CSE 11: Lecture 8

- ✔ Objects, classes, and inheritance
- ✔ Visibility and inheritance
- ✔ Overriding methods
- ✔ Dynamic method binding and the Java type system

(Reading: Savitch, Ch. 7)

Objects and classes

- ✔ You know that an OOP *object* is something like a real-world object:
  - ✗ every object has some properties or state
  - ✗ every object has some possible behaviors or actions
  - ✗ every object is an instance of some class

- ✔ Well, OOP *classes* are also something like real-world classes:
  - ✗ a class can have a superclass that it is a subclass of
    (the class of mammals is a subclass of the class of animals, which is a subclass of the class of living things, etc.)
  - ✗ a class can have a subclass that it is a superclass of
    (the class of mammals is a superclass of the class of primates, which is a superclass of the class of humans, etc.)
  - ✗ every instance of a class is an instance of the class's superclass, too
    (every eagle IS-A bird, every bird IS-A animal, every animal IS-A living thing...)
  - ✗ a subclass inherits the properties and behavior of its superclass, and adds some more
    (every animal has a size and weight and can reproduce; every bird does too, plus it has wings and feathers, and can lay eggs, etc.)

Objects and classes and inheritance

- An object gets instance variables and methods depending on what class it is an instance of:
  - *The definition of its class* specifies what kind of properties and behavior the object will have
    - properties <--> instance variables
    - behavior <--> methods

- But an object also gets properties and methods of its superclass, and of its superclass's superclass, etc.

- This is called *inheritance*

Inheritance and OOP

- ✔ Inheritance is defined as:

  The acquisition by one class (called the *derived class* or *subclass* or *child class*) of the variables and methods of another class (called the *base class* or *superclass* or *parent class*)

- ✔ Inheritance is an essential aspect of object-oriented programming:

  - ✗ Designing a derived class D with inheritance from a base class B lets you create D as a class that has lots of behavior and state (the methods and variables inherited from B) that you don’t have to write... you just inherit it!

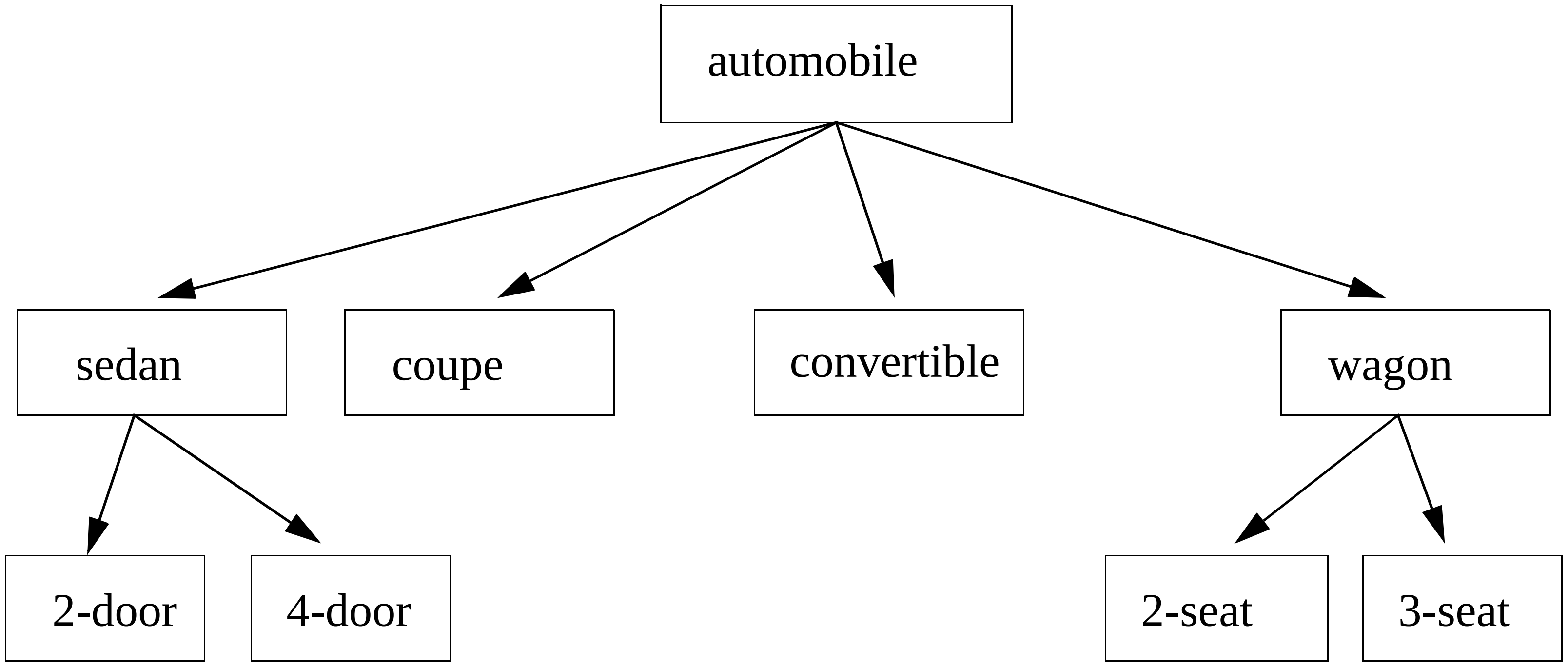
  - ✗ When this works right, it is a great example of software re-use, and is an important reason that object-oriented programming improves software productivity

