Organization of Classes

Java classes can be arranged in a large program in different ways. Classes can be in packages, local to methods, or members of other classes. This type of program and class organization is an important part of developing a large program.
There is another important, but completely different, way of organizing classes. It is possible to organize classes by the behavior of their instances and to take advantage of common behavior in objects in a program. By organizing classes in the *class hierarchy* one can increase flexibility and code reuse.

Kinds of Organization

Classes can be organized by their placement in files.

1. Classes are found inside other classes:
   - As members (inside the {} of another class), usually static, and
   - Inside a method (of a class), sometimes anonymously.
2. Classes can be grouped (organized) in directories called packages.
3. Java modules (introduced in Java 9) provides a level of aggregation of classes above packages. Modules support stronger encapsulation.

Package

Definition

A *package* in Java is an *ad hoc* collection of classes in a directory/folder of the operating system allowing certain, not quite public, access to members.

A class is identified as belonging to a package, by a package declaration at the beginning of the class source file:

```java
package dir1.dir2;
```

Also the source file must be located in the directory.

```
dir1/dir2  # Unix
dir1\dir2  # Windows
```

Module

A Java module can specify which of the Java packages it contains that should be visible to other Java modules which uses this module. A Java module must also specify which other Java modules is requires to do its job.
Large libraries, like the entire Java platform APIs, can be separated into pieces and only the required pieces needed to be deployed. Missing modules can be reported at application start-up time and not, like classes, until the application actually tried to use it.

Some Modules

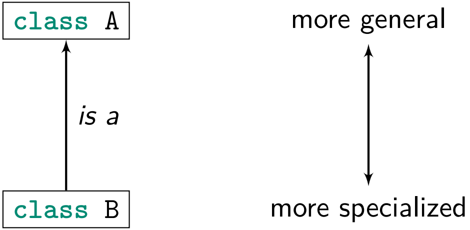
```
$ java -list-modules
java.base@16.0.1
java.compiler@16.0.1
java.instrument@16.0.1
java.management@16.0.1
java.naming@16.0.1
java.net.http@16.0.1
java.rmi@16.0.1
java.scripting@16.0.1
java.se@16.0.1
java.security.jgss@16.0.1
java.smartcardio@16.0.1
java.sql@16.0.1
```

Subclass Hierarchy

Our main topic here is the organization of classes in the Java subclass hierarchy.

Organization of Classes

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



Organization of Classes

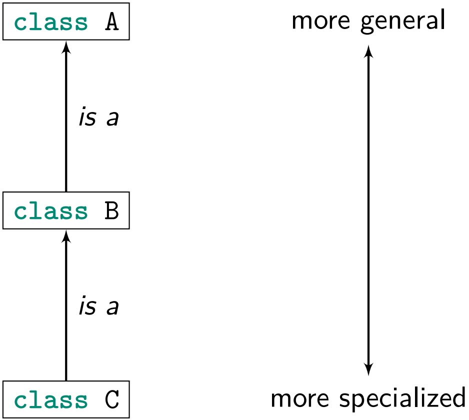
The relationship between two classes is thought of as being

"is a"

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

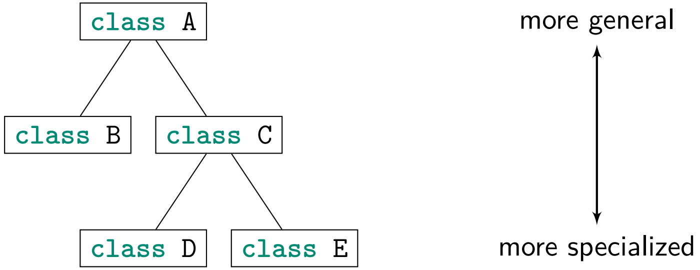
The wider more general concept (writing instrument) contains all of the more specialized items (all pencils) plus potentially a lot more (fountain pens, chalk, and so on).

