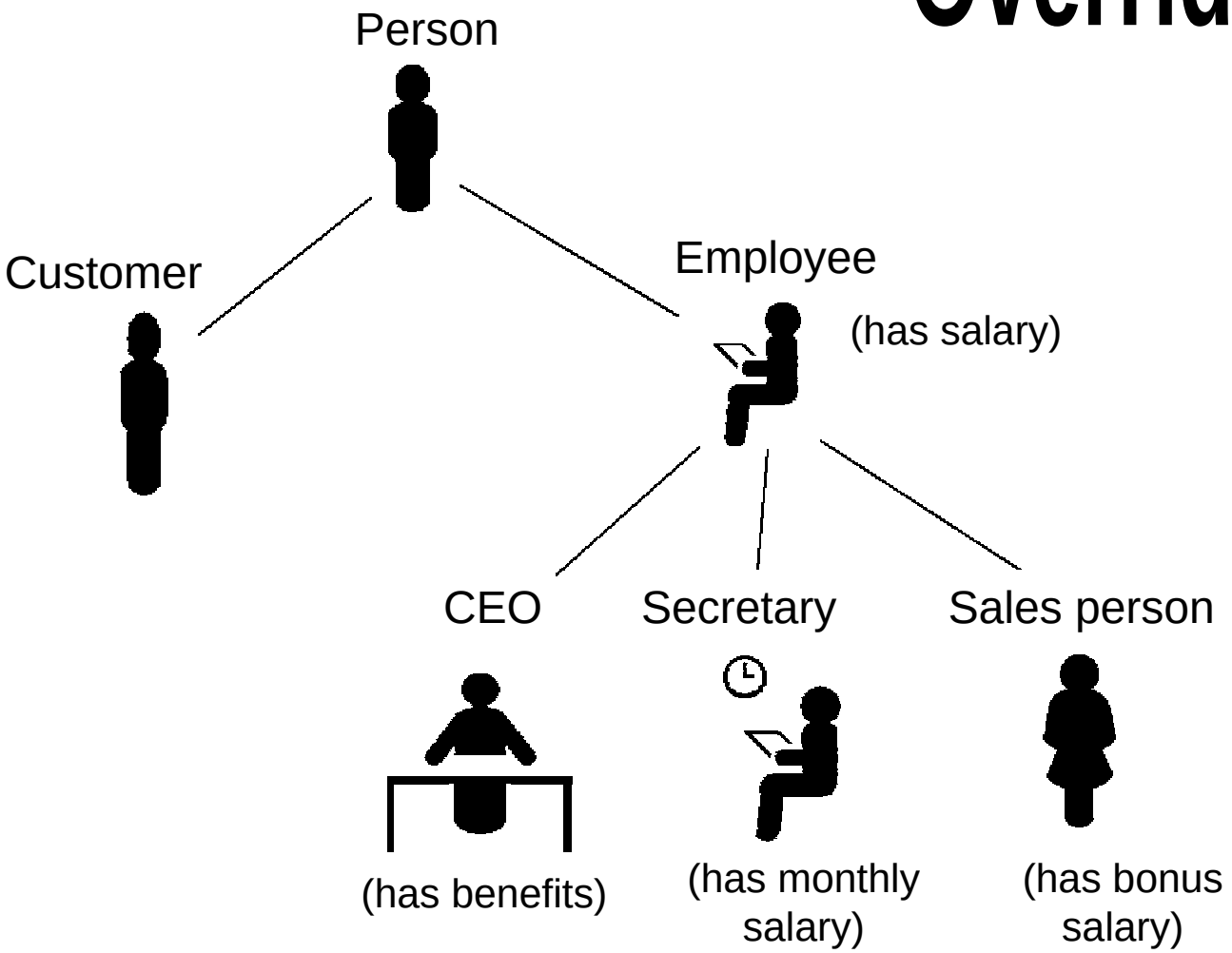
Type hierarchy in Java

- Superclass can have
  - Full implementation
  - Partial implementation (abstract class)
  - No implementation (interface)
- A subclass can
  - re-implement (override) superclass methods
  - provide new, additional methods
- A subclass can have access to the superclass data
  - through public methods (API)
  - through protected methods
  - directly (non-private attributes)
- All classes inherit from **`Object`**

