the two classes have some methods in common, but a manager really is an employee. This helps to verify that inheritance is the right relationship between these classes.

- ***Substitution principle***. Another test is: Can a Manager instance be used whenever an Employee instance is expected? The answer is yes, since every manager really is an employee, and partakes of all the properties and behavior of an employee, though managers support extra behavior.

- *Dynamic binding* : When the getSalary() method is called on staff[0], the version of getSalary that is used is the version that is found *in the Manager class*.
- This is possible because the JVM keeps track of the actual type of the object when it was created (that type is set with execution of the "new" operator).
- The correct method body (the version that is in Manager) is associated with the getSalary() method at runtime – this "binding" of method body to method name is called *late binding* or *dynamic binding*.

```
Employee[] staff = new Employee[3];
Manager boss = new Manager("Boss Guy", 80000, 2009, 12, 15);
staff[0] = boss;
staff[1] = new Employee("Jimbo", 50000, 2012, 10, 1);
staff[2] = new Employee("Tommy", 40000, 2013, 3, 15);
for (Employee e : staff) {
System.out.println("Name: " + e.getName() + "\nSalary: "
+ e.getSalary() + "\nHire Day :" + e.getHireDay());
}
```

Dynamic Binding : Example

```
// Class Super
class Sup {
void who() {
System.out.println("who() in Sup");
}
}
// Subclass1
class Sub1 extends Sup {
void who() {
System.out.println("who() in Sub1");
}
}
```

```
//Subclass2

class Sub2 extends Sup {
void who() {
System.out.println("who() in Sub2");
}
void who1() {
System.out.println("who1() in Sub2");
}
}
```

Dynamic Binding : Example

```
class DynDispDemo {
public static void main(String args[]) {
Sup superOb = new Sup();
Sub1 subOb1 = new Sub1();
Sub2 subOb2 = new Sub2();
Sup supRef;
supRef = superOb;
supRef.who();
supRef = subOb1;
supRef.who();
supRef = subOb2;
supRef.who();
 }
}
```

