Reusing Classes

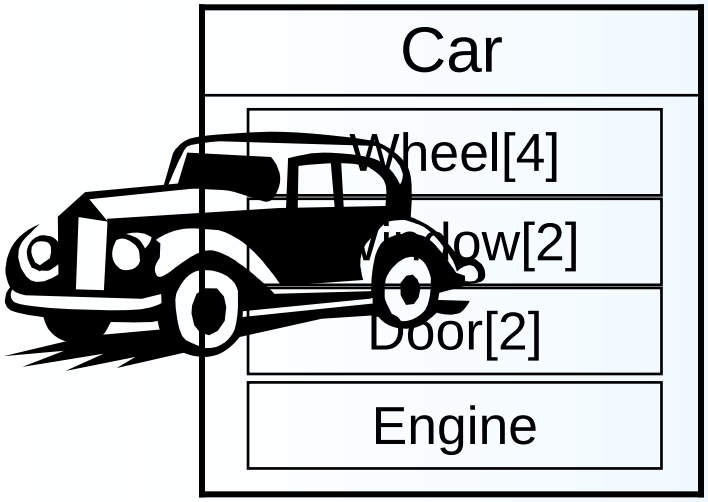
Qiuyan Huo
Software Engineering Institute
qyhuo@mail.xidian.edu.cn

Reusing Classes

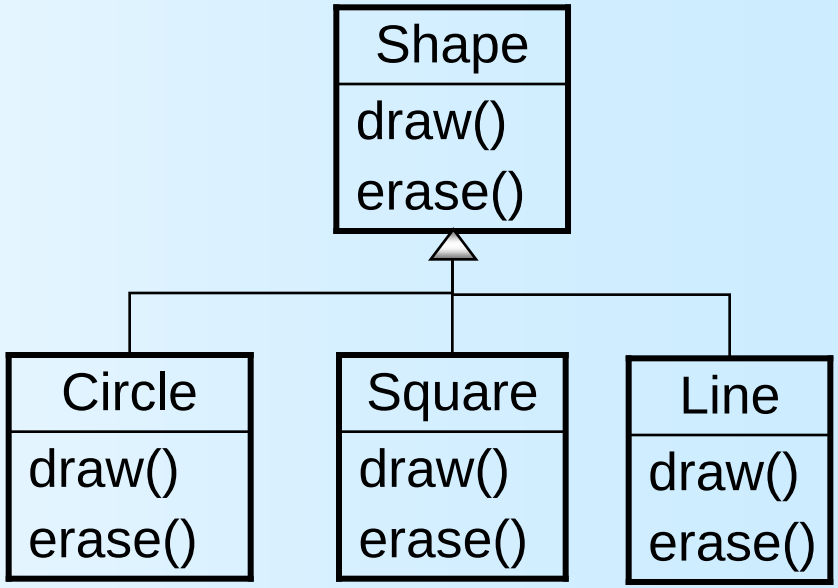
When you need a class, you can:

1) Get the perfect one off the shelf (one extreme)
2) Write it completely from scratch (the other extreme)
3) Reuse an existing class with composition
4) Reuse an existing class or class framework with inheritance