Overriding, ex



Overriding, ex

```
class Employee extends Person{
   private int salary;
   public int getSalary() {
       return salary;
   }
}

class SalesPerson extends Employee{
   public int getSalary() { // overridden method
       return super.getSalary()+computeBonus();
   }
   private int computeBonus() { … }
}
```


Overriding, ex

```
class SalesPerson extends Employee{
   public int getSalary() { // overridden method
       return super.getSalary()+computeBonus();
   }
   private int computeBonus() { … }
}

class TravellingSalesPerson extends SalesPerson {
   public int getSalary() { // overridden method
       return super.getSalary()+
                           computeMileageAllowance();
   }
   private int computeMileageAllowance() { … }
}
```

– Can we reach `getSalary()` in `Employee` from `TravellingSalesPerson` ?

Overriding, ex

```
Employee[] personnel; // heterogeneous collection
// fill array with Employee objects
.
.
int totalSalaryCost = 0;
for (int i=0;i<personnel.length;i++) {
      totalSalaryCost += personnel.getSalary();
}
```

How does the object know what type it is, and thus which method to invoke?

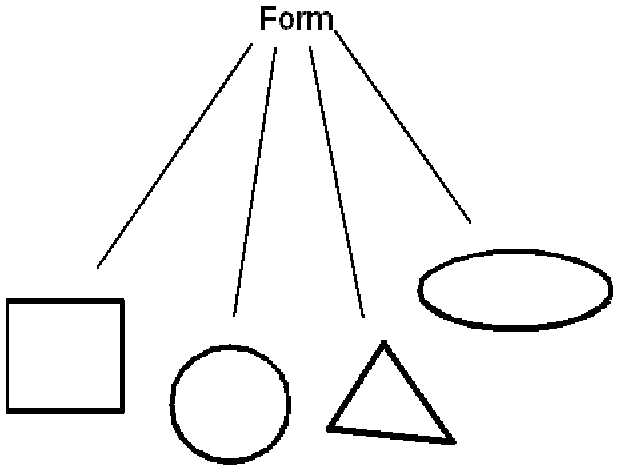
Virtual method invocation

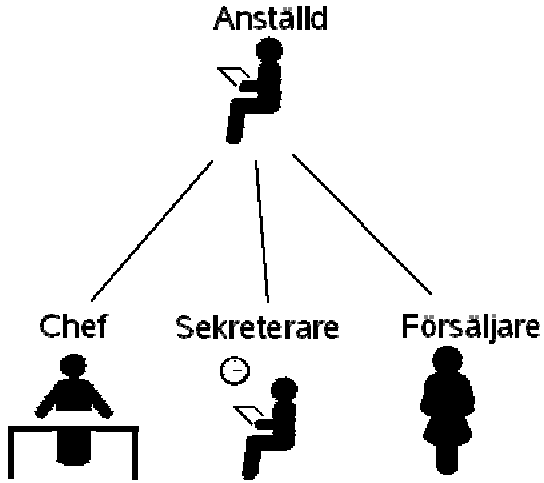
- It is the method of the runtime type that is used (and this is not necessarily the same as the declared type)
- Compiler only sees apparent type
- Runtime uses actual type (subtype of apparent type)
- (Also referred to as “dispatching”)

Polymorphism

poly = “many”

morph = “form”





Polymorphism, ex

```
class Pet{
    public void sound(){
      System.out.println(””);
    }
}

class Dog extends Pet{
    public void sound(){ //overridden method
      System.out.println("Vov");
    }
}

class Cat extends Pet{
    public void sound(){ //overridden method
      System.out.println("Mjau");
    }
}
```


Package

- Packade is given first in source file

```
package brokersystem;
```

- Subpackages can be used

```
package brokersystem.gui;
```

- If no package is defined, the class will belong to the no-name, default package
- Use packages! (except for small test programs)
- (“java” and “javax” are reserved)

Package, ex

```
package brokersystem;

import java.util.ArrayList;
import java.io.*;

public class Broker{
        .
        .
        .
}
```