Organization of Classes



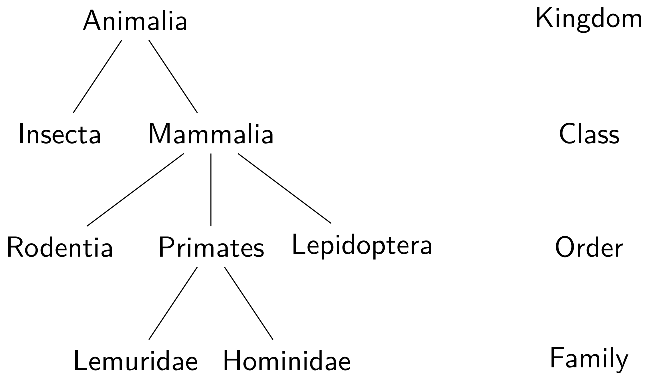
Any number of levels in the hierarchy. And no cycles.

Organization of Classes

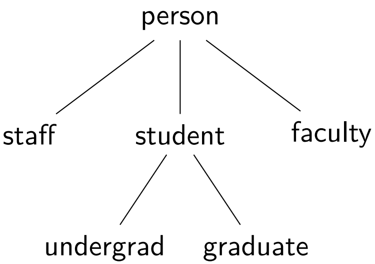


Each class has one superclass; but any number of subclasses can have the same superclass.

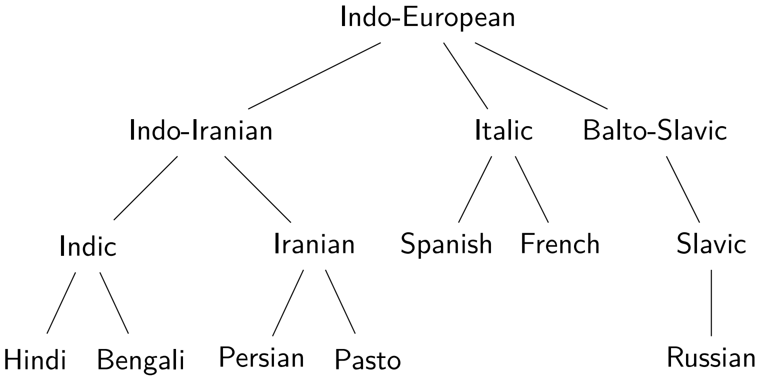
Example: Biological Classification



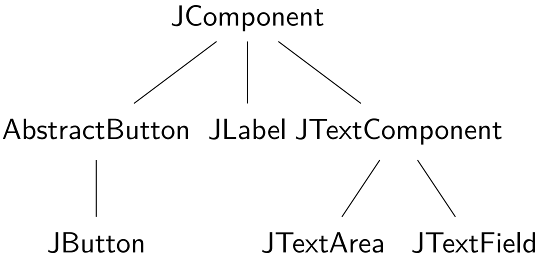
Hierarchical Organization



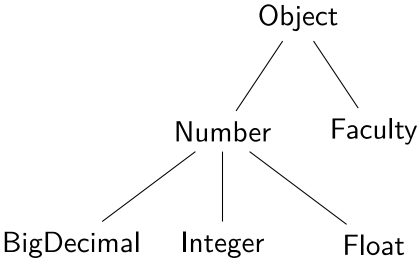
Hierarchical Organization



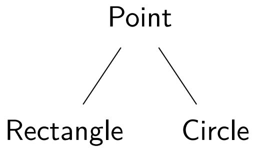
Example: Hierarchical Organization from Java API



Example: Hierarchical Organization



Hierarchical Organization



Class Hierarchy

The Java 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.

Nominal Subtyping

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

```java
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 omitted then a class is declared to be a direct subclass of `Object`.

It is calld *nominal subtyping* when the position in the class hierarchy is determined explicitly by the programmer using the names, like the edges of a graph.

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 Superclasses

```
// Not syntactically correct!
class X extends Y, Z {
}
```

This is not allowed because of the conflicts it causes–like having two bosses that require you to do two different things.

No Cyclic Structure

```
// Not semantically correct!
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.
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.