Composition vs. Inheritance

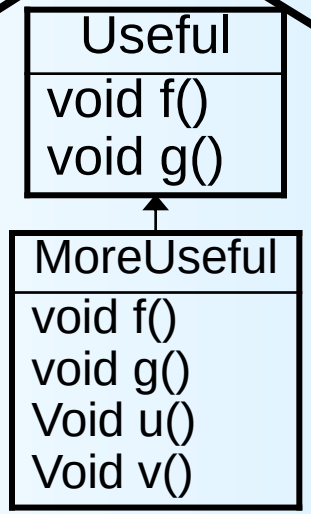


Pure composition
Code reuse



Pure inheritance

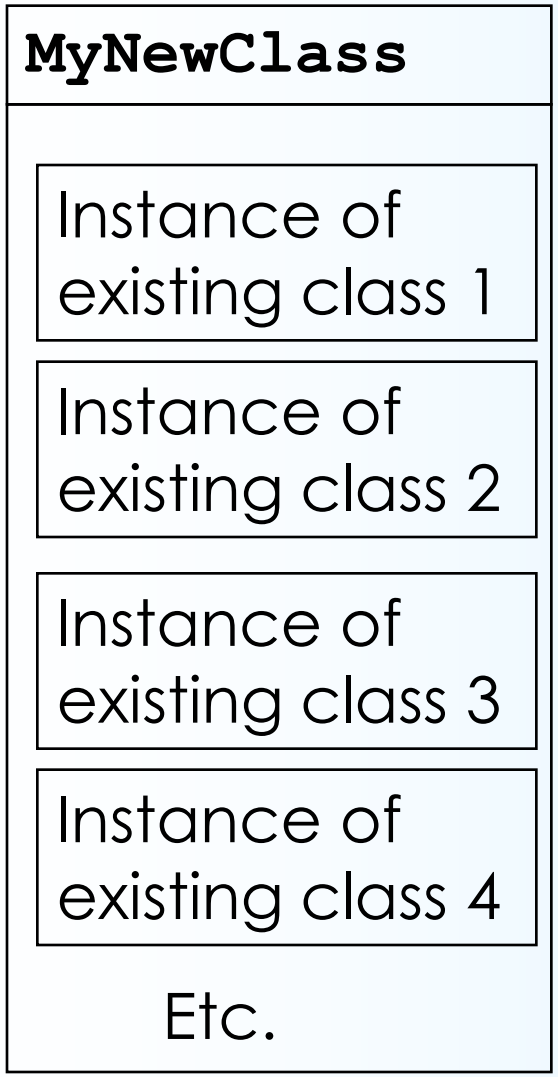
Interface duplication for interchangeable objects



Extension

Inherit to extend the interface

Composition Syntax




```
class MyNewClass
{
        Foo x = new Foo();
        Bar y = new Bar();
        Baz z = new Baz();
        //...
}
```

- Composition: make new objects by combining other objects
- Can also initialize in the constructor
- Flexibility: Can change objects at run time!
- "Has-A" relationship

Composition Syntax – cont.

```
class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}
```

Composition Syntax – cont.

```
public class Bath {
  private String
    // Initializing at point of definition:
    s1 = new String("Happy"),
    s2 = "Happy",
    s3, s4;
  private Soap castile;
  private int i;
  private float toy;
  public Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castile = new Soap();
  }
```

they'll always be initialized before the constructor is called

Initialized In the constructor for that class

Composition Syntax – cont.

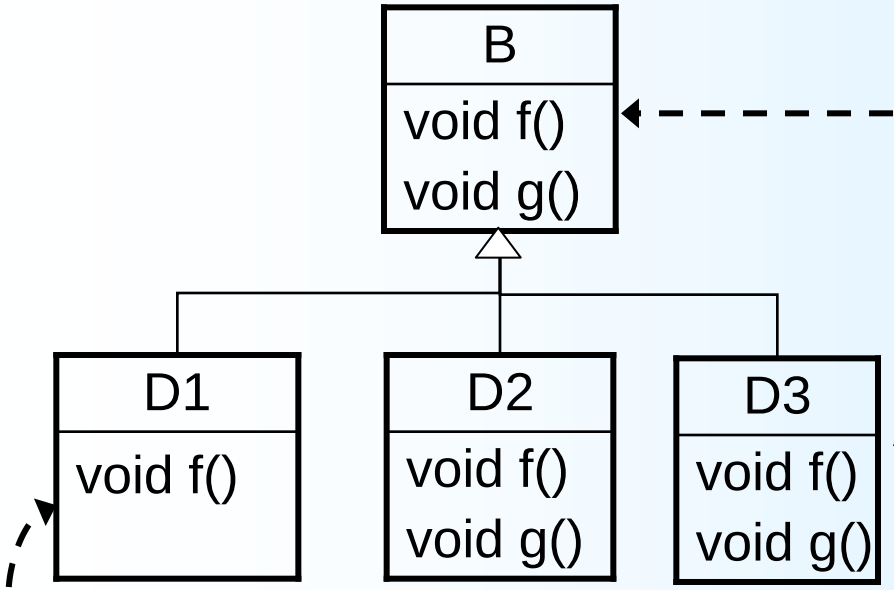
```
public String toString() {
      // Delayed initialization:
      if ( s4 == null)
            s4 = new String("Joy");
      return
"s1 = " + s1 + "\n" +
"s2 = " + s2 + "\n" +
"s3 = " + s3 + "\n" +
"s4 = " + s4 + "\n" +
"i = " + i + "\n" +
"toy = " + toy + "\n" +
"castile = " + castile;
}
public static void main(String[] args) {
      Bath b = new Bath();
      System.out.println(b);
}
}
```

