Review

- Inheritance:
  - Represents the “is-a” relationship between classes
  - Represents specialization of classes (subsets)
  - Represents a way of describing alternatives (alternative subclasses)
  - Is a mechanism for reusability

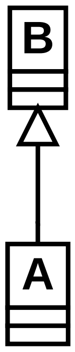
Inheritance

- Whenever we have a situation which states that “every A is a B”, we model this as

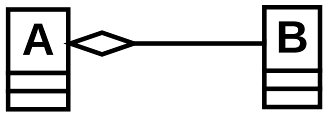
  ```
  class A extends B { ... }
  ```

- All attributes and methods from the parent class (or super class) B are “inherited” by the subclass (or derived class) A.

- Class A can have (and usually does have) additional attributes and methods.



represents:
"every A is a B"
(inheritance)



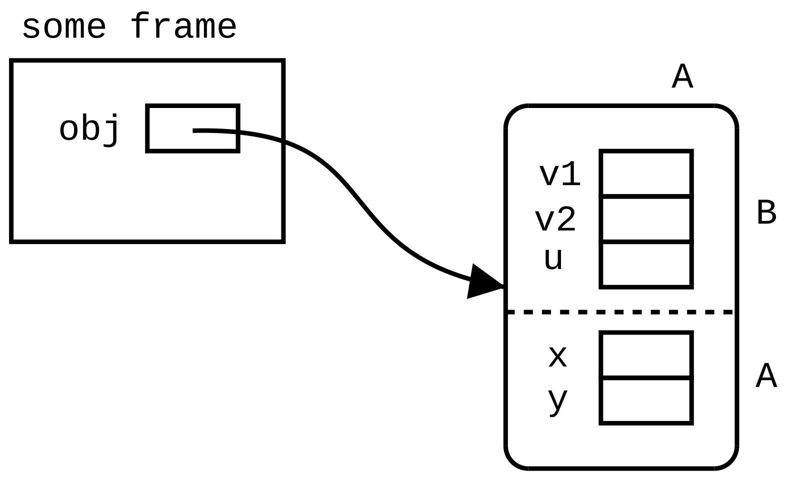
represents:
"every A has a B"
(aggregation)

Inheritance

```
class C { ... }
class D { ... }
class E { ... }
class B {
  C v1, v2;
  D u;
  void m() { ... }
}
class A extends B {
  E x;
  C y;
  void p() { ... }
  void s() { ... }
}
```

Inheritance

```
// In some client
A obj = new A();
obj.p();
obj.m();
// We can refer to ... obj.x ... obj.y ...
// ... obj.u ... obj.v1 ... obj.v2 ...
```



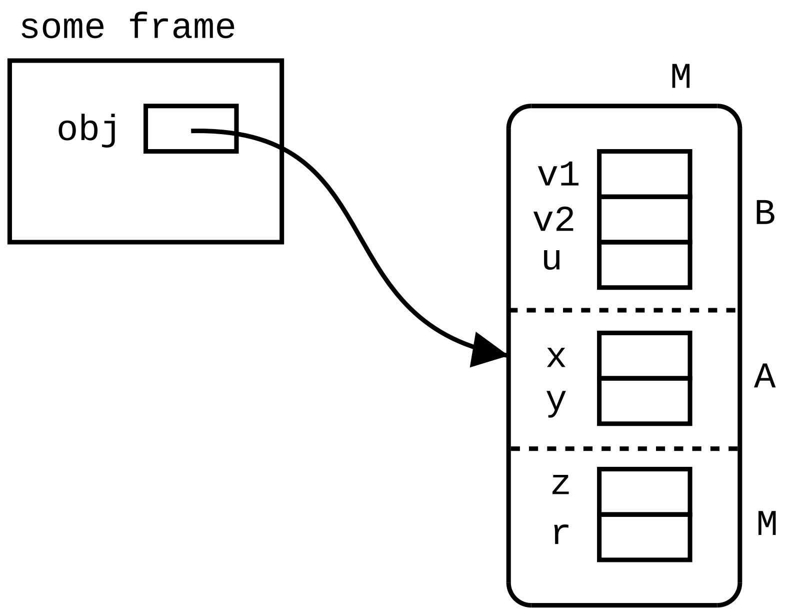
Inheritance

- A method in a subclass can access the attributes and methods of its super class.

```
class C { ... }
class D { ... }
class E { ... }
class B {
  C v1, v2;
  D u;
  void m() { ... v1 ... v2 ... u ... m() ... }
}
class A extends B {
  E x;
  C y;
  void p()
  {
    ... x ... y ... p() ... v1 ...
    ... v2 ... u ... m() ...
  }
  void s() { ... }
}
```

