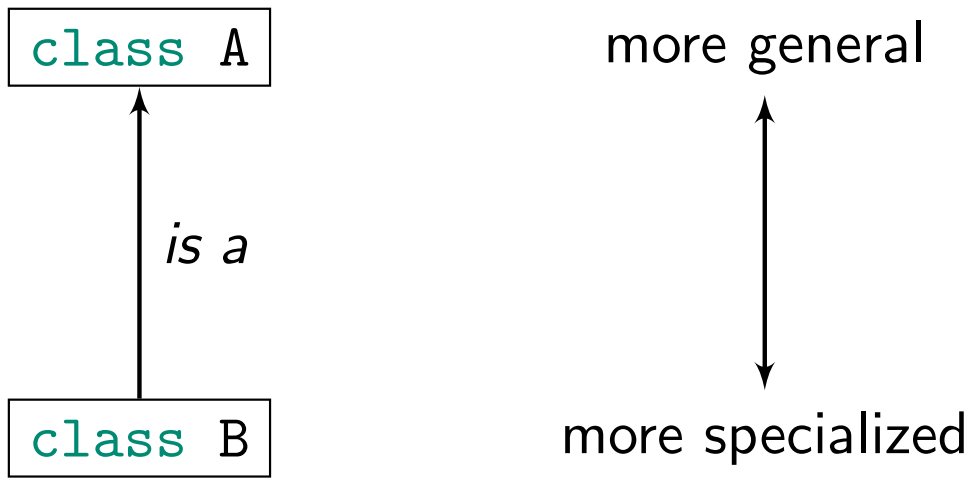
Organization of Classes

Java classes are organized structurally in a hierarchy or tree with the class `Object` (cf the API) as the ancestor or root of all classes



Organization of Classes

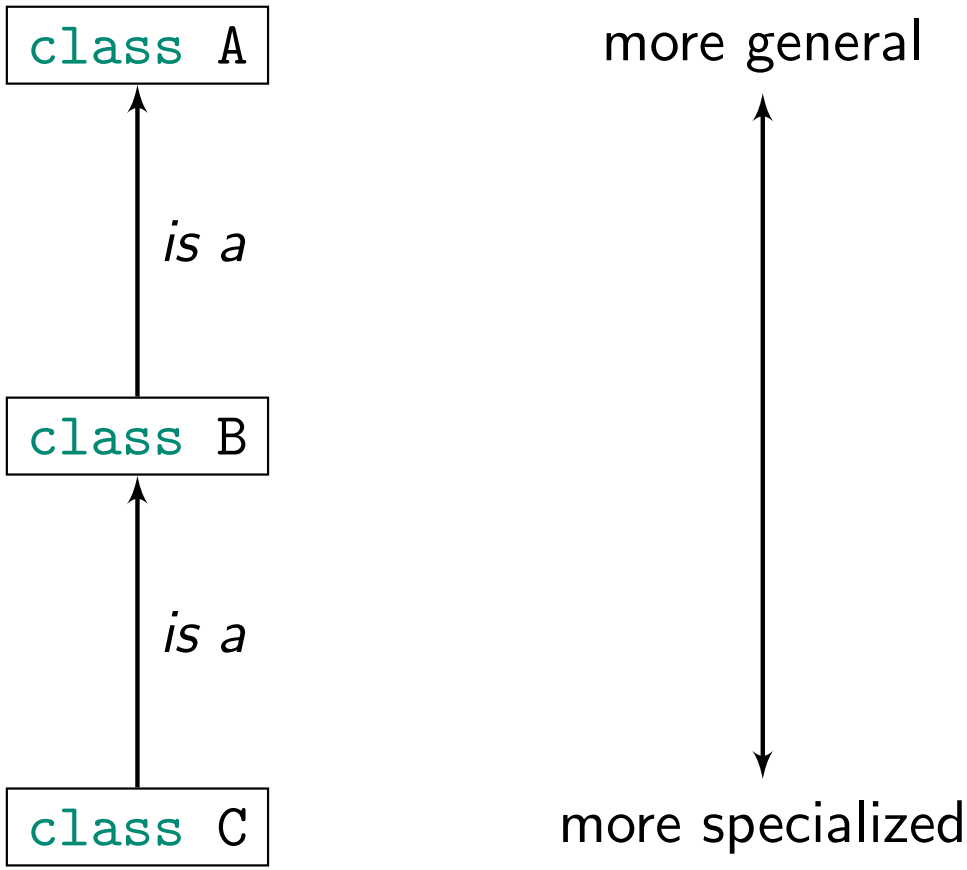
The relationship between two classes is thought of as being

"is a"

— as in a pencil *is a* kind of writing instrument.