Right before you actually need to use the object. This is often called **lazy** initialization.

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

return

Inheritance



Base interface automatically duplicated in derived classes

Base data members also duplicated

If a derived member is not redefined, base definition is used

继承是子类利用父类中定义的方法和变量就像它们属于子类本身一样。
方法的覆盖：在子类中重新定义父类中已有的方法。

Inheritance syntax

```
class B {
        public void f() {/*...*/}
        public void g() {/*...*/}
}


class D1 extends B {
        public void f() {/*...*/}
}
```

通过在类的声明中加入**extends**子句来创建一个类的子类.

如果缺省**extends**子句，则该类为**java.lang.Object**的子类。

Inheritance syntax – cont.

```
class Cleanser {
  private String s = new String("Cleanser");
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public String toString() { return s; }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    System.out.println(x);
  }
}
```

子类可以继承父类中访问权限设定为**public**、**protected**、 **default**的成员变量和方法。但是不能继承访问权限为**private**的成员变量和方法。

Inheritance syntax – cont.

```
public class Detergent extends Cleanser {
  // Change a method:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    System.out.println(x);
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
}
```

方法覆盖(overriding)指子类对父类的方法进行改写。

子类覆盖的方法同父类的方法要保持名称、返回值类型、参数列表的统一。

- 改写后的方法不能比被覆盖的方法有更严格的访问权限
- 改写后的方法不能比被覆盖的方法产生更多的异常

```
Cleanser dilute() apply() Detergent.scrub() scrub() foam()
Testing base class:
Cleanser dilute() apply() scrub()
```

Initializing the base class

- Java ***automatically*** calls the base-class' default constructor in the derived-class constructor

无super和this，编译器调用super()

```
class Art {
  Art() {
    System.out.println("Art constructor");
  }
}
class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}
public class Cartoon extends Drawing {
  public Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
} ///:~
```

Output:

```
Art constructor
Drawing constructor
Cartoon constructor
```

Constructors with arguments

- Base constructor call must happen first
- Use **super** keyword

通过super实现对父类成员的访问：

- 用来访问父类被隐藏的成员变量，如：**super.variable**
- 用来调用父类中被重写的方法，如：**super.Method([paramlist]);**

```
class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}
```

调用父类的构造方法，如：

super([paramlist]);

Top 10 Mistakes Java Developers Make: #9

```
class Super {
    String s;

    public Super(String s) {
        this.s = s;
    }
}

public class Sub extends Super {
    int x = 200;
    public Sub(String s) {

    }

    public Sub(){
        System.out.println("Sub");
    }

    public static void main(String[] args){
        Sub s = new Sub();
    }
}
```

- Constructor of Super and Sub
  - 1) add a **`Super()`** constructor to the Super class
  - 2) remove the self-defined **`Super`** constructor
  - 3) add **`super(value)`** to **`sub`** constructors

Delegation → forwarding methods

- *Not directly supported by Java*
- Midway between inheritance and composition
  - Place a member object in the class be building☺
  - Expose all the members from the member object in the new class☻

```
public class SpaceShipControls {
  void up(int velocity) {}
  void down(int velocity) {}
  void left(int velocity) {}
  void right(int velocity) {}
  void forward(int velocity) {}
  void back(int velocity) {}
  void turboBoost() {}
}
```

```
public class SpaceShipDelegation {
  private String name;
  private SpaceShipControls controls =
    new SpaceShipControls(); ☺
  public SpaceShipDelegation(String name) {
    this.name = name;
  }
  // Delegated methods:☻
  public void back(int velocity) {
    controls.back(velocity);
  }
  public void down(int velocity) {
    controls.down(velocity);
  }
  public void forward(int velocity) {
    controls.forward(velocity);
  }
```

```
  public void left(int velocity) {
    controls.left(velocity);
  }
  public void right(int velocity) {
    controls.right(velocity);
  }
  public void turboBoost() {
    controls.turboBoost();
  }
  public void up(int velocity) {
    controls.up(velocity);
  }
  public static void main(String[] args) {
    SpaceShipDelegation protector =
      new SpaceShipDelegation("NSEA Protector");
    protector.forward(100);
  }
}
```

