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. It is possible 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.
In the source code:

- Class inside the {} of another class. (Usually static)
- Class inside a method (inside a class). Also anonymously.

Also classes can be grouped in directories called packages.

Java 9 modules provides a level of aggregation above packages supporting stronger encapsulation. `java -list-modules`
Package

An ad hoc collection of classes in directory allowing certain, not quite public access to members.
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 startup time and not, like classes, until the application actually tried to use it.
Our main topic here is the organization of classes in the Java subclass hierarchy.
All 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.
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

class A

more general

is a

class B

is a

class C

more specialized

Any number of levels in the hierarchy. And no cycles.
Each class has one superclass; but any number of subclasses can have the same superclass.
Example: Biological Classification

```
Example: Biological Classification

Animalia
  /     /
Insecta Mammalia
 /     /
Rodentia Primates Lepidoptera
    /
Lemuridae Hominidae
```

Kingdom
Class
Order
Family
Hierarchical Organization

- person
  - staff
  - student
    - undergrad
    - graduate
  - faculty
Hierarchical Organization

Indo-European

Indo-Iranian
- Indic
  - Hindi
  - Bengali
- Iranian
  - Persian
  - Pasto

Italic
- Spanish
- French

Balto-Slavic
- Slavic
  - Russian
Example: Hierarchical Organization from Java API

```
JComponent
   ├── AbstractButton
   │     ├── JButton
   │     └── JLabel
       └── JTextComponent
               ├── JTextArea
               └── JTextField
```
Example: Hierarchical Organization

Object

Number
- BigDecimal
- Integer

Faculty
- Float
Hierarchical Organization

Point

\[
\begin{array}{c}
\text{Rectangle} \\
\text{Circle}
\end{array}
\]
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.
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`. 
Hierarchical Organization

```java
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 {  // ...
```
// 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.
(Java interfaces – discussed later – can be used to create a kind of multiple superclasses.)
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. In object-oriented programming we eventually learn the idioms or design patterns to solve different problems using this organization. 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?
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 means 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);
    public int hashCode (); // encoding as integer
    protected Object clone (); // copy
    public Class<? extends Object> getClass (); // meta information
    public void notify (); // synchronization of threads
    public void wait (); // synchronization of threads
}

For example, assignment

```java
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.
Subclass Polymorphism

The flexibility only works one way.

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

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

```java
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*. 
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`. 
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 $\text{int } \text{getX}()$, then the subclass must also have method $\text{int } \text{getX}()$.

Therefore: the subclass inherits all the member methods and fields of the superclass.
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) {
      System.out.println(d.bark());
    }
  }
}
```
Extend

Since,

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<? extends Object> getClass ();  // meta information
public void notify ();  // synchronization
public void wait ();  // synchronization
Inheriting the `toString()` method

*Every* object has a `toString()` method!

```java
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].toString());
        }
    }
}
```

By the way, the output is not very specific:

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

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

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);
        System.out.println (c.x +"", "+ c.y +", "+ c.radius);
    }
}
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. This is allowed because the author of the subclass should not have to know what names the author of the superclass might have picked. To forbid the class would enable the subclass author to “peek” inside the superclass.
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 (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 is implicitly declared.”

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

is equivalent to the declaration

```java
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.”
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.

```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){super(x);this.y=y;}
    @java.lang.Override
    int sum () { return x+y; }
}
```
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 only 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.
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. For example,

```java
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.”
Overriding Example

Point2D.java
Overriding Example

Advice.java
Animals.java
Overriding Example

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

class SubA extends SuperA {
    @java.lang.Override
    public int getVal() { return 1; }
}

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

class C extends B {
    @java.lang.Override
    public SubA foo() {

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: Dispatch.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"); }
}
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 different
- `AccessMethod.java` — casts make no difference
Object-Oriented Design

1. Identify the problem’s objects.
   1. If the object cannot be directly represented using the existing types, then design a class to do so.
   2. If two or more classes share common attributes, then design a hierarchy to store their common attributes.

2. Identify the problem’s operations.
   1. Define a method to do the operations.
   2. Structure the method within the class hierarchy so as to take advantage of inheritance.
   3. Where necessary, have subclasses override inherited definitions.

3. Solve the problem.
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.