Subclass Polymorphism

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

java.lang.Object Is A Special Class
The Top of the Hierarchy

```
class Object {
   public String toString ()
   public boolean equals (Object obj)
   public int hashCode ()        // encoding as integer
   protected Object clone ()     // copy
   public Class<?> getClass ()  // meta information
   public void notify ()         // synchronization of threads
   public void wait ()           // synchronization of threads
}
```

java.utils.Objects

```
class Objects {
   static boolean equals(Object a, Object b)
   static boolean deepEquals(Object a, Object b)
   static in hash (Object... values)
   static boolean isNull (Object)
   static boolean nonNull (Object)
   static <T> T requireNonNull (T obj)
   static <T> T requireNonNullElse (T obj, T default)
}
```

java.lang.Class
Reflection

These are mostly instant methods.

```
class Class<T> {
  Construct<?>[] getDeclaredConstructors()
  Field[] getDeclaredFields()
  Method[] getDeclaredMethods()
}
```

Other classes exist for referning to Java's constructors, field, methods and so on.
Also in class Class the interesting static method:

```
  static Class<?> forName (String className)
```

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).

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<String>(); // ArrayList "is-a" object
obj = new int [4];        // int array "is-a" object

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 sorta a Number (auto-boxing)
```

Polymorphism

```
Number num;

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

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.

Also called Liskov's substitution principle after 2008 Turing Award winner Barbara Liskov.


A.M.
TURING
AWARD
2008
BARBARA LISKOV
Developed the Liskov substitution principle
acm

Barbara Liskov (–)

When she was still a young professor at the Massachusetts Institute of Technology, she led the team that created the first programming language that did not rely on goto statements. The language, CLU (short for “cluster”), relied on an approach she invented — data abstraction — that organized code into modules. Every important programming language used today, including Java, C++ and C#, is a descendant of CLU.
In 2008, Liskov won the Turing Award.

Subclass Polymorphism

The flexibility only works one way.

```
Object  o=new Integer(4);// OK
Integer i=new Object();// semantic error: incompatible types
```

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 IndoEuropean [100];
      languages[0] = new Hindi ();
      languages[1] = new Persian ();
      languages[2] = new Spanish ();
      languages[3] = new French ();
      languages[4] = new Russian ();
   }
}
```

Another Example

```java
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 sub-part 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*.

Extending

How do you extend another class in Java?

```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`.

Polymorphism

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

Answer: the interface of the superclass must also be included in the interface of the subclass. Every thing the superclass can do, the subclass can do as well. If the superclass has a member field `x`, then the subclass must also have member field `x`. 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.

Constructors are *not* inherited.

Inheriting Member Fields

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 (SuperClass c: a) {
         System.out.println (c.x);
      }
   }
}
```

Inheriting Member Methods

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

```java
class Dog { void bark () { System.out.println ("bark"); }
class Poodle extends Dog { }

class Main {
   public static void main (String[] args) {
      Dog[] dogs = new Dog [2];
      dogs[0] = new Dog ();
      dogs[1] = new Poodle ();
      for (Dog d: dogs) {
         d.bark();
      }
   }
}
```

Extending

Since,

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

is the same as:

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

It follows, that every class has:

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

Inheriting the toString() method

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 SubClass()};
      for (int i=0;i<a.length;i++) {
         // Unnecessary call to 'toString()'
         System.out.println (a[i].toString());
      }
   }
}
```

Overloaded `print()`

The definition of print and println make an explicit call to `toString()` unnecessary. The call to `toString()` and the conversion from the primitive data types is by overloading the definition of `print()` and `println()`.

Overloaded print()

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
}
```

Overloaded `print()`

Ironically, printing `null` results in a compile-time error, because there are two choices.

```
System.out.print (null);   // ambiguous
```

Each prints the same thing: `null`.

```
System.out.print ((Object)null);
System.out.print ((String)null);
```