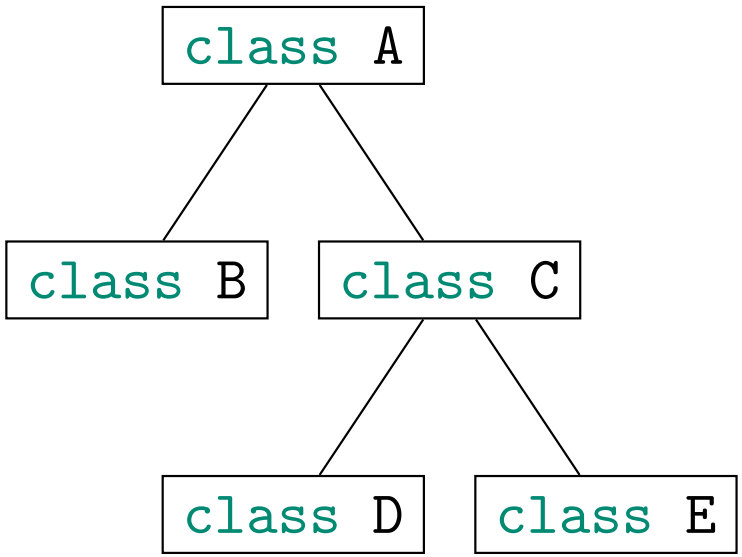
The wider more general concept (writing instrument) contains all of the more specialized items (all pencils) plus potentially a lot more (fountain pens, chalk, and so on).

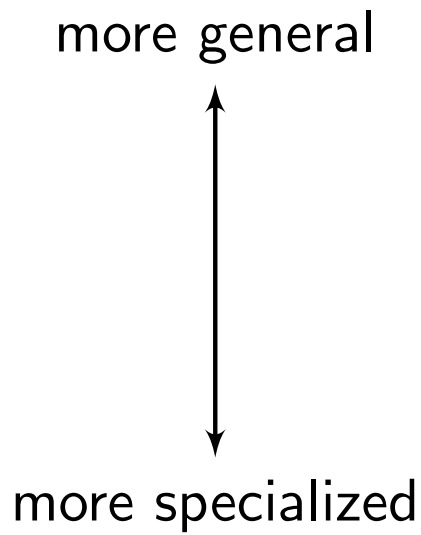
Organization of Classes



Any number of levels in the hierarchy.