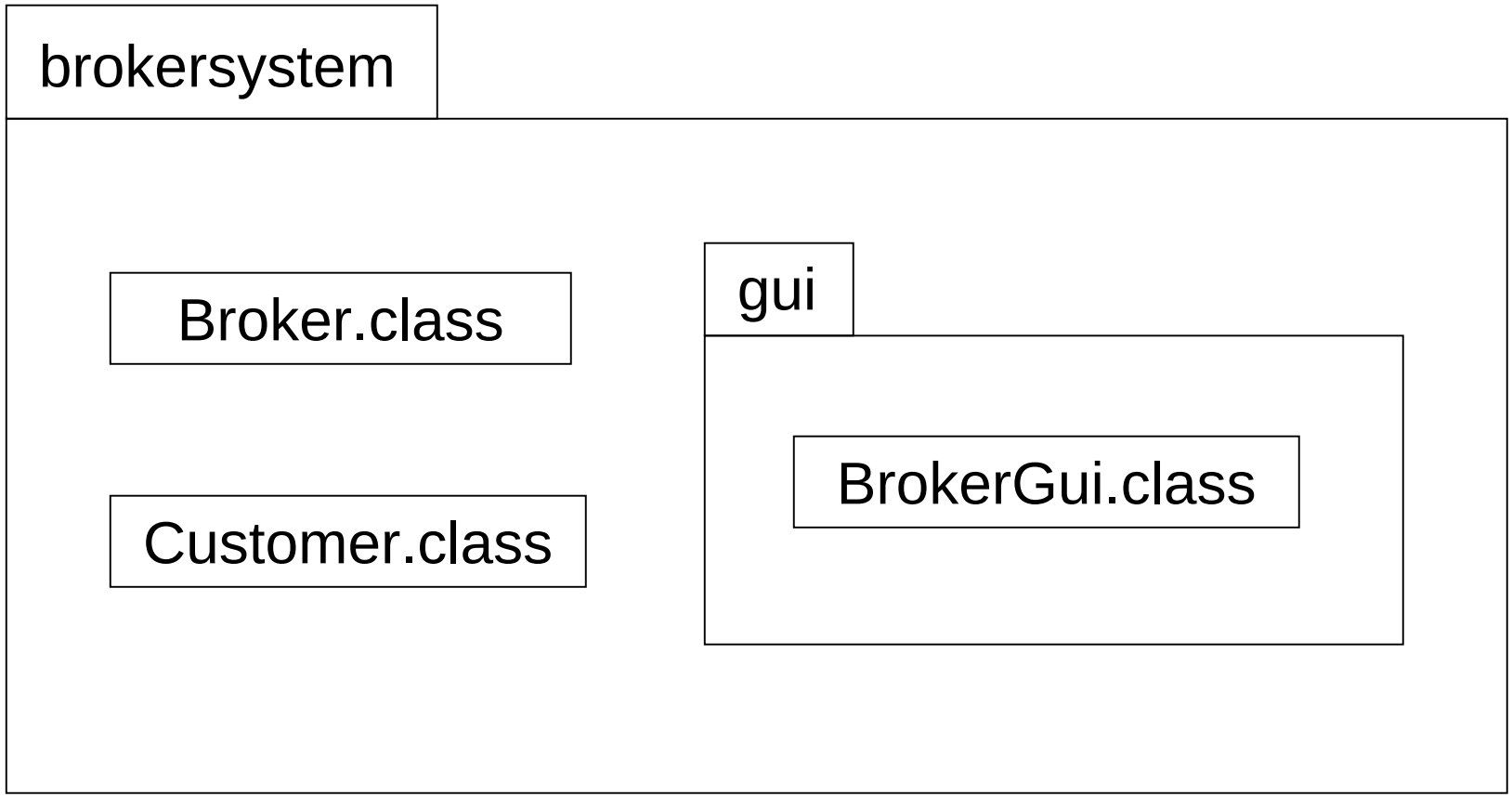
```
package brokersystem.gui;

import java.awt.*;
import java.awt.event.*;

public class BrokerGui[
        .
        .
        .
}
```