Inheriting the `toString()` method

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 SubClass()};
      for (int i=0;i<a.length;i++) {
         System.out.println (a[i]);
      }
   }
}
```

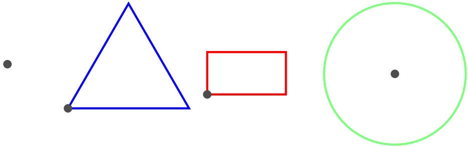
By the way, the output is not very specific:

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

More on that later.

Shapes Example

Suppose we want write a program to compute with points, circles, and rectangles. There are different ways to define the data structure of each of the shapes. One possible framework of definitions which may prove useful is to think of shapes as having a reference point.



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); // Circle inherited 'move()'
        System.out.printf ("%d,%d,%d%n", c.x, c.y, c.radius);
    }
}
```

Fields Can Be Hidden

```java
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`.
This is allowed because the author of the subclass should not have to know what names the author of the superclass might have picked. Forbidding this would enable the subclass author to “peek” inside the superclass.

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

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

class SubSubClass extends SubClass {
    // Can access beyond super class
    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 (final String[] args) {
      IndoEuropean.info ();
      German.info ();
      new German().info();  // Warning
   }
}
```

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

Using an instance for a static method is legal and has no advanatages at all.

Not a javac warning, but "lint."

```
$ javac -Xlint Info.java
Info.java:12: warning: [static] static method should be qualified by type
      new German().info();  // Warning
                  ^
1 warning
```

Not caught by CSE1002 checkstyle. Indeed, no such checkin checkstyle at all. Use `-Xlint`!

Constructors and super

Default constructor.

"If a class contains no constructor declarations, then a default constructor is implicitly declared."

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

is equivalent to the declaration

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

"It is a compile-time error if a default constructor is implicitly declared but the superclass does not have an accessible constructor that takes no arguments and has no throws clause."

Pitfall: Constructors and Subclasses

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

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

Java Language Rule: Each subclass constructor must implicitly or explicitly call one of its superclass's constructors. This is used to properly initialize the superclass including its instance fields. This is important to the subclass which inherits and may depend on the superclass's instance fields.

```
class Super {
  final int x;
  Super (int x) { this.x = x; }
  int sum () { return x; }
}
class Sub extends Super {
  final int y;
  Sub (int x, int y){super(x);this.y=y;}
  @java.lang.Override
  int sum () { return x+y; }
}
```

```java
class Super {
  final int x;
  Super (int x) { this.x = x; }
  int sum () { return x; }
}
class Sub extends Super {
  final int y;
  Sub (int x, int y) {this.y=y;}//Error
  @java.lang.Override
  int sum () { return x+y; }
}
```

Do not call instance (non-static) methods from constructors. Call either private or final methods from inside constructors. The reason is that Java uses dynamic dispatch and this could result in a call a method on a half-initialized object.

Anyway, constructors should focus on basic creation and initilization and not on complex computation.

Constructors *Not* Inherited

Constructors are not class members; they are not inherited.

Final

Methods marked `final` are inherited but cannot be overridden. Classes marked `final` cannot be extended.

See `Finality.java`

Bad Inheritance

Inheriting the interface is necessary for substitution to work. Inheriting the behavior or code of the methods can occasionally save keystrokes, but leads to subtle problems.

A problem with inheriting behavior is that the behavior of a method must often be specialized to fit with the subclass. It is naturally expected that the specialize behavior be used for all speciized objects. The consequences can be tricky to understand.

Methods Can Be Overridden

Sometimes the behavior of inherited methods is close, but not 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. For example,

```
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."

toString

> *Providing a good `toString` implementation makes your class much more pleasant to use and makes systems using the class easier to debug.*

Block, *Effective Java*, third edition, "Item 12: Always override `toString`," 2017, page 55

Overriding Example

`Point2D.java`

Overriding Example

Advice.java

Animals.java

Overriding and Covariance

[Covariance is an advanced topic.]