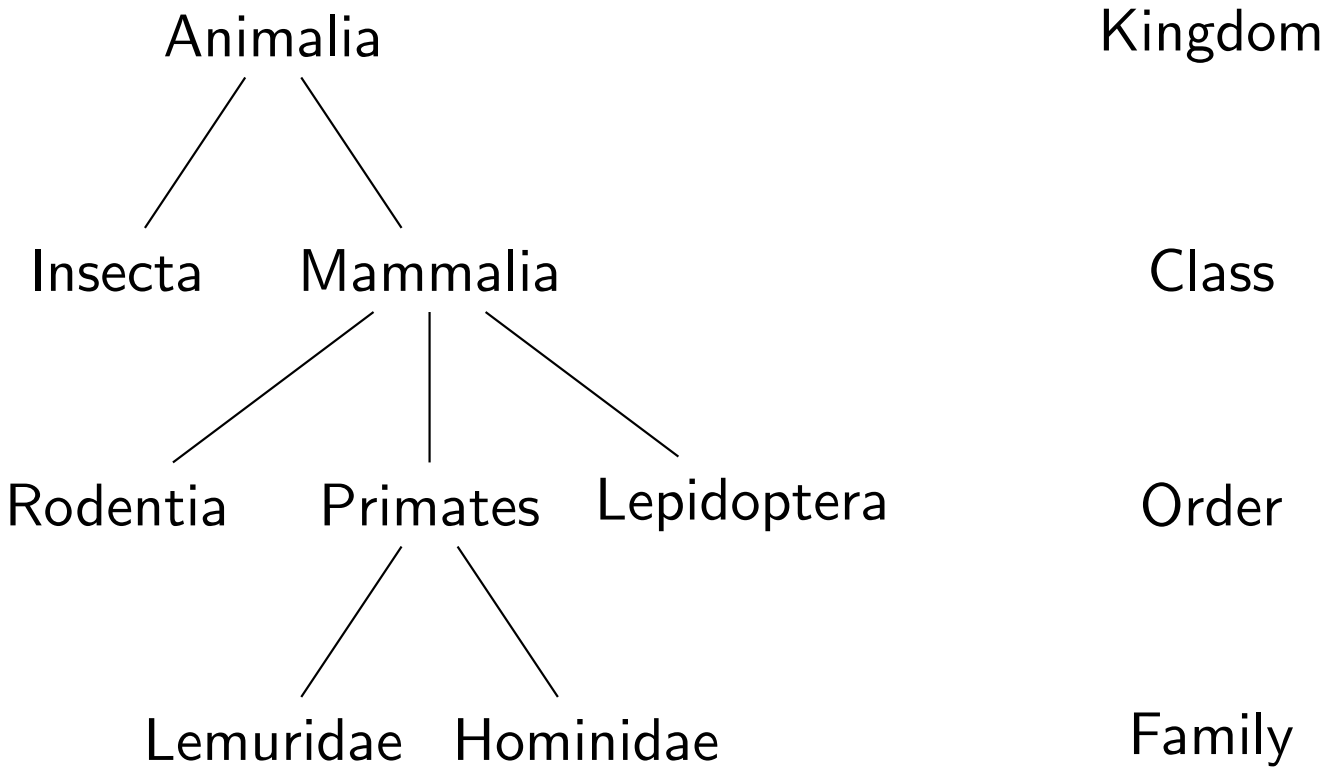
Organization of Classes



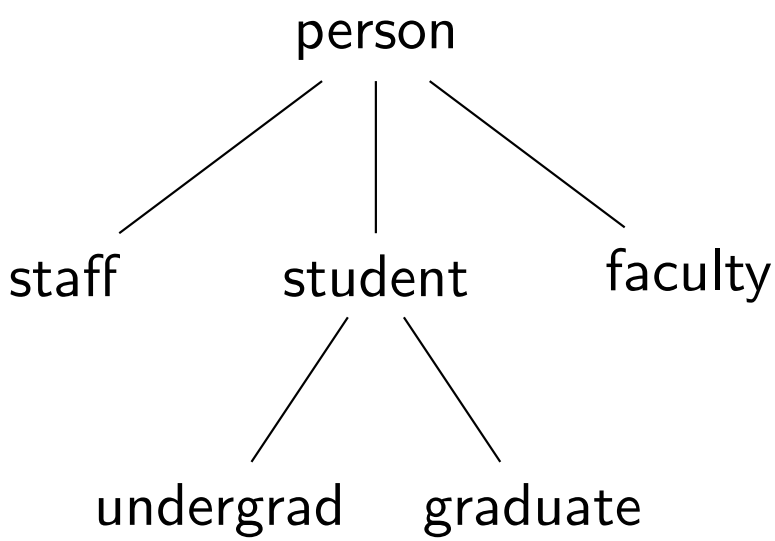


Each class has one superclass; but any number of subclasses can have the same superclass.