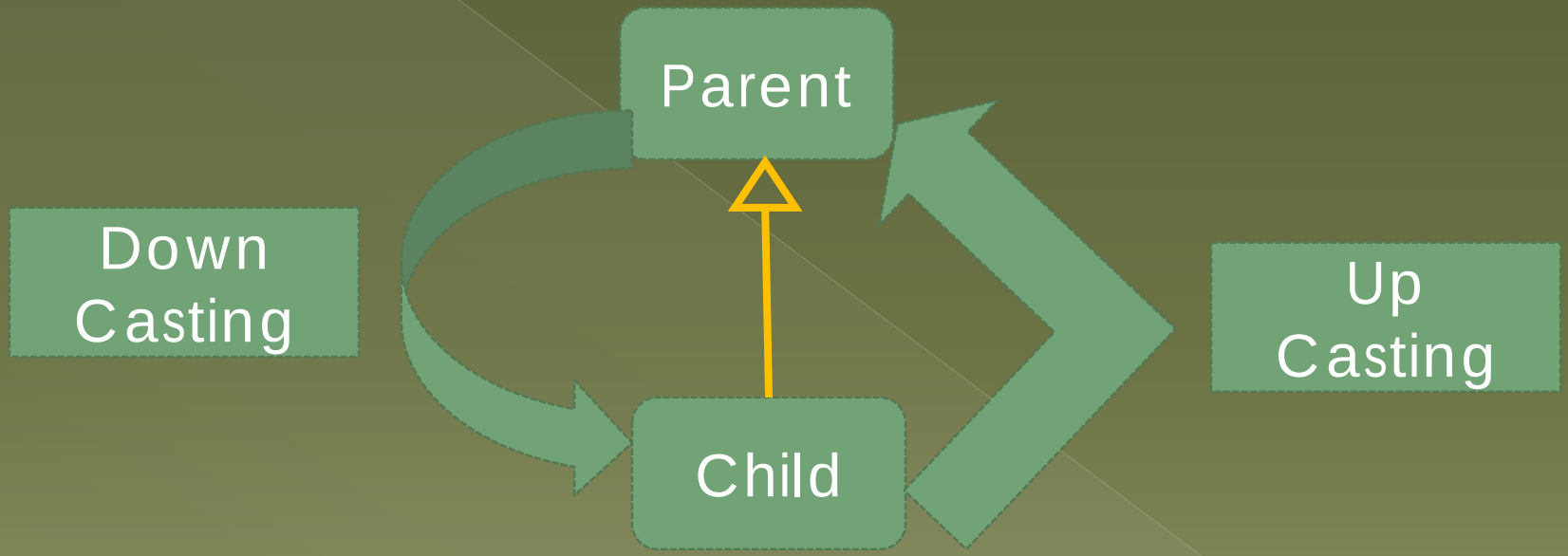
Output

who() in Sup
who() in Sub1
who() in Sub2

Casting of objects

- *One object reference can be typecast into another object reference. This is called* casting object.
- It is always possible to cast an instance of a subclass to a reference of a superclass (known as *upcasting*), because an instance of a subclass is *always* an instance of its superclass. It is type safe.
- When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used to confirm your intention to the compiler with the **(SubclassName)** cast notation.
- If the superclass object is not an instance of the subclass, a runtime **ClassCastException** occurs.

Upcasting / Downcasting



Up casting – Down casting

- Upcasting is a very powerful feature of inheritance.
- It lets you write polymorphic code that works with classes that exists and classes that will be added in future.
- Use a simple rule to check if an assignment is a case of upcasting.
- Look at the compile-time type (declared type) of the
- expression on the right side of the assignment operator (e.g. b in a = b).
- If the compile-time type of the right hand operand
- is a subclass of the compile-time type of the left-hand operand, it is a case of upcasting, and the assignment is safe and allowed.

Example - Upcasting

```
Object obj;
Employee emp;
Manager mgr;
PartTimeManager ptm;
// An employee is always an object
obj = emp;
// A manager is always an employee
emp = mgr;
// A part-time manager is always a manager
mgr = ptm;
// A part-time manager is always an employee
emp = ptm;
```

Down casting

```
Employee emp;
Manager mgr = new Manager();
emp = mgr; // Ok. Upcasting
mgr = emp; // A compiler error. Downcasting
```

- The assignment emp = mgr is allowed because of upcasting. However, the assignment mgr = emp is not allowed because it is a case of downcasting where a variable of superclass (Employee) is being assigned to a variable ofsubclass (Manager).
- The compiler is right in assuming that every manager is an employee (upcasting). However, not every employee is a manager downcasting).

```
mgr = (Manager)emp; // OK. Downcast at work
```

instanceof *operator*

- The instanceof operator can be used to determine that whether a reference variable has a reference to an object or a subclass of the class at runtime.
- It takes two operands and return true or false
- Syntax : <Class reference Variable> instanceof <Class name>
- Eg : String s = "Hello";

```
if (s instanceof java.lang.String) {
  System.out.println("is a String"); // true
```

instaceof and casting

```
Object[] stuff = {"Java", 10.11,12,13,16.11,20,"Hi"};
double sum = 0;
for(int i=0;i<stuff.length;i++){
if(stuff[i] instanceof Number) // checking instance
{
Number next = (Number)stuff[i]; // Down casting
sum+=next.doubleValue();
}
}
System.out.println("Sum of Doubles = " + sum);
```

Main Point

As a matter of good design, a class C should not be made a subclass of a class D unless C "IS-A" D.

Likewise, individual intelligence "is" cosmic intelligence, though this relationship requires time to be recognized as true.

Demo Code

- SimpleInheritance.java
- ManagerTest1.java
- MoreInherit.java
- SuperThisDemo.java
- OverrideDemo.java
- DynamicBind.java
- InstanceofDemo.java
- Hierarchical.java
- UpcastDemo.java

Practice