Combining composition & inheritance

```
class Plate {
  Plate(int i) {
    System.out.println("Plate constructor");
  }
}

class DinnerPlate extends Plate {
  DinnerPlate(int i) {
    super(i);
    System.out.println("DinnerPlate constructor");
  }
}

class Utensil {
  Utensil(int i) {
    System.out.println("Utensil constructor");
  }
}
```

Combining composition & inheritance - cont.

```
class Spoon extends Utensil {
  Spoon(int i) {
    super(i);
    System.out.println("Spoon constructor");
  }
}

class Fork extends Utensil {
  Fork(int i) {
    super(i);
    System.out.println("Fork constructor");
  }
}

class Knife extends Utensil {
  Knife(int i) {
    super(i);
    System.out.println("Knife constructor");
  }
}
```

Combining composition & inheritance - cont.

```
// A cultural way of doing something:
class Custom {
  Custom(int i) {
    System.out.println("Custom constructor");
  }
}
public class PlaceSetting extends Custom {
  private Spoon sp;
  private Fork frk;
  private Knife kn;
  private DinnerPlate pl;
  public PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    System.out.println("PlaceSetting constructor");
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
  }
}
```

```
Custom constructor
Utensil constructor
Spoon constructor
Utensil constructor
Fork constructor
Utensil constructor
Knife constructor
Plate constructor
DinnerPlate constructor
PlaceSetting constructor
```

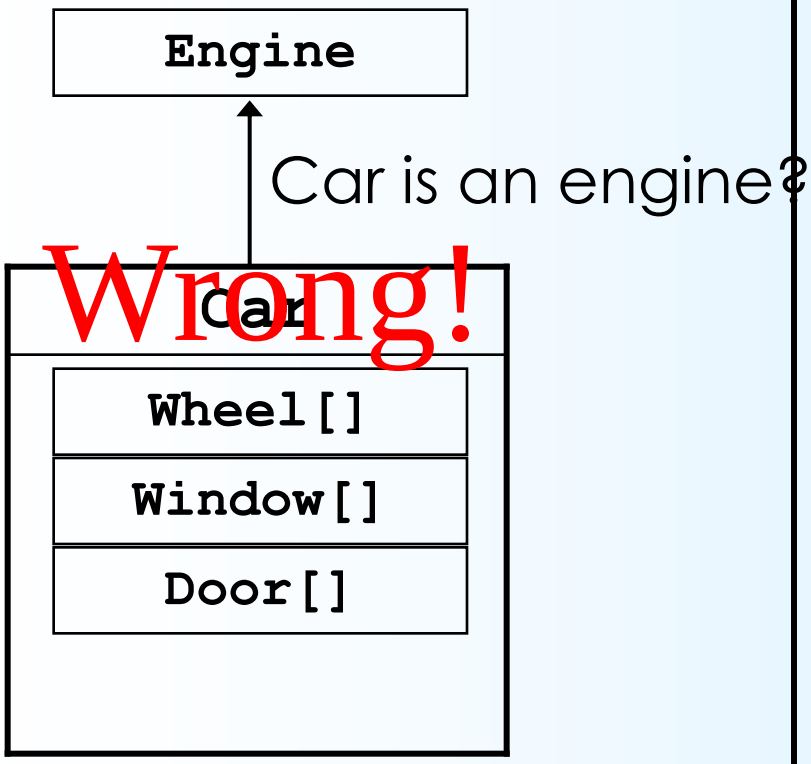
Name hiding

```
class Homer {
  char doh(char c) {
    System.out.println("doh(char)");
    return 'd';
  }
  float doh(float f) {
    System.out.println("doh(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void doh(Milhouse m) {
    System.out.println("doh(Milhouse)");
  }
}

public class Hide {
  public static void main(String[] args) {
    Bart b = new Bart();
    b.doh(1);
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhouse());
  }
}
```