Inheritance

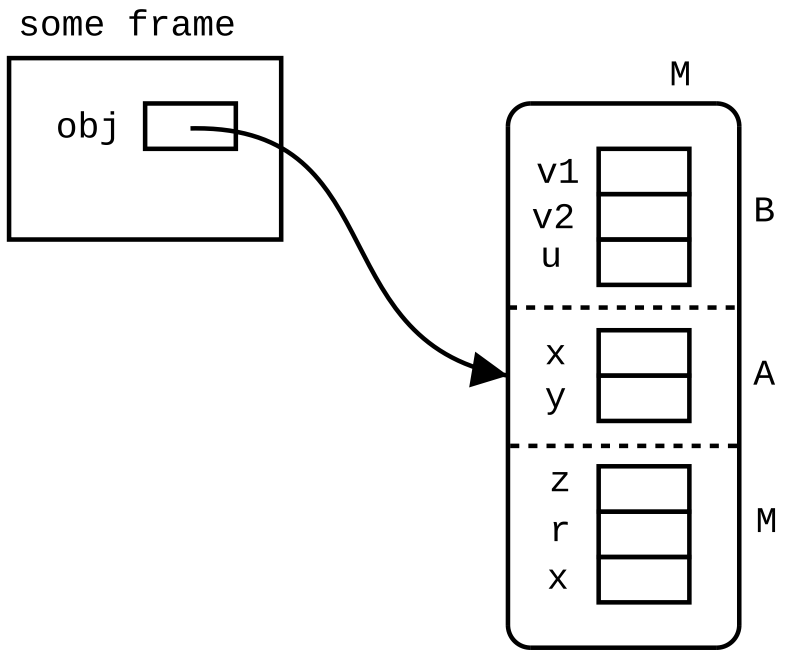
```
class M extends A {
  E z;
  D r;
  void q() { ... }
}
// Somewhere else
M obj2 = new M();
```



Shadowing variables

- An attribute or instance variable can be redefined in a subclass. In this case we say that the variable in the subclass *shadows* the variable in the parent class.

```
class M extends A {
  E z;
  D r, x;
  void q() { ... }
}
```



Accessing variables from the super class

- The `super` reference is used to access an attribute or method in a parent class.

```
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ... super.x ...
  }
}
```

Overriding methods

- A method can be redefined in a subclass. This is called *overriding* the method.

```
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ... super.x ...
  }
  void p()
  {
    ...
  }
}
```

Accessing a method or attribute

- When we try to access a method or attribute of an object, it is looked up by the Java runtime system in the class of the object first. If it is not found there, it is looked up in the parent class. If it is not found there, it is looked up in the grand-parent, etc...

```
M obj3 = M();
obj3.q();  // From class M
obj3.m();  // From class B
obj3.p();  // From class M
obj3.s();  // From class A
```

- Attributes and methods declared as `private` cannot be accessed directly by the subclasses, even though they are present in the object. They can be accessed only indirectly by public accessor methods in the class that declared them as private.

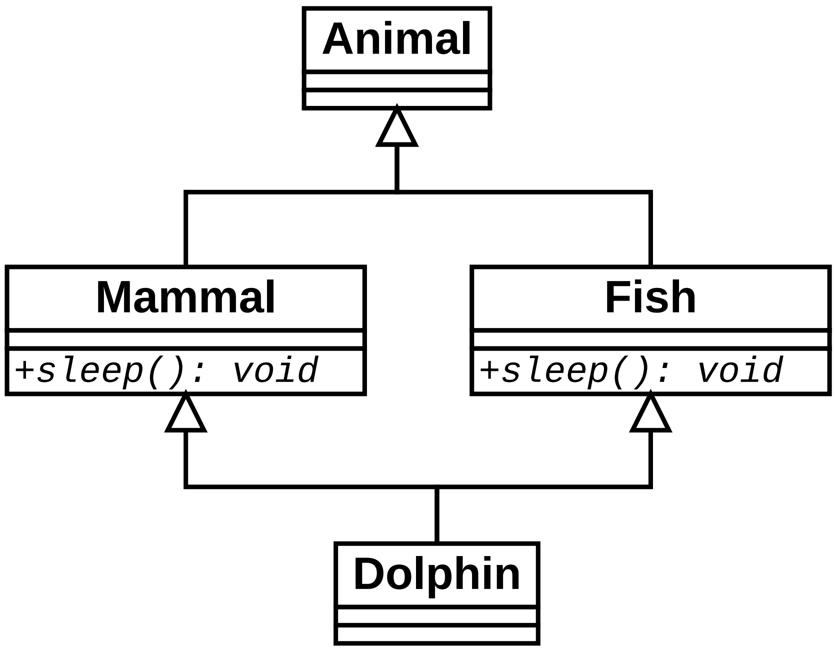
Accessing a method or attribute

```
class A extends B {
  private E x, y;
  void p() { }
  void s() { }
  public E get_x() { return x; }
}
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ...
    // instead of super.x ...
    ... get_x() ... or ... super.get_x() ...
  }
}
```