Package structure, ex

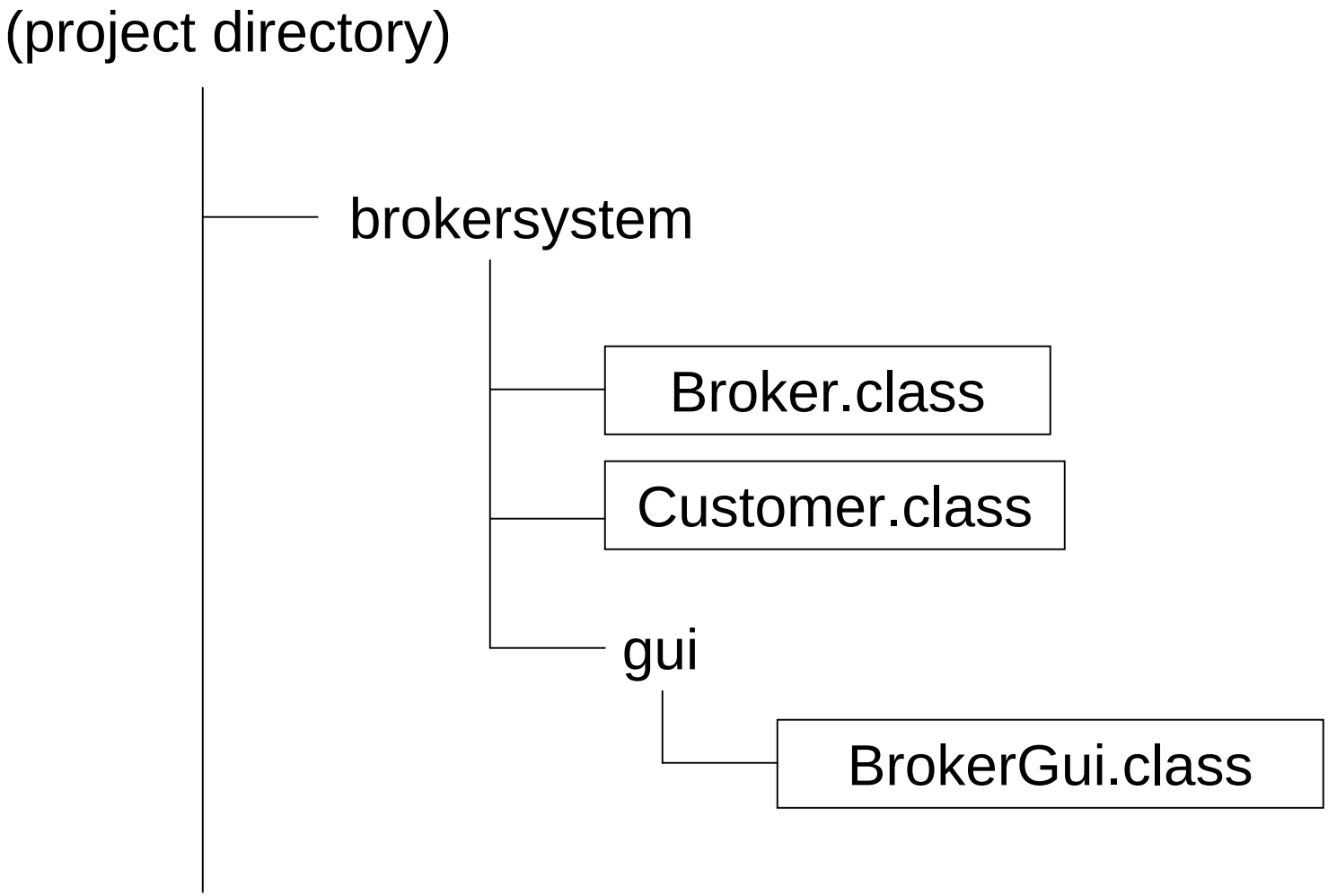
```
>javac –d . *.java
```

or:

```
>javac –d . Broker.java
```



Package, compiling



Why packages?

- Encapsulation – a higher level of encapsulation; only **`public`** classes and interfaces can be used outside a package

- Naming – a “fully qualified” name = package name + class name (avoids name conflicts; e.g. **`Date`** in **`java.util`** and in **`java.sql`**).