If a Java base class has a method name that's overloaded several times, redefining that method name in the derived class will ***not*** hide any of the base-class versions

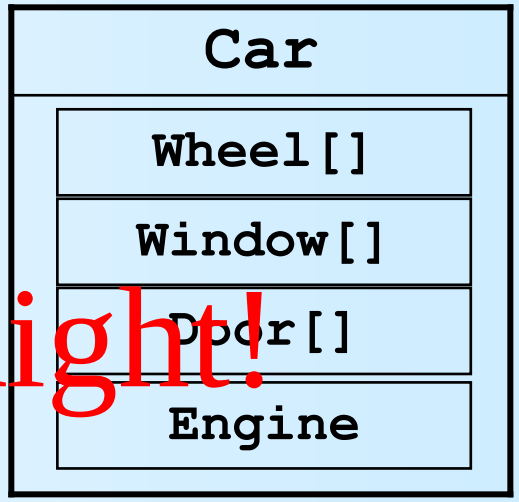
```
doh(float)
doh(char)
doh(float)
doh(Milhouse)
```

Choosing composition vs. inheritance



- You start an Engine and you start a Car, so…?

“B是一个A吗?”



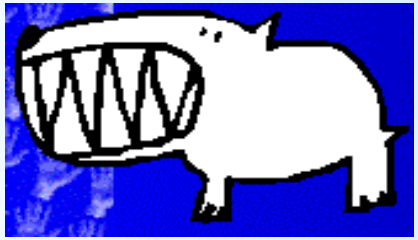
- Inheritance is determined at compile time; member binding can be delayed until run time
- Member rebinding is possible
- In general, prefer composition to inheritance as a first choice

```
class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
}

class Wheel {
  public void inflate(int psi) {}
}

class Window {
  public void rollup() {}
  public void rolldown() {}
}

class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}
```

"Simple Aggregation"

```
public class Car {
  public Engine engine = new Engine();
  public Wheel[] wheel = new Wheel[4];
  public Door
    left = new Door(),
    right = new Door(); // 2-door
  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
}
```

protected



- Inheritors (and the package) can access protected members

  BUT

- They are then vulnerable to changes in the base-class implementation
- Class users are prevented from accessing protected members

Protected Members Example

```
class Villain {
  private int i;
  protected int read() {return i;}
  protected void set(int ii) {i = ii;}
  public Villain(int ii) {i = ii;}
  public int value(int m) {return m*i;}
}

public class Orc extends Villain {
  private int j;
  public Orc(int jj) {super(jj); j = jj;}
  public void change(int x){set(x);}
};
```

Incremental development

- inheritance is that it supports ***incremental development***
- how cleanly the classes are separated
- program development is an incremental process
- Remember: inheritance is meant to express a relationship that says: **"This new class is a *type of* that old class."**

take a new look at your class hierarchy with an eye to collapsing it into a sensible structure.

Upcasting

"The new class *is a type of* the existing class."

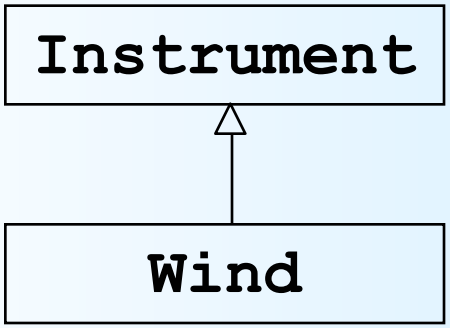
- How compiler support

```
class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
}
```

we can accurately say that a **Wind** object is also a type of **Instrument**.

Why "upcasting"?

- the way class inheritance diagrams have traditionally been drawn: with the root at the top of the page, growing downward.



- Casting from a derived type to a base type moves *up* on the inheritance diagram, so it's commonly referred to as *upcasting*.