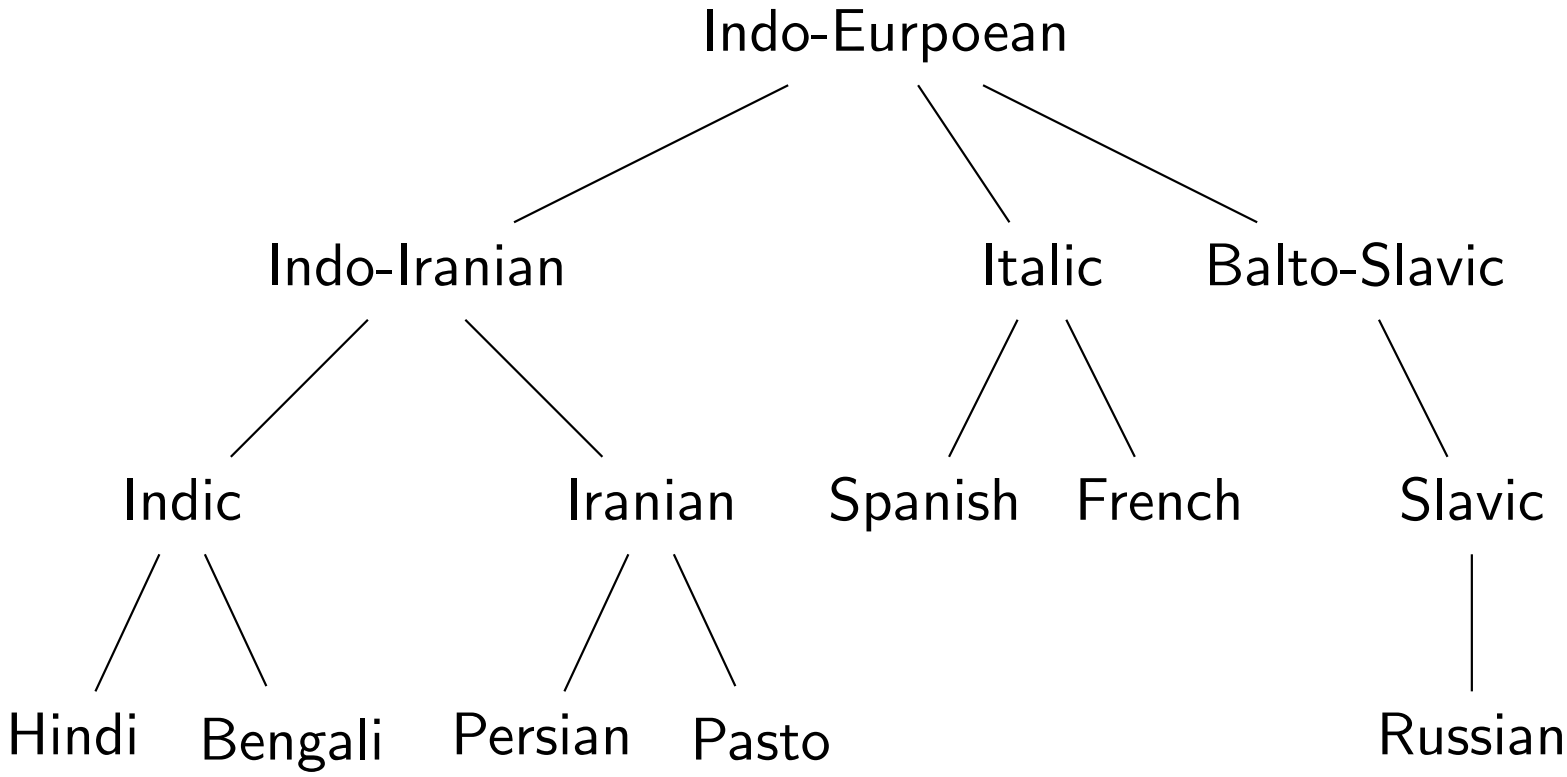
Example: Biological Classification



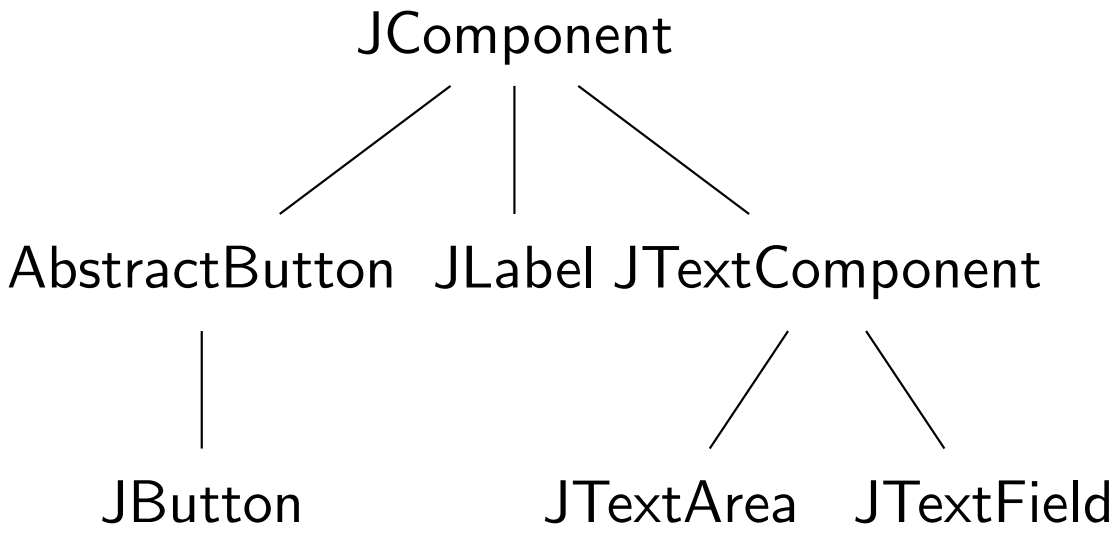
Hierarchical Organization



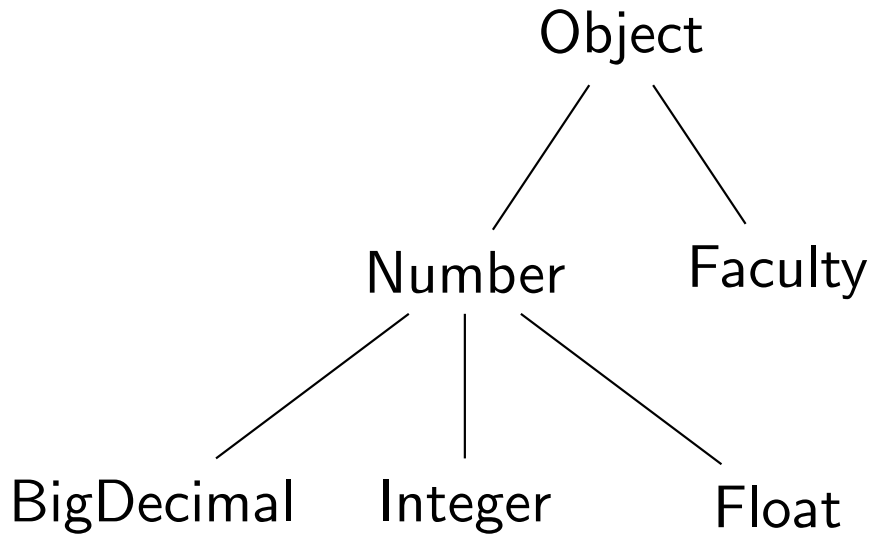
Hierarchical Organization



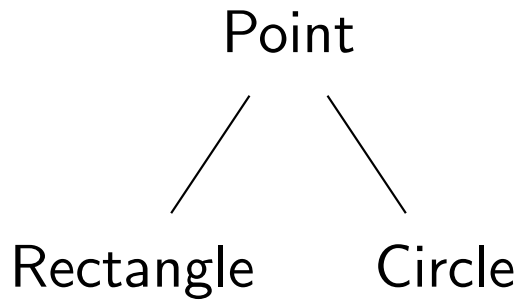
Hierarchical Organization



Hierarchical Organization



Hierarchical Organization



Class Hierarchy

The class hierarchy is a tree. A *tree* is a kind of structure with a root and the other elements are organized so that each element has one branch connecting it to the root.

1. Every class descends from the class `Object` (the root of the tree).
2. Every class has exactly one superclass (except the class `Object`).
3. No class can descend directly or indirectly from itself.

extends

In Java, the relation or organization of classes is made explicitly by name by the programmer.

```
class X extends Y {
}
```

The class `X` is declared a subclass of the class `Y` using the `extends` keyword. The `extends` clause is optional and if ommitted then a class is declared to be a direct subclass of `Object`.