Some definitions

- ✔ *Parent* class: same as *base* class
- ✔ *Child* class: same as *derived* class. If class P is parent of class C, then C is child of P
- ✔ *Ancestor* class (or superclass): If A is parent of parent (or some other number of "parent" iterations) of D, then A is an ancestor of D
- ✔ *Descendant* class (or subclass): If A is ancestor if D, then D is descendant of A
- ✔ The relationship among classes can be shown in a *class hierarchy diagram*
  - ✗ In a class hierarchy diagram, usually a parent class has its child classes drawn beneath it, connected by arrows

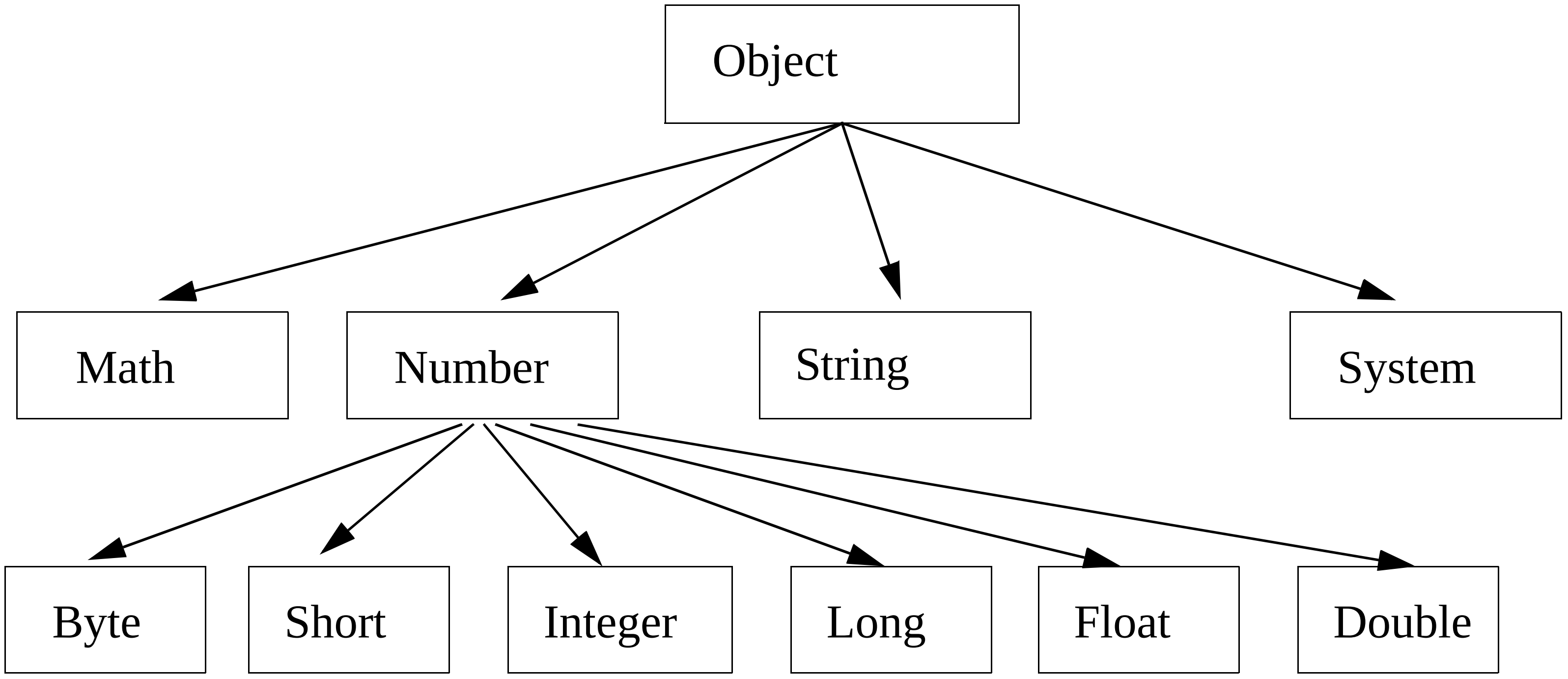
Single inheritance

- Single inheritance: every class has at most one parent class
- Java permits only single inheritance from classes
- An example of a class hierarchy diagram showing single inheritance:

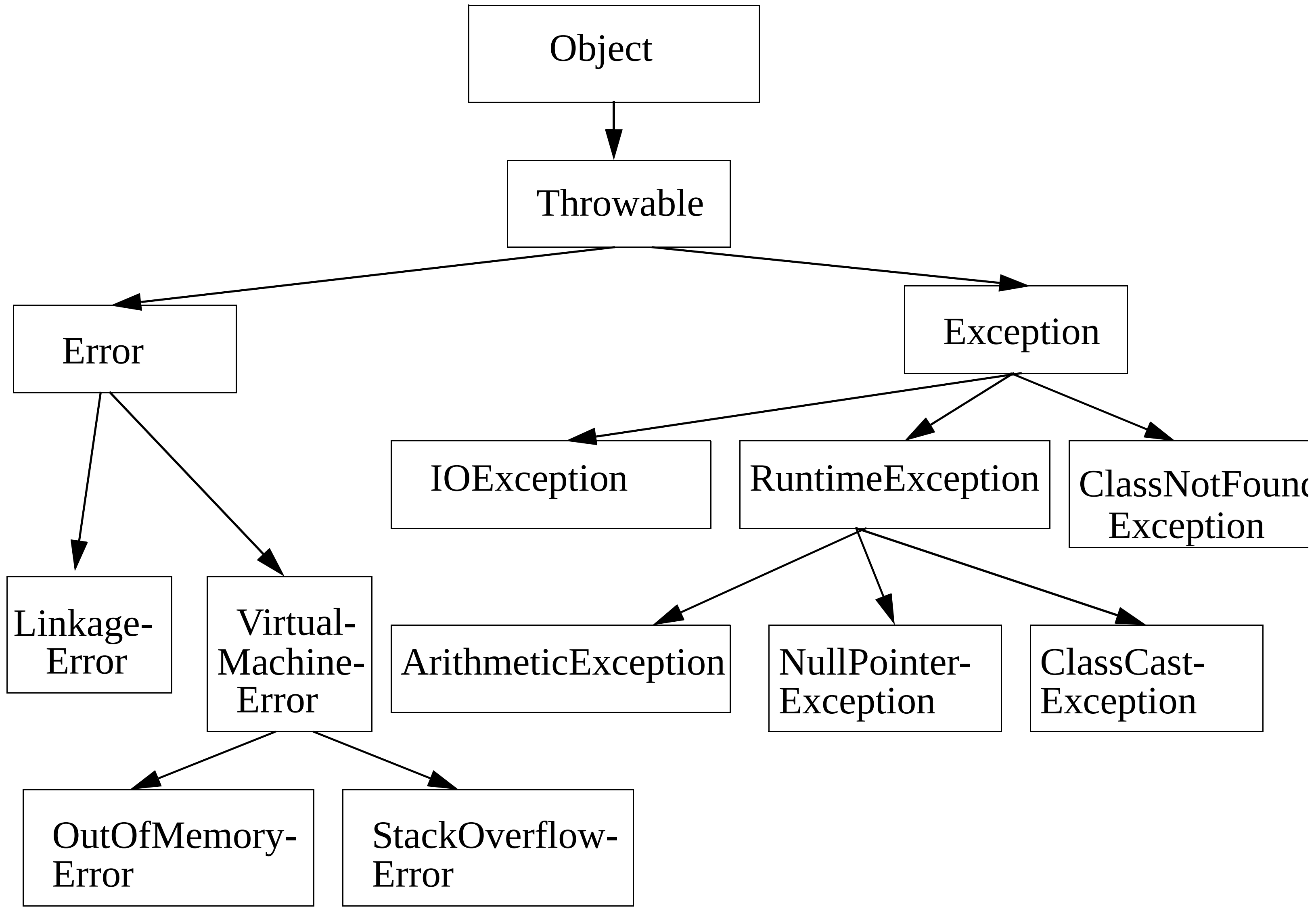


- What is the parent of the sedan class? What are the children of the wagon class? What are the ancestors of the 4-door class? What are the descendants of the automobile class?

Part of the Java standard library class hierarchy

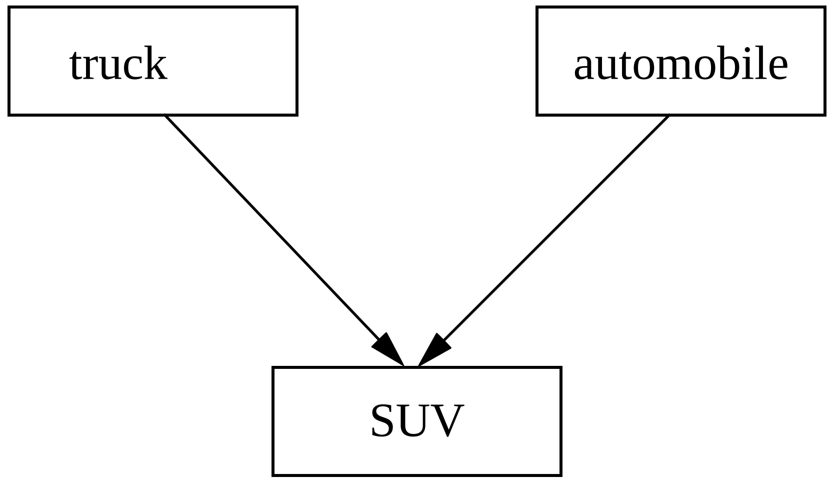


Another part of the Java standard library class hierarchy



Multiple inheritance

- Multiple inheritance: when a class has more than one parent class
- An example of a class hierarchy diagram showing multiple inheritance:



- Java does not permit multiple inheritance of classes, but does permit multiple inheritance of *interfaces* (more on interfaces later in the course...)

How to do inheritance in Java

- ✔ Actually, every user-defined class in Java is a derived class! So you have done this already (if you don't specify a base class, the class `Object` will be the base class.)
- ✔ But let's do an example with a user-defined class we will use as a base class:

```
public class Jar {
   public Jar() {  // construct an empty Jar (with 0 units)
      numUnits = 0; }

   public Jar(int n)  { // construct a Jar with n units
      numUnits = n; }

   public void add(int n) {   // add n units to the Jar
      numUnits += n; }

   public int quantity() {   // return number of items in Jar
      return numUnits; }

   private int numUnits;
}
```

- ✔ We will use `Jar` as a base class, and create a derived class

First: Why bother?