```java
class SuperA {
  public int getVal() { return 0; }
}

class SubA extends SuperA {
   // No covariance in the return type
   @java.lang.Override
   public int getVal() { return 1; }
}
```

Overriding and Covariance

[Covariance is an advanced topic.]

```java
class B {
    public SuperA foo() { return new SuperA(); }
}

class C extends B {
    // Demonstrates covariance
    @java.lang.Override
    public SubA foo() { return new SubA(); }
}
```

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.

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. No "method not found" errors because subtypes have the same interface as the supertypes.

Unique Names

```
class A{
  static void p (int x) {
     B.q(2,3);
  }
  static void t () {}
}

ckass C {
  void r (char c) {
     D.s(5);
  }
}
```

```
class B {
   static q (int a, int b) {
      C c = new C ();
      c.r ('z'); //instance
   }
}
class D {
   static s (int ) {
      A.t();
   }
}
```

Overloading

```
class A{
  static void p (int x) {
     B.q(2,3);
  }
  static void t () {}
  static int t (int i) {
     B.q(4);
  }
}
class C {
  void r (int t) {
     D.s (3.14);
  }
  void r (char c) {
     D.s(5);
  }
}
```

```
class B {
   static C c = new C();
   static q (int a, int b) {
      c.r ('z');
   }
   static q (int c) {
      c.r (5);
   }
}

class D {
   static s (int i)   { A.t(5); }
   static s (float f) { A.t();  }
}
```

Overriding Member Methods

```
class Dog { void bark () { System.out.println ("bark"); } }
class Poodle extends Dog { }
class Shih_tzu extends Dog {
   void bark(){System.out.println();}
}

class Main {
   public static void main (String[] args) {
      Dog[] dogs = {new Dog(), new Poodle(), new Shi_tzu()};
      for (Dog d: dogs) {
         d.bark();
      }
   }
}
```

Dynamic Dispatch

A simple example: `Dispatch.java`
A simple example: `Animals.java`
An example: `Test.java` (LineBuffer subclass).

Override Annotation

`@java.lang.Override`

An example: `Simple.java`

Overloading and Overriding

```java
public static class A {
    public void f(A a) { System.out.println("A1"); }
    public void f(B b) { System.out.println("A2"); }
    public void g(A a) { System.out.println("A3"); }
    public static void h(A a) { System.out.println("A4"); }
};
public static class B extends A {
    public void f(A a) { System.out.println("B1"); }
    public void f(B b) { System.out.println("B2"); }
    public void g(B b) { System.out.println("B3"); }
    public static void h(A a) { System.out.println("B4"); }
}
```

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

- `AccessField.java` – casts make a difference for fields
- `AccessMethod.java` – casts make no difference for methods

Casting Classes Summary

```
class Mammal {}
class Dog extends Mammal {}
class Cat extends Mammal {}
Mammal m = (Math.random()<0.5)?new Mammal():new Dog();
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; runtime check
felix = (Cat) m;      // Valid at compile time; runtime check
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 ();
}
```

But the use of `intanceof` and casts is a sign of poor design.

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), It is used as superclass to define a subclass with the responsibility of implementing the missing behavior.

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.

May override versus *must* override

First, *may* override:

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

Second, *must* override:

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

In both cases the override annotation can and ought to be used.

Abstract List