Hierarchical Organization

```
class IndoEuropean { // ...
class IndoIranian extends IndoEuropean { // ...
class Indic extends IndoIranian { // ...
class Hindi extends Indic { // ...
class Bengali extends Indic { // ...
class Iranian extends IndoIranian { // ...
class Persian extends Iranian { // ...
class Pasto extends Iranian { // ...
class Italic extends IndoEuropean { // ...
class Spanish extends Italic { // ...
class French extends Italic { // ...
class BaltoSlavic extends IndoEuropean { // ...
class Slavic extends BaltoSlavic { // ...
class Russian extends Slavic { // ...
```

No Multiple Inheritance

```
class X extends Y, Z {
}
```

(But Java interfaces can be used to play this role.)

No Cyclic Inheritance

```
class X extends Y {
}

class Y extends X {
}
```

Hierarchical Organization

Sometimes the problem domain is naturally organized in a tree-like hierarchy. Sometimes the problem domain is *not* naturally organized like that.

In object-oriented programming we eventually learn the idioms or design patterns to solve different problems using this organization.

First, we must learn more of the structure Java provides for object-oriented programming.

Note that each class forms an interface, a suite of facilities or methods.

Interface. In general, an *interface* is the boundary between distinct systems. Specifically, the specification or protocol governing their interaction.

Note that Java uses the keyword `interface` and has a construct called an interface.

Polymorphism

What is the advantage of organizing classes in a tree structure?

Polymorphism

What is the advantage of organizing classes in a tree structure?

The answer is flexibility which we call subclass polymorphism. (Polymorphism is a word meaning *many forms*.) An object or instance of a class can be viewed as having more than one type (form).

Subclass Polymorphism

Any object can be viewed as being a kind of `Object`. (Since `Object` is at the top of the hierarchy.) This mean it has the collection of methods or interface as does any `Object`.

Object Is A Special Class
The Top of the Hierarchy

```
class Object {
    public String toString ();
    public boolean equals (Object obj);
    protected Object clone ();    // copy
    public Class<?> getClass (); // meta informati
    public void notify ();        // synchronizatio
    public void wait ();
}
```

For example, assignment

```
Object obj;
Number num;

obj = new String ();      // string "is-a" object
obj = new Integer (4);
obj = new Float (4.0f);
obj = new ArrayList ();  // ArrayList "is-a" objec
obj = new int [4];        // int array "is-a" objec

num = new Integer (4);
num = new Float (4.0f); // Float "is-a" Number
num = new BigDecimal (4.0d);
num = new Double (7.0d);// Double "is-a" Number
num = 4.0d;               // double is a Number
```

Subclass Polymorphism

```
Object[] objArray = new Object[5];
Number[] numArray = new Number[5];

objArray[0] = new String ();      // string "is-a"
objArray[1] = new Integer (4);
objArray[2] = new Float (4.0f);
objArray[3] = new ArrayList ();   // ArrayList "is-
objArray[4] = new int [4];        // int array "is-

numArray[0] = new Integer (4);
numArray[1] = new Float (4.0f);   // Float "is-a" N
numArray[2] = new BigDecimal (4.0d);
numArray[3] = new Double (7.0d);// Double "is-a"
numArray[4] = 4.0d;               // double is a Nu
```

Polymorphism

```
Number num = new Number[5];

num = new String ();   // a string is NOT a Number
num = new ArrayList(); // an ArrayList is NOT a Nu
num = new int [4];     // an int array is NOT a Nu
num = new Object();    // an object is NOT a Numbe
```

Compile-time, semantic error

```
incompatible types
```

Subclass Polymorphism

Substitution Principle. A variable of a given type may be assigned a value of any subtype of that type, and a method with a parameter of a given type may be invoked with an argument of any subtype of that type.

Subclass Polymorphism

The flexibility only works one way.

```
Object  o = new Integer (4);// OK
Integer i = new Object ();  // Semantic Error: in
```

And remember, primitive types are not technically classes. Yet:

```
Object  o = 4;  // autoboxing
Integer i = 4;  // autoboxing
Number  n = 4;  // autoboxing
int i = new Integer (4);  // auto-unboxing
int i = new Object ();  // compilation error
```

Another Example

An instance of a subclass "is-a" instance of the superclass.

```
class Main {
    public static void Main (String[] args) {
        IndoEuropean[] languages = new IndoEurpoean
        languages[0] = new Hindi ();
        languages[1] = new Persian ();
        languages[2] = new Spanish ();
        languages[3] = new French ();
        languages[4] = new Russian ();
    }
}
```

Another Example

```
import java.math.BigDecimal;

public class NumberMain {

    public static long add (Number n1, Number n2) {
        return n1.longValue() + n2.longValue();
    }

    public static void main (String[] args) {
        // BigDecimal and Long are each a Number.
        System.out.println (add (
            new BigDecimal ("32.1"), 34L));
    }
}
```

Vocabulary

extend. To make a new class that inherits the members of an existing class.

superclass. The parent or base class. "Super" in the sense of "above" not "more."

subclass. The child or derived class that inherits or extends a superclass. It represents a subpart of the universe of things that make up the superclass.

inheritance. A subclass implicitly has the member fields and methods of a class by virtue of extending that class.

Important terms coming up: *overriding*, and *dynamic dispatch*.

Extend

How do you extend another class in Java?

```
class SubClass extends SuperClass {
    // additional fields ...
    // constructors ...
    // additional methods ...
}
```

If the `extends` clause is omitted from a class, then it is as if you have extended the class `Object`.