- ✔ Q: Why bother?
- ✔ A: Software re-use!
  - ✗ The class `Jar` implements an ADT
  - ✗ It has been tested and debugged (hopefully)
  - ✗ You want to create a class that has all the functionality of `Jar`, plus more
    - • `LabeledJar`: like `Jar`, but has a string label which can be printed out
  - ✗ Instead of modifying the definition of the `Jar` class to do this... (which may introduce bugs and break existing code... and you may not even have access to Jar.java!) ...
  - ✗ ... make `LabeledJar` a derived class of `Jar`
    - • inherit `Jar`'s methods and variables, and add more

Creating a derived class

- ✔ To create a derived class in Java, use the `extends` keyword with the name of the base class in the derived class definition header:

```
public class LabeledJar extends Jar {

   public LabeledJar(String _label) { // ctor initializing label
      label = _label;
   }
   // ctor that initializes label and number of items
   public LabeledJar(String _label, int _nitems) {
      super(_nitems);  // call the base class ctor
      label = _label;
   }

   public String getLabel() { // return the label
      return label;
   }
   private String label;

}
```

Using a derived class

```
public class FooBar {

   public static void main (String args[]) {

      LabeledJar j = new LabeledJar("Nails",25);

      j.add(3);  // calling an inherited method to add 3 items!

      System.out.print("The " + j.getLabel() +
         " jar contains " + j.quantity() + " items.\n");

   }

}
```

✔ Note that add() and quantity() do not appear in the definition of LabeledJar... and yet we can call them as instance methods of a LabeledJar object.

✔ This is inheritance! How does it work?

Inheritance rules in Java

- ✔ `public` members (variables and methods) in the base class are inherited as public members of the derived class
  - ✗ (except: constructors are not inherited)
- ✔ `protected` members of the base class are inherited as protected members of the derived class
- ✔ `private` members of the base class are inherited by the derived class, but they do not become public, protected, or private members of the derived class...

  ... they cannot be accessed by the derived class, except through inherited public or protected methods
- ✔ "package" members of the base class are inherited as "package" members of the derived class, if both classes are in the same package... Otherwise, they are inherited like private members are

Visibility and inheritance: an example

- ✔ This example uses instance variables, but the same rules apply to inheritance of instance methods:

```
class Bar {
   public int i;
   protected int j;
   int k;
   private int p;
}

-----------------------------------------

class Foo extends Bar {
   public void doStuff() {
      i = 3;   // okay, i is inherited public
      j = 4;   // okay, j is protected & we are in a subclass
      k = 5;   // okay, if Foo is in same package as Bar
      p = 6;   // ERROR! p is inherited, but not accessible!
   }
}
```

Calling base class constructors

- Every instance of a derived class contains its own copy of all base class instance variables, along with any instance variables defined in the derived class
- When a derived class constructor is executing, it is important that initialization of the base class part of the object is already complete (the new derived class object may depend on it)
- You can explicitly call a base class constructor in the body of a derived class constructor: refer to the base class constructor as **`super()`**
  - If the base class constructor you want to call takes arguments, you can pass it arguments
- If you call the base class constructor explicitly this way, it *must* be the first statement in the body of the derived class constructor
- If you do not call the base class constructor (or another derived class constructor) explicitly as the first statement, Java will implicitly insert a call to the base class's default constructor before the first statement in the body of the derived class constructor
- (Examples in the LabeledJar constructor definitions...)

Extending and overriding

- The basic idea behind inheritance: if B is the base class of a derived class D, then
  - each D "is-a" B, and has all the public methods and variables that a B has, plus possibly some more
  - in Javanese: D "extends" B

- But sometimes this is not what you want! Example:
  - Intuitively, class Rectangle is a superclass of Square... each Square is a Rectangle... BUT:
    - a behavior of a Rectangle is that its height and width can be changed independently.....this is *not* a behavior of a Square!
  - A Bird class should have a "fly" method, because that's a behavior that all Birds have. BUT:
    - A Hawk is a Bird, and a Hummingbird is a Bird, and though they both fly, they fly differently. (And what about the Penguin or Ostrich class?...)