- Organization

Access modifiers

Visibility	Object of same type	Object in same package	Children in other packages	All
`private`	**yes**	**no**	**no**	**no**
default	**yes**	**yes**	**no**	**no**
`protected`	**yes**	**yes**	**yes**	**no**
`public`	**yes**	**yes**	**yes**	**yes**

Bird, ex

Using mutator:

```
class Bird{
  String name;
  setName(String s){
     name = s;
  }
}
```

Using constructor:

```
class Bird{
  String name;
  Bird(String s){
     name = s;
  }
}
```


Constructors

- Same name as class
- No return type
- Can be overloaded
- If no constructor is defined, a no-argument default constructor is used
- No constructor is same as:

```
public MyClass() {}
```

…which is same as:

```
public MyClass() { super(); }
```


Constructors

- Constructors are not inherited!
- Constructor of superclass is invoked before that of subclass (first **`Car`**, then **`SportsCar`**)
- **`this`** and **`super`** must be used on first line of constructor
- (any non-private variables and methods of the superclass can be accessed using **`super`**)

this, ex

```
public class Bird{

    private String name;

    Bird(String name){
      this.name = name;
    }

    Bird(){
      this(“Gråsparv");
    }

    public static void main (String []args){
      new Bird(“Trana");
      new Bird();
    }
}
```


super, ex

```
class Car{
   int doors;
   Car(int doors){
       this.doors = doors;
   }
   Car(){
       this(4);
   }
   public void equipment(){
       System.out.println(”Number of doors: ” + doors);
   }
}

class SportsCar extends Car{
   boolean turbo;
   SportsCar(boolean turbo){
       super(2);
       this.turbo = turbo;
   }
   public void equipment(){
       super.equipment();
       if (turbo) System.out.println(”Has turbo”);
   }

}
```


Pet again...

Would we ever want to instantiate this class?

```
class Pet{
   public void sound(){
      System.out.println(””); // Typical sound?
   }
}
```

Better to make it abstract:

```
abstract class Pet{
   public abstract void sound();
}
```


Abstract classes

- Partially implemented classes
- Has one or more “abstract” methods – i.e. methods that are not implemented
- Can have instance variables
- Cannot be instantiated – only used as superclasses
- Abstract methods implemented in subclasses
- Implemented methods can call abstract methods (example of the Template pattern”; e.g. AbstractBankListener)

Abstract classes, ex

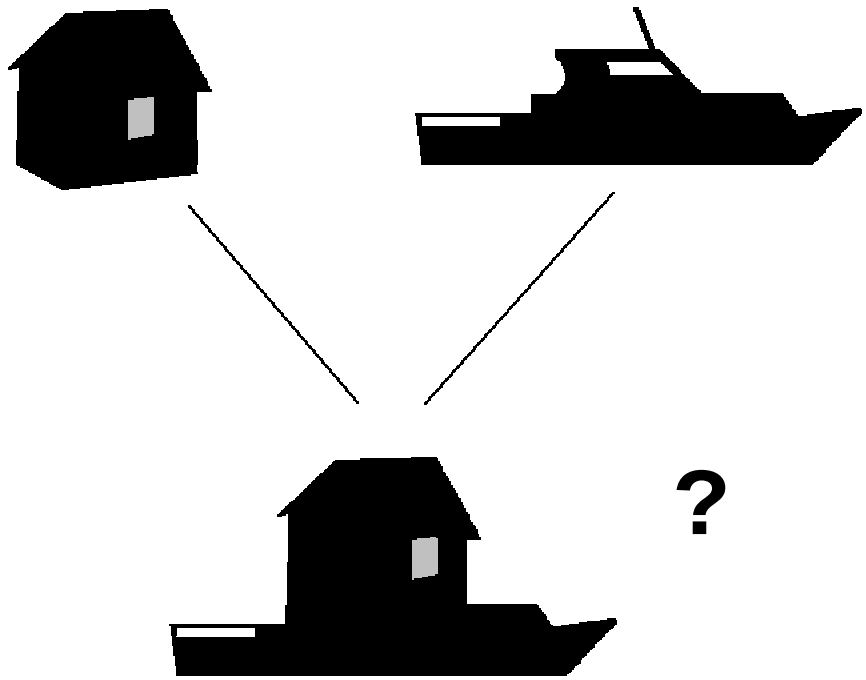
```
abstract class Vehicle{

    public abstract void sound();

    public void testSound(){
        this.sound();
    }
}
```