Extend

So,

```
class SubClass  {
    // ...
}
```

is the same as:

```
class SubClass extends Object {
    // ...
}
```

It follows, that every class has:

```
public String toString ();
public boolean equals (Object obj);
protected Object clone ();    // copy
public Class<?> getClass (); // meta information
public void notify ();        // synchronization c
public void wait ();
```

Polymorphism

Conundrum: how can one class also be another class at the same time?

Answer: the interface of the superclass must also be the interface of the subclass. Every thing the superclass can do, the subclass can do as well. If the superclass has a method `int getX()`, then the subclass must also have method `int getX()`.

Therefore: the subclass inherits all the member methods and fields of the superclass.

An instance of a subclass “is-a” instance of the superclass.

```
class SuperClass { int x; }
class SubClass extends SuperClass { }

class Main {
    public static void main (String[] args) {
        SuperClass[] a = new SuperClass [2];
        a[0] = new SuperClass ();
        a[1] = new SubClass ();
        for (int i=0; i<a.length; i++) {
            System.out.println (a[i].x);
        }
    }
}
```

toString()

Every object has a `toString()` method!

```
class SuperC { int x; }
class SubClass extends SuperC { }
class Main {
    public static void main (String[] args) {
        Object[]a={new Object(),new SuperC(),new SubCla
        for (int i=0;i<a.length;i++) {
          System.out.println (a[i].toString());
        }
    }
}
```

By the way, the output is not very specific:

```
java.lang.Object@16930e2
SuperC@108786b
SubClass@119c082
```

Overloaded println()

The implementation of the java.io.PrintStream class:

```
void print (Object o) {print(o.toString());}
void print (boolean b){print(String.valueOf(b));}
void print (char c) {print(String.valueOf(c));}
void print (int i)  {print(String.valueOf(i));}
void print (long l) {print(String.valueOf(l));}
void print (float f) {print(String.valueOf(f));}
void print (double d) {print(String.valueOf(d));}

void print (String s) {
  // Do the real print work
}
```

Fields are Inherited

```
class Point {
    int x,y;
}

class Circle extends Point {
    int radius;
}

class Main {
    public static void Main (String[] args) {
        Circle c = new Circle ();
        System.out.printf ("%d,%d,%d%n",
            c.x, c.y, c.radius);
    }
}
```

Methods are Inherited

```
class Point {
    int x, y;
    void move (int dx, int dy) { x += dx; y += dy;
}
class Circle extends Point {
    int radius;
}
class Main {
    public static void Main (String[] args) {
        Circle c = new Circle ();
        c.move (2,3);
        System.out.println (c.x +", "+ c.y +", "+ c
    }
}
```

Fields Can Be Hidden

```
class SuperClass {
    int x, y;
}

class SubClass extends SuperClass {
    int x, y;
}
```

The class `SubClass` has two fields named `x` and two fields named `y`.

```
class SuperClass {
    int x=2;
}

class SubClass extends SuperClass {
    int x=super.x+1;
}

class SubSubClass extends SubClass {
    int x=((SuperClass)this).x+3;
}
```

If the integer `x` in the class `SuperClass` is declared `private`, then access to it from a subclass causes a compile-time, semantic error.

Static Methods

You can use the name of the subclass to access static methods of the superclass. (Not so terribly important.)

```
class IndoEuropean {
    static void info () {
        System.out.println ("To find out more ...")
    }
}
class German extends IndoEuropean {}
class Main {
    public static void Main (String[] args) {
        IndoEuropean.info ();
        German.info ();
    }
}
```

It might be better to always use the class name `IndoEuropean` when accessing the method `info()`, to show where to actually find the code.

Constructors *Not* Inherited

Constructors are not class members; they are not inherited.

Constructors and super

Default constructor. "If a class contains no constructor declarations, then a default constructor that takes no parameters is automatically provided."

```
class Point {
    int x, y;
}
```

is equivalent to the declaration

```
class Point {
    int x, y;
    Point() { super(); }
}
```

"A compile-time error occurs if a default constructor is provided by the compiler but the superclass does not have an accessible constructor that takes no arguments."

Pitfall: Constructors and Subclasses

```
class Super {
   final int i;
   Super (int i) { this.i = i; }
}

class Sub extends Super { }   // Illegal!
```

Methods Can Be Overridden

Sometimes the behavior of inherited methods is close, but not quite right for the subclass. In these cases it is appropriate to *override* the method.
A subclass overrides a method by defining a method of the same name and signature.

```
public String toString()
```

"A class type may contain a declaration for a method with the same name and the same signature as a method that would otherwise be inherited from a superclass. In this case, the method of the superclass is not inherited. The new declaration is said to override it."

Dynamic Dispatch

Dynamic dispatch (aka single dispatch, aka virtual function call)
In most OO systems, the concrete function that is called from a function call in the code depends on the type of a single object at runtime.

Dynamic Dispatch

A simple example: `class/Dispatch.java`

Overriding

The following is not so very important, but everyone asks. [Don't ask because if you do, either:

- your OO design is bad,
- you don't understand the subclass contract, or
- you have been looking at C++.]