The `final` Keyword

- "This cannot be changed"
- A bit confusing: two reasons for using it
  - Design
  - Efficiency

- final data
  - 修饰变量，变量就变成了常量；
- final methods
  - 修饰方法，方法就不能再覆盖；
- final class
  - 修饰类，类就不能再继承。

`final` data

- Compile-time constant
  - Must be given a value at point of definition
    ```
    final static int NINE = 9;
    ```
  - May be "folded" into a calculation by the compiler
- Run-time constant
  - Cannot be changed from initialization value
    ```
    final int RNUM = (int)(Math.random()*20);
    ```
  - **`final static`**: only one instance per class, initialized at the time the class is loaded, cannot be changed.
  - final references: cannot be re-bound to other objects

Blank `finals`

- fields are declared as **`final`** but **not** given an initialization value
- blank final ***must*** be initialized before it is used

```
class Poppet {
  private int i;
  Poppet(int ii) { i = ii; }
}
```

Blank **finals** example

```
public class BlankFinal {
  private final int i = 0; // Initialized final
  private final int j; // Blank final
  private final Poppet p; // Blank final reference
  // Blank finals MUST be initialized in the constructor:
  public BlankFinal() {
    j = 1; // Initialize blank final
    p = new Poppet(1); // Initialize blank final reference
  }
  public BlankFinal(int x) {
    j = x; // Initialize blank final
    p = new Poppet(x); // Initialize blank final reference
  }
  public static void main(String[] args) {
    new BlankFinal();
    new BlankFinal(47);
  }
}
```

`final` arguments

```
void f(final int i) { // ...
```

- Primitives can't be changed inside method

```
void g(final Bob b) {
    b = new Bob();
}
```

- References can't be rebound inside method
- Generally, neither one is used

`final` Methods

- Put a "lock" on a method to prevent any inheriting class from overriding it (design)
- Efficiency (try to avoid the temptation…)
  - Compiler has permission to "inline" a final method
    - Replace method call with code
    - Eliminate method-call overhead
  - Programmers are characteristically bad about guessing where performance problems are
  - Limits use of class (e.g., can't override Vector)
- **`private`** methods are implicitly **`final`**

`final` Classes

- Cannot inherit from **`final`** class
- All methods are implicitly **`final`**
- Fields may be **`final`** or not, as you choose

Initialization & class Loading

- A Java program is a collection of **`.class`** files, each containing a single **`Class`** object
- Class code is loaded at point of first use
  - Typically, first time you create an object of that type
  - **`static`**s are initialized upon loading, in textual order

Initialization with Inheritance

```
class Insect {
      int i = 9;
      int j;
      Insect() {
            print("i = " + i + ", j = " + j);
            j = 39;
      }
      static int x1 =
       print("static Insect.x1 initialized");
      static int print(String s) {
            System.out.println(s);
            return 47;
      }
}
```

Initialization Example

```
public class Beetle extends Insect {
  private int k = print("Beetle.k initialized");
  public Beetle() {
    System.out.println("k = " + k);
    System.out.println("j = " + j);
  }
  private static int x2 =
    print("static Beetle.x2 initialized");
  static int print(String s) {
            System.out.println(s);
            return 47;
  }
  public static void main(Stri
    System.out.println("Beetle
    Beetle b = new Beetle();
  }
}
```

```
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
```

Summary of Reusing Classes

- Easy to think that **OOP** is *only* about inheritance
- Often easier and more flexible to start with composition. Remember to say "**has-a**" and "**is-a**"
- Use inheritance when it's clear that a new type **is a kind of** a base type
- … and especially when you discover a need for polymorphism (next chapter)

Exercise

- Design **`BankSystem`** class. You need to design more than one class. For example:
  - You need to design a class **`BankAccount`** to model users' bank accounts. Probably different bank accounts (**`CashAccount`**, **`CreditAccount,`** …). The account should keep a user's name and balance, accurate to the nearest cent…
  - The user should be able to make deposits and withdrawals on his/her account, as well as changing the account's name at any time.
  - Also, the system needs to be able to find out how many **`BankAccount`**s have been created in total.
  - For each account, only the last 6 **`Transaction`**s should be able to store in ascending order and be printed.