```
abstract class AbstractList implements List {
  // 1. Operations that are in common to all lists
  // 2. Operations that have different behavior,
  //    but common interface
}
```

In Java `List` is an interface—it describes what a list can do. In Java `ArrayList` is a class that implements `List`—it gives a specific way of implementing the operations. (`LinkedList` gives another way.) In Java `AbstractList` is a class that provide some functionality common to all lists. (All these Java classes and interfaces are generic.)

`AbstractList` also extends the abstract class `AbstractCollection` which implements the interface `Collection`.

Typical use of abstract classes to factor out commonality in mututally exclusive and exhaustive subclasses.

`Main2.java` – abstract class example

Advanced use of abstract classes in mutually recursive subclasses traversed using the visitor design pattern

`InheritV.java` – visitor design pattern

Interfaces

Head First Java: "A class defines who you are, and an interface tells what roles you could play."

Superpower: implements multiple interfaces.

Definition

An *interface* defines the communication boundary between two entities, such as a piece of software, a hardware device, or a user. It generally refers to an abstraction that an entity provides of itself to the outside.

Definition

A Java `interface` a construct describing the protcol (not the behavior) of instances of a class. It is somewhat like a contract beween the objects of the class and the users.

Interface

`ListExample.java` simple case
`Example.java` implement multiple interfaces
`PointPack.java` like serializatin
`Reactive.java` reactive programming

Common Interfaces

- interface `Comparable`.
- interface `Comparator`.
- interface `Iterable` (has an iterator).
- interface `Iterator` (optional remove).
- interface `ListIterator`
- interface `AutoClosable`.

Important Interfaces

Important interfaces to the Java programming language that we do not wish discuss.

- 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.

For Each Loop

```java
public interface Iterable<T> {
             Iterator<T>     iterator();
  default Spliterator<T> spliterator();
  default void               forEach(Consumer<? super T> action);
}
```

Note the special java syntax that takes advantage of the interface `Iterable`.

ForMain.java – Example for-each loop

Default

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

When you implement an interface, you may

- not mention the default method at all, which lets the class inherit the method and behavior,
- redefine the default method which overrides it.

Comparing Data

Why both `Comparable` and interface `Comparator`?

(Comparing doubles is not a good idea.)

`Main.java`
`MainR.java`

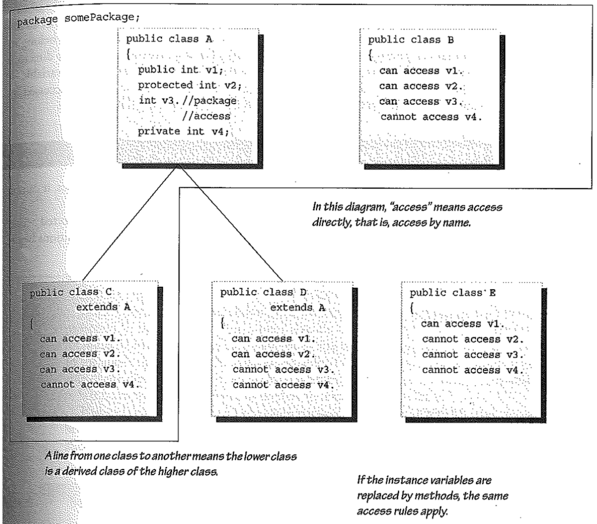
`ComparingRecords.java`

- *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 Control in Java language specification.

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, other package	no	no	yes	yes
non-subclass, other package	no	no	no	yes

Display 7.9 Access Modifiers



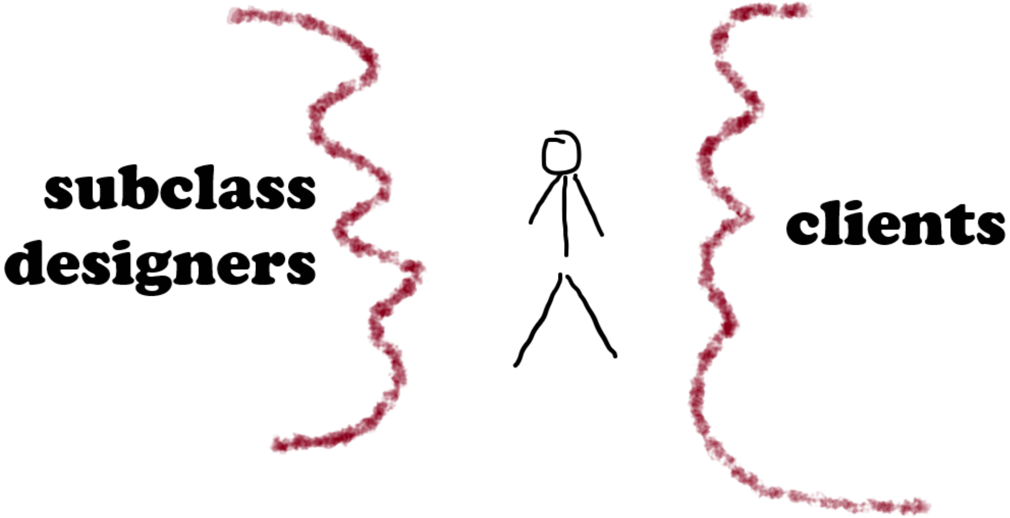
```
package p;
public class A {
  public    int v1;
  protected int v2;
            int v3;
  private   int v4;
}