What if you want some particular method to be called. You don't want the method in the subclass called, but the method somewhere up in the subclass hierarchy.
Casting does not help for methods

- `class/AccessField.java` – casts make a different
- `class/AccessMethod.java` – casts make no difference

Object-Oriented Design

1. Identify the problem's objects.
   - 1.1 If the object cannot be directly represented using the existing types, then design a class to do so.
   - 1.2 If two or more classes share common attributes, then design a hierarchy to store their common attributes
2. Identify the problem's operations.
   - 2.1 Define a method to do the operations.
   - 2.2 Structure the method within the class hierarchy so as to take advantage of inheritance
   - 2.3 Where necessary, have subclasses override inherited definitions.
3. Solve the problem

Attribute or Method

If a value depends on parameter, then use a method. If a value needs to be computed (like with a random number generator), then use a method. If no behavioral variation, then use an attribute. Inherited attributes, might be private and given access through getter and setter methods.

```
private int x;
protected int getX () {return x;}
protected void setX (int i) {x=i;}
```

This way the validity of any new value can be checked

Calling Procedure

- `call P(a)` – compiler looks up address of `P` and jumps to instruction
- `call P(a)` – (overloading) compiler chooses from among several procedures based on the static types of arguments
- `o.P(a)` – (dynamic dispatch) the runtime system chooses from among several procedures based on the subtype of object o

Note that static type checking is possible in all cases.

Casting

Upcast or widening — OK
Downcast or narrowing — dangerous
Narrowing often must be used in OO languages.

Narrowing

Coercions can be classified into those that preserve information (*widenings*) and those that lose information (*narrowings*). The coercion `int` to `long` is a widening; `int` to `short` is a narrowing. See the table of Java coercions at

`/ ryan/java/language/java-data.html`

The terms apply to the OO hierarchy as well. OO programming often requires narrowing which defeats the purpose of strong typing. See the Java program example:

`misc/Points.java class/Widening.java`

Casting Classes Summary

```
class Mammal {}
class Dog extends Mammal {}
class Cat extends Mammal {}
Mammal m = (Math.random()<0.5)?new Mammal():new D
Dog spot = new Dog();
Cat felix = new Cat();

m = spot;   m = felix; // Valid (no cast needed)
spot = m;              // Compile-time error
spot  = (Dog) m;       // Valid at compile time; r
felix = (Cat) m;       // Valid at compile time; r
felix = spot;          // Compile-time error
felix = (Cat) spot;    // Compile-time error
```

Wary programmer:

```
if (m instanceof Cat) {
  felix = (Cat) m;
}
```

Runtime system:

```
if (m instanceof Cat) {
  felix = (Cat) m;
} else {
  throw new ClassCastException ();
}
```

Abstract Classes

An abstract class is a class that has some abstract methods. Abstract methods have a specification but lack code/instructions/behavior. An abstract class cannot be created/instantiated (but can have constructors), but can be used as a superclass to define a subclass.

`abstract`

In a class hierarchy, If a method's behavior depends the class, it is natural to override it. But if some class has no special behavior for the method, then there are two choices.

1. Define a meaningless or generic default behavior and let subclass override it. (Think of `toString()` for `Object`.)
2. Declare the method `abstract`

If you declare a method `abstract` in a class, then the class is `abstract`. Meaning that the class is not used for instantiation, but only for defining other classes.

Subclass responsibility. All (non-abstract) subclasses are given the requirement (not just the opportunity) that the method be overridden.

```
class Object {
    public String toString () { return "Object"; }
}
class SubClass extends Object {
    int x;
    public String toString () { return Integer.toS
}

abstract class Object {
    abstract public String toString ();
}
class SubClass extends Object {
    int x;
    public String toString () { return Integer.toS
}
```

Interface

An interface is a set of methods describing functionality common across several classes.
Interfaces are totally abstract classes.
Advantage: can implement as many of them as you like.
Disadvantage: can't implement code.
Used as (rather poor) enumeration types. Important in threads.
Important as callbacks (especially in GUI code). Important in collection classes.

Interface

```
interface/Verbose.java
interface/List.java
interface/Example.java
interface/PointPack.java
misc/Reactive.java
```

Common Interfaces

- interface `Comparable`.
- interface `Comparator`.
- interface `Iterator`.
- interface `Runnable`.
- marker interface `Serializable`.
- marker interface `Cloneable`.

A marker interface has no methods; hence any class can "implement" it. It is used as a signal to the JVM. A class the implements such an interface is allowed to be serialized, cloned, etc.

Nested Classes

```
inner/Nest.java
inner/Local.java
inner/Iter.java
misc/Anon.java
inner/Searcher.java
```

Equals

`equals/Main.java`
`equals/Override.java`

Clone

Why clone? Because assignment just copies references. This creates aliases and this leads to programming mistakes.

```
BankAccount ba153 = new BankAccount (500);
BankAccount ba714 = ba153;    // sharing
ba153.deposit (25);           // both get deposit!

BankAccount ba153 = new BankAccount (500);
BankAccount ba714 = ba153.clone(); // a copy
ba153.deposit (25);             // only one deposit
```