```java
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:

Points.java  Widening.java
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:

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

Runtime system:

```java
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), It is used as superclass to define a subclass with the responsibility of implementing the missing behavior.
In a class hierarchy, if a method’s behavior depends on 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 the 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.
Compare the following:

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

abstract class Object {
    abstract public String toString ();
}
class SubClass extends Object {
    int x;
    public String toString () { return Integer.toString(x); }
}
```
abstract class AbstractList implements List {
    // 1. Operations that are in common to all lists
    // 2. Operations that have different behavior, but common interface
}
Main2.java – abstract class example
InheritV.java – visitor design pattern
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

Verbose.java
List.java
Example.java
PointPack.java
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.
Table 2-1. Important functional interfaces in Java

<table>
<thead>
<tr>
<th>Interface name</th>
<th>Arguments</th>
<th>Returns</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate&lt;T&gt;</td>
<td>T</td>
<td>boolean</td>
<td>Has this album been released yet?</td>
</tr>
<tr>
<td>Consumer&lt;T&gt;</td>
<td>T</td>
<td>void</td>
<td>Printing out a value</td>
</tr>
<tr>
<td>Function&lt;T,R&gt;</td>
<td>T</td>
<td>R</td>
<td>Get the name from an Artist object</td>
</tr>
<tr>
<td>Supplier&lt;T&gt;</td>
<td>None</td>
<td>T</td>
<td>A factory method</td>
</tr>
<tr>
<td>UnaryOperator&lt;T&gt;</td>
<td>T</td>
<td>T</td>
<td>Logical not (!)</td>
</tr>
<tr>
<td>BinaryOperator&lt;T&gt;</td>
<td>(T, T)</td>
<td>T</td>
<td>Multiplying two numbers (*)</td>
</tr>
</tbody>
</table>
Nested Classes

Nest.java
Local.java
Iter.java
Anon.java
Searcher.java
Equals

Main.java what?
Override.java
Example.java
Why clone? Because assignment just copies references. This creates aliases and this leads to programming mistakes.

```java
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.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 Control** in Java language specification.

<table>
<thead>
<tr>
<th>access from</th>
<th>private</th>
<th>“package”</th>
<th>protected</th>
<th>public</th>
</tr>
</thead>
<tbody>
<tr>
<td>same class</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>in subclass, same package</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>non-subclass, same package</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>in subclass, other package</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>non-subclass, other package</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Display 7.9 Access Modifiers

```java
package somePackage;

public class A {
    public int v1;
    protected int v2;
    int v3; //package
    //access
    private int v4;
}

public class B {
    //can access v1.
    can access v2.
    //can access v3.
    cannot access v4.
}

public class C extends A {
    //can access v1.
    can access v2.
    can access v3.
    cannot access v4.
}

public class D extends A {
    //can access v1.
    can access v2.
    cannot access v3.
    cannot access v4.
}

public class E {
    //can access v1.
    cannot access v2.
    cannot access v3.
    cannot access v4.
}
```

In this diagram, "access" means access directly, that is, access by name.

A line from one class to another means the lower class is a derived class of the higher class.

If the instance variables are replaced by methods, the same access rules apply.
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).

private < “package” < protected < public

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;
   }
}
subclass designers

clients

Class design
Attacked from two sides
protected is not really protection.
Design for extension (or don’t use extension).
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.

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
// 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

Note the special java syntax that takes advantage of the interface Iterable.
// There are many kinds of sets
/*1*/ HashSet<Integer> list = new HashSet<>();
/*2*/ Collection<Integer> list = new HashSet<>();

// Works just of 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 (aspect programming)
Is-a versus Has-a

Design consideration.
Favor composition over (implementation) inheritance.
It is easy to misuse inheritance.
“is-a” or “has-a”.
Aspect programming

- `Point.java`
- `SubPoint.java`
- `Aspect.java`
Summary

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