- So: Sometimes you do not want to inherit a base class's method; instead you want to define a different version of the method in the derived class.
  ...This is called *overriding* the base class method (not overloading!)

Overriding inherited methods

- ✔ A derived class normally just inherits public methods of a base class, and adds additional ones of its own
- ✔ But the derived class may redefine (override) an inherited method
  - ✗ It does this by defining a method with exactly the same signature: same name, and number and type of arguments, and return type
  - ✗ This is necessary if the derived class needs not only to extend, but also to specialize the base class in some way
- ✔ In the definition of derived class's methods, the base class version of an overridden method can be called by preceding the call with **`super.`** (that's super followed by dot)
- ✔ (Note : If the method is declared **`final`** in the base class, this means it cannot be overridden in a derived class.)

Overriding a method: an example

- ✔ Consider this class, to be used as part of an inventory control system:

```
public class Supply {
   public Supply(double cost, int howMany) {
      this.cost = cost;
      this.howMany = howMany;
   }
   ....

   // display data of a Supply object
   public void display() {
      System.out.println("Cost: $" + cost);
      System.out.println("# in stock: " + howMany);
   }
   ...
   ...
   private double cost;  // the cost of this Supply in $
   private int howMany;  // how many in stock
}
```

Overriding a method: an example, cont’d

- ✔ A Pencil is-a Supply, but with different display() behavior:

```
public class Pencil extends Supply {
   ...
   public Pencil(double cost, int howMany, String _color) {
      super(cost, howMany);  // call base class ctor
      color = _color;
   }
   ...
   // display data of a Pencil object
   public void display() {  // override inherited display()
      System.out.println("Type: Pencil");
      super.display();  // call base class method
      System.out.println("Color: " + color);  // display color
   ...
   private String color;  // aditional instance variable
}
------------------------------------------------------------
Pencil p = new Pencil(0.25, 100, "blue");
p.display();    // prints ??
```

Classes, subclasses, and the “is-a” relationship

- ✔ Class derivation implies a basic “is-a” relationship between objects that are instances of the classes

  If A is a class, and class B is a subclass of A, C is a subclass of B, and D is a subclass of C, then...

  every A is-a A
  every B is-a A
  every C is-a B
  every D is-a A, etc.
  ... but NOT the case that every A is-a B, and NOT the case that every B is-a D, etc.

- ✔ You can think of this in terms of real-world classes and objects:

  The class Human is a subclass of the class Primate, the class Primate is a subclass of Mammal, Mammal is a subclass of Animal, etc., and so...

  every Human is-a Animal
  every Mammal is-a Animal, etc.
  ... but NOT the case that every Mammal is-a Human, etc.

Class derivation, the is-a relationship, and the Java type system

- The basic is-a relationship between objects of classes that have a derivation relationship is reflected in the Java type system:
  - A pointer of class type can point to an object of any descendant of that class
  - So assignment from a reference type expression to a variable of any ancestor type is allowed
- Example: suppose class B extends class A, C extends B, D extends C. Declare a pointer of type A, and one of type B

```
A myA;
B myB;
```

- Now because of the is-a relationship implied by class derivation, `myA` can point to an A, B, C, or D; `myB` can point to a B, C, or D:

```
myA = new A();  // okay, a A is-a A
myA = new B();  // okay, a B is-a A
myA = new C();  // okay, a C is-a A
myB = new B();  // okay, a B is-a B
myB = new C();  // okay, a C is-a B
myA = myB;      // okay, a B is-a A
myB = new A();  // ERROR! not every A is-a B!
myB = myA;      // ERROR! not every A is-a B!
```

Method overriding and class derivation

- ✔ So, a pointer of type A can point to an object that is an instance of any descendant of A
- ✔ But subclasses might override -- have different versions of -- instance methods of A:

```
   // in A.java
   public class A {
   public void act() { System.out.println("I’m an A!"); }
   }

--------------------------------------------------------------
   // in B.java
   public class B extends A {
   public void act()  { System.out.println("I’m a B!"); }
   }

--------------------------------------------------------------
  // in C.java
   public class C extends B {
   public void act()  { System.out.println("I’m a C!"); }
   }
```

Calling overridden methods: what version gets called?