Clone

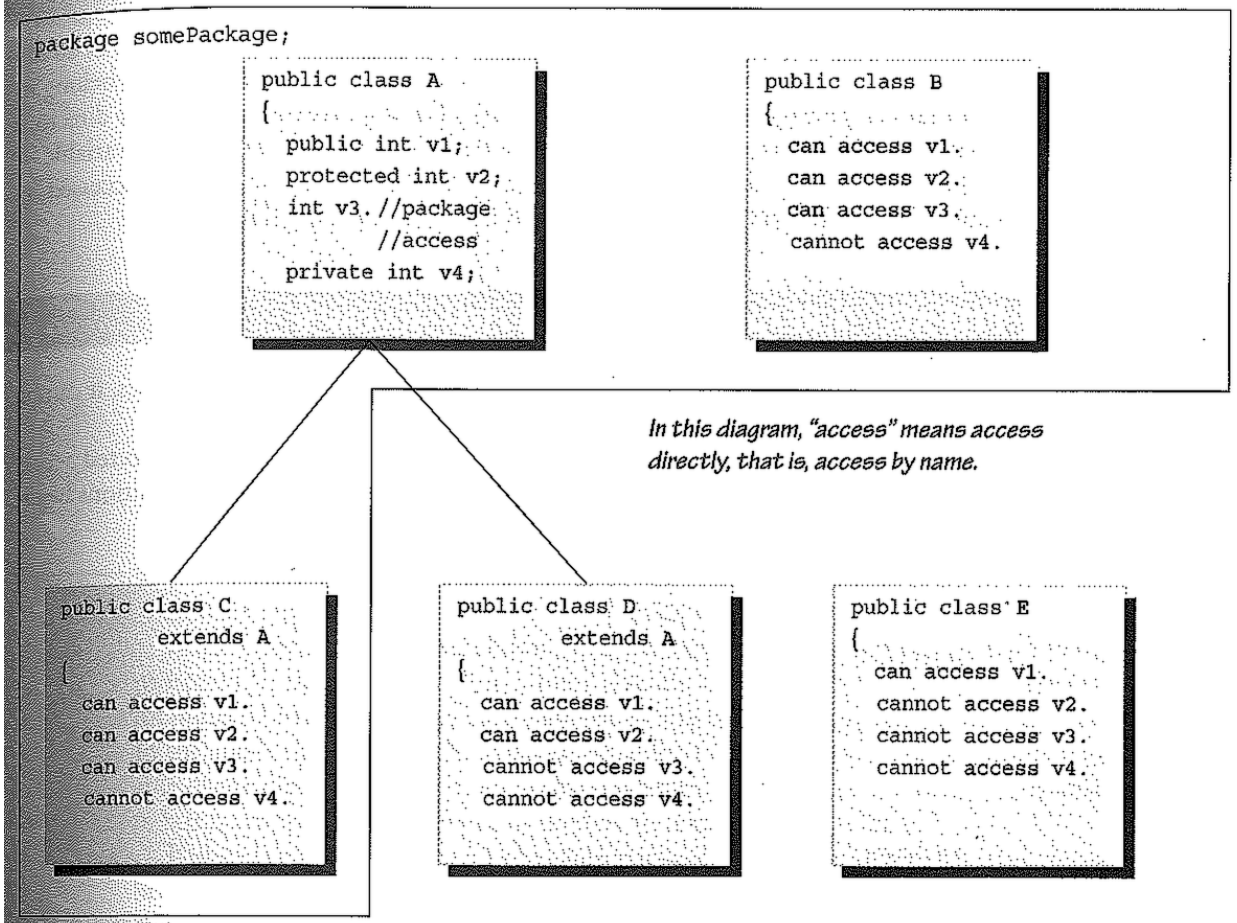
`class/Clone.java`

Superclass does the work and even copies the added fields (`x`). The class must be marked `Cloneable` or the unchecked exception `CloneNotSupportedException` will be raised. The `protected` method `clone` is overridden with a `public` method. This is permitted. You can override with less restrictive access, but not more restrictive access.

- *private*—members declared `private` are only accessible within the class itself.
- *"package"*—members declared with no access modifier are accessible in classes in the same package.
- *protected*—members declared `protected` are accessible in subclasses (in the same package or not) and in the class itself.
- *public*—members declared `public` are accessible anywhere the class is accessible.

access from	`private`	"package"	`protected`	`public`
same class	yes	yes	yes	yes
in subclass, same package	no	yes	yes	yes
non-subclass, same package	no	yes	yes	yes
in subclass, out of package	no	no	yes	yes
non-subclass, out of package	no	no	no	yes

Display 7.9 Access Modifiers



Overriding: same name, different classes, same signature, at least as much access (cf. §8.4.8.3 JLS 3rd).

$$\texttt{private} < \text{“package”} < \texttt{protected} < \texttt{public}$$

```
class Restrictive {
    // Semantic error!
    // "attempting to assign weaker access privile
    private boolean equals (Object x) {
        return false;
    }
    // OK.  But, overloading not overriding!!
    private boolean equals (Restrictive x) {
        return true;
    }
}
```

Is-a versus Has-a

Design consideration.
It is easy to misuse inheritance. “is-a” or “has-a”.
Aspects.

- `aspect/Point.java`
- `aspect/SubPoint.java`
- `aspect/Aspect.java`

Summary

- *class hierarchy*
- *subtype polymorphism*
- *inheritance*
- *overriding*
- *dynamic dispatch*
- Java's *abstract classes*
- Java's *interfaces*