package p;  public class B           { /* v1, v2, v3, xx */}
package p;  public class C extends A { /* v1, v2, v3, xx */}

package q;  public class D extends A { /* v1, v2, xx, xx */}
package q;  public class E           { /* v1, xx, xx, xx */}
```

Restrictiveness

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

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

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



Class design
Attacked from two sides

`protected` is not really protection.
Design for extension (or don't use extension).

OO Alternatives

1. Interface inheritance
2. Aspect programming

I once attended a Java user group meeting where James Gosling (Java's inventor) was the featured speaker. During the memorable Q&A session, someone asked him: "If you could do Java over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the extends relationship). Interface inheritance (the implements relationship) is preferable. You should avoid implementation inheritance whenever possible.

Allen Holub, JavaWorld, 1 Aug 2003,
`https://www.infoworld.com/article/2073649/why-extends-is-evil.html`

Interface Inheritance

> *The extends keyword is evil; maybe not at the Charles Manson level, but bad enough that it should be shunned whenever possible. The Gang of Four Design Patterns book discusses at length replacing implementation inheritance (extends) with interface inheritance (implements).*

READING: Extends Is Evil

Interface Inheritance

```
// There are many kinds of lists
/*1*/ LinkedList<Integer> list = new LinkedList<> ()
/*2*/ Collection<Integer> list = new LinkedList<> ()

// Works just for linked lists
void g (LinkedList<Integer> list) {
   list.add (/*...*/);
   for (int i: list) /* use 'i' */
}

// With the interface 'Collection',
// the method works for 'ArrayList' as well
void g (Collection <Integer> list) {
   list.add (/*...*/);
   for (int i: list) /* use 'i' */
}
```

Interface Inheritance

```
// There are many kinds of sets
/*1*/ HashSet<Integer> list = new HashSet<> ()
/*2*/ Set<Integer> list = new TreeSet<> ()
/*3*/ Set<Integer> list = new HashSet<> ()
/*4*/ Collection<Integer> list = new HashSet<> ()

// Works just for hash sets
void g (HashSet<Integer> set) {
   list.add (/*...*/);
   for (int i: set) /* use 'i' */
}

// With the interface 'Collection',
// the method works for 'TreeSet' as well
void g (Collection <Integer> set) {
   list.add (/*...*/);
   for (int i: set) /* use 'i' */
}
```

- `Over.java` `toString` overload [skip]
- `Abs.java` abstract classes [skip]
- `Comp.java` composition and interfaces

Is-a versus Has-a

Design consideration.
Favor composition over (implementation) inheritance. Composition over inheritance
It is easy to misuse inheritance.
"is-a" or "has-a".

- `Point.java`
- `SubPoint.java`
- `Aspect.java` Really composition!

Avoid using `protect`.

Use composition instead of inheritance and without inheritance no need for `protect`. It is difficult to protect the base class from misuse by the the subclasses.

Summary

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

Additional, Odd Topics

- clone see `clone.tex`
- functional interfaces see `inter.tex`
- Nested Classes
  1. `Nest.java`
  2. `Local.java`
  3. `Iter.java`
  4. `Anon.java`
  5. `Searcher.java`