✔ Now what happens if you call the method `act()` in various ways?

```
A myA;  B myB; C myC;

myA = new A();  myB = new B();  myC = new C();
myA.act();   // prints ??  (no surprise)
myB.act();   // prints ??  (no surprise)
myC.act();   // prints ??  (no surprise)
```

✔ ... but consider these:

```
myA = new B();

myA.act();   // prints ??

myA = new C();

myA.act();   // prints ??
```

Dynamic binding

- Java uses "dynamic binding" (also called "virtual" or "late binding") of public instance method calls: the method called depends on the *type of the actual calling object* the pointer points to, not the type of the pointer!

  ("dynamic" because this can in general only be determined at run time. One reason static methods are called "static" is that calls to them are bound statically at compile-time, not dynamically)

- Why want this? Because it can increase the reusability of code...
  example: write a single method that can handle any object that is-a A

```
void ProcessA(A obj)
{
   ..... // stuff

   obj.act();  // want this to do the right thing...
               // call the act() method for the object passed
               // as an argument, as long as it's an instance of
               // a class that's descended from A
}
```

- This is an example of "polymorphism" in Java

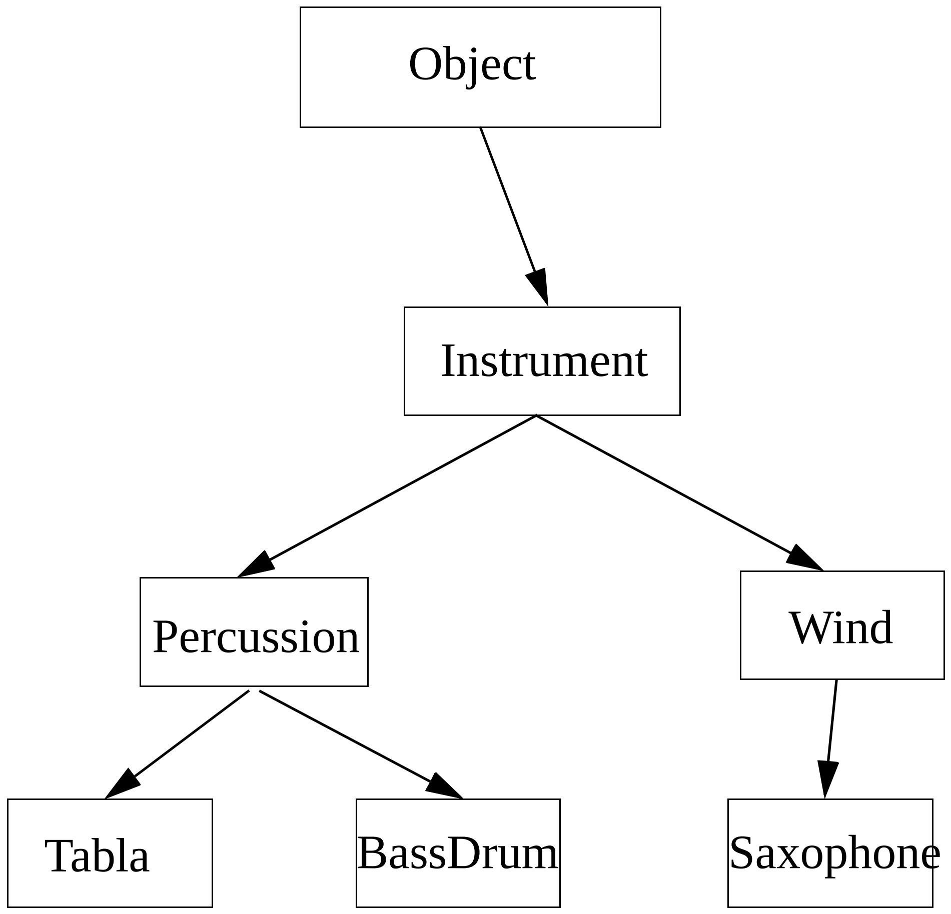
Dynamic method binding explored

- ✔ Suppose a base class has a public method which is overridden in derived classes...

```
   public class Instrument {
      public void play() { /* do something generic */ }
   }
--------------------------------------------------------------
   public class Percussion extends Instrument {}
--------------------------------------------------------------
   public class Tabla extends Percussion {
      public void play()  { /* play like a tabla */ }
   }
--------------------------------------------------------------
   public class BassDrum extends Percussion {
      public void play()  { /* play like a bass drum */ }
   }
--------------------------------------------------------------
   public class Wind extends Instrument {}
--------------------------------------------------------------
   public class Saxophone extends Wind {
      public void play()  { /* play like a saxophone */ }
   }
```

Derivation and the Java type system

- ✔ Inheritance between classes implies an “is-a” relationship, which can be shown in a class hierarchy diagram:



Derivation and the type system, cont'd

- ✔ This is reflected in the Java type system:
  - ✗ Java will permit an object created as an instance of class T to be pointed to by a reference variable of type T, or of any type that is an ancestor of T