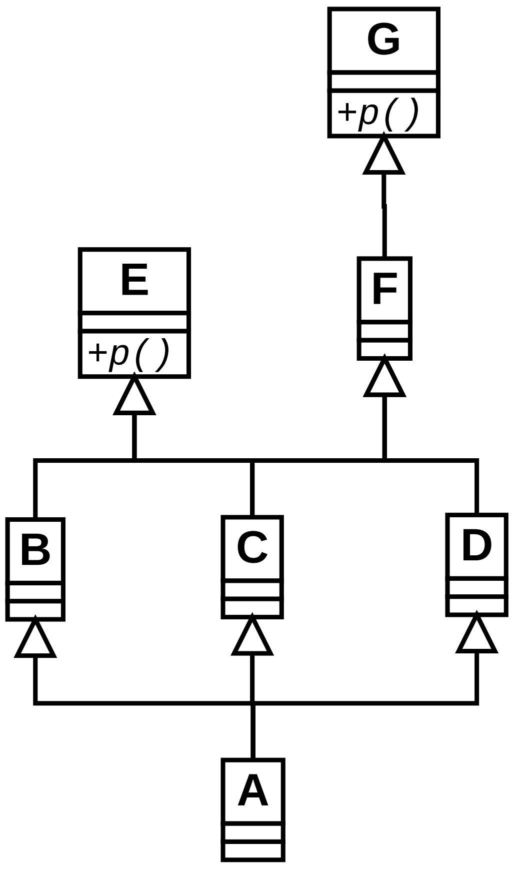
Accessing a method or attribute

- An attribute or method declared as `protected` can be accessed by any subclass, even if it is in a different package.
- An attribute or method declared as `final`, is not inherited at all, i.e. it forbids overriding.
- A class declared as `final`, cannot have subclasses.

Multiple inheritance



Multiple inheritance



Polymorphism

- Polymorphism means "many forms."
- Polymorphism is the characteristic of being able to assign a different meaning or usage to something in different contexts
- A polymorphic method is a method which can accept more than one type of argument
- Kinds of polymorphism:
  - Overloading (Ad-hoc polymorphism): redefining a method in the same class, but with different signature (multiple methods with the same name.) Different code is required to handle each type of input parameter.
  - Parametric polymorphism: a method is defined once, but when invoked, it can receive as arguments objects from any subclass of its parameters. The same code can handle different types of input parameters.

Polymorphism

```
class Creature {
  boolean alive;
  void move()
  {
    System.out.println("The way I move is by...”);
  }
}
class Human extends Creature {
  void move()
  {
    System.out.println("Walking...");
  }
}
class Martian extends Creature {
  void move()
  {
    System.out.println("Crawling...");
  }
}
```

Ad-hoc Polymorphism (Overloading)

```
class Zoo {
  void animate(Human h)
  {
    h.move();
  }
  void animate(Martian m)
  {
    m.move();
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human joseph = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(joseph);  // Polymorphic call
  }
}
```

Parametric Polymorphism

```
class Zoo {
  void animate(Creature c)
  {
    c.move();
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human joseph = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(joseph);  // Polymorphic call
  }
}
```

Accessing super

```
class Human extends Creature {
  void move()
  {
    super.move();
    System.out.println(“Walking...”);
  }
}
class Martian extends Creature {
  void move()
  {
    super.move()
    System.out.println(“Crawling...”);
  }
}
```

Casting and instanceof

- Casting is like putting a special lens on an object
- A casting expression is of the form

```
(type) expr
```

where `type` is any type (primitive or user-defined) and `expr` is an expression which evaluates to an object reference whose type is compatible with `type`.

- Not all casts are possible

```
(int) ‘‘Hello’’
(Engine) joseph
```

Casting

- If a variable is a reference of type A, it can be assigned any object whose type is a subclass of B.

  ```
  Human greg = new Human();
  Creature c = greg;
  ```

- But a reference of type B cannot be assigned directly reference of type A, if B is a subclass of A (because A has less attributes than required by B):

  ```
  Creature d = new Creature();
  Martian m = d;
  ```

- ...however, if we know that a reference x of type A points to an object of type B (and B is a subclass of A,) then we can force to see x as being of type B by using a casting expression:

  ```
  Creature e = new Martian();
  Martian f = (Martian)e;
  ```

Checking the type of a reference

- To find out whether a reference `r` is an instance of a particular class `C` we use the boolean expression:

```
r instanceof C
```

- This is normally used whenever we do casting:

```
class Human extends Creature {
  void move()
  {
    System.out.println(“Walking...”);
  }
  void jump()
  {
      System.out.println("Up and down");
  }
}
```

Checking the type of a reference

```
class Martian extends Creature {
  void move()
  {
    System.out.println("Crawling...");
  }
  void hop()
  {
      System.out.println("Down and to the left");
  }
}
class Zoo {
  void move(Creature c)
  {
    if (c instanceof Human)
      ((Human)c).jump();
    else if (c instanceof Martian)
      ((Martian)c).hop();
    c.move();
  }
}
```