  - ✗ But such an object can never be pointed to by a reference variable of any other type
- ✔ Given the class hierarchy shown, then...

```
Instrument i1, i2;
i1 = new BassDrum();  // okay, a BassDrum is-a Instrument
i2 = new Saxophone();  // okay, so is a Saxophone

Wind w;
w = new Tabla();      // ERROR!  a Tabla is not a Wind
```

- ✔ This relationship can be made explicit with a cast (called an "upcast": casting from a class to an ancestor class), but since Java will do it automatically there is never a real need to:

```
Instrument i3;

i3 = (Instrument) new Tabla();   // upcast
```

Calling overridden methods: dynamic binding

- ✔ Now what happens if you call the method **`play()`** accessing different type objects through pointers of the same type?
- ✔ Because of dynamic method binding, it does the “right thing”, invoking the **`play()`** method of the object pointed to:

```
Instrument i1, i2, i3;
i1 = new BassDrum();  i2 = new Saxophone();  i3 = new Tabla();

i1.play(); // plays like a BassDrum
i2.play(); // plays like a Saxophone
i3.play(); // plays like a Tabla
```

Limits to dynamic method binding

- ✔ Java always uses “dynamic binding” (also called “virtual” or “late binding”) of public instance method calls: the method called depends on the *type of the actual calling object* the pointer points to, not the type of the pointer
- ✔ But the method called must first be available in the class that is the type of the pointer, either by being defined in that class or by inheritance.
- ✔ The compiler will check this!

```
 Object o = new Saxophone();  // okay, a Saxophone is-a Object
 o.play();  // compile-time error: play() is not a method of Object
```

- ✔ You can convert the type of a pointer expression to a descendant type with an explicit cast (called a “downcast”: casting from a class to a descendant class):

```
Instrument i1 = o; //compile-time error:  Object isn’t a Instrument
Instrument i2 = (Instrument) o;  // okay with downcast
```

- ✔ Downcasts are always accepted by the compiler, but they will throw a ClassCastException at runtime if the object pointed to is not actually of the cast-to type!
- ✔ With a downcast, you can access members of the descendant type:

```
((Instrument)o).play();  //okay... play() is a method of Instrument
```

Next time

- ✓ Constructors, inheritance, this() and super()
- ✓ Instance methods, inheritance, and overriding
- ✓ Java’s Object class
- ✓ Overriding vs. overloading
- ✓ Final classes
- ✓ Static members and inheritance
- ✓ Instance variables, inheritance, and variable hiding

(Reading: Savitch, Ch. 7)