Abstract classes, ex

```
abstract class Vehicle{
    public abstract void sound();
    public void testSound(){
       this.sound();
    }
}

class Car extends Vehicle{
    public void sound(){
       System.out.println("brum, brum");
    }
}

class Drive{
    public static void main(String [] args){
       Car lada  = new Car();
       lada.testSound();
    }
}
```


Multiple inheritance

Conceptual problems…



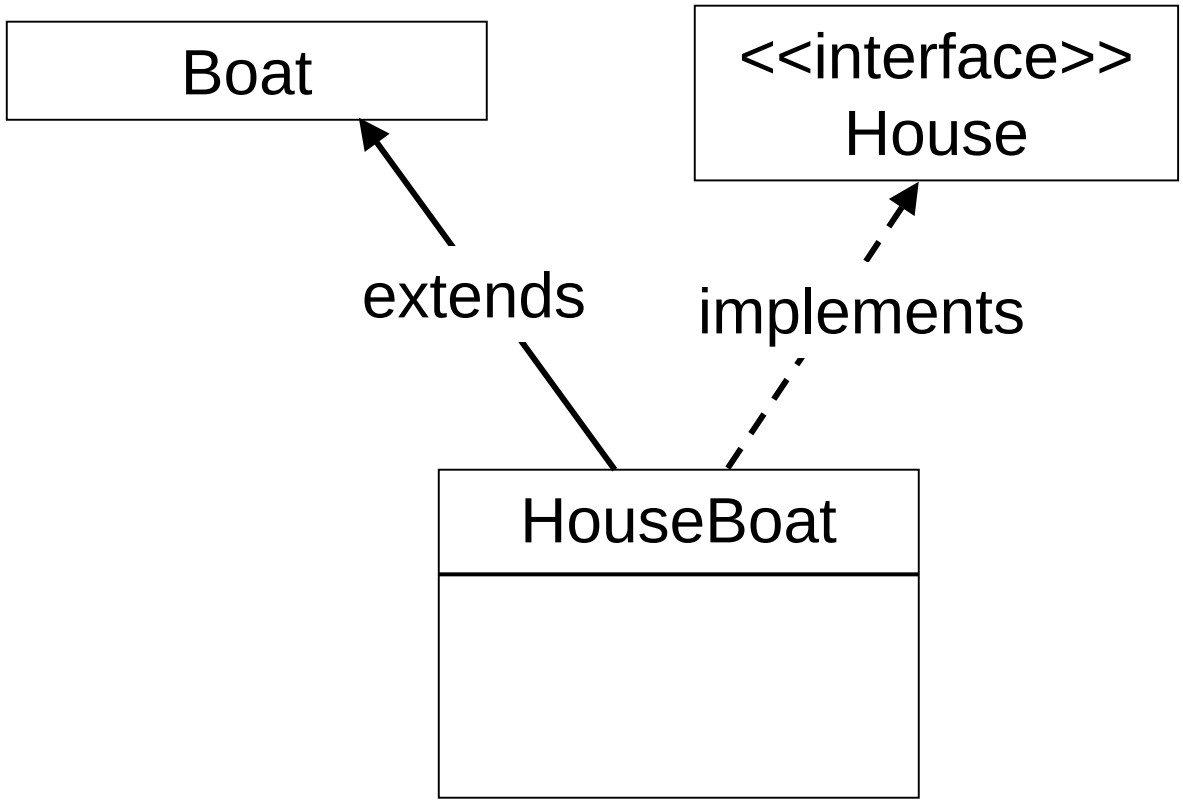
Multiple inheritance

- What if two superclasses have the same method?

```
class House {
    public void maintenance() {
        // repair foundation
    }
}

class Boat {
    public void maintenance() {
        // paint hull
    }
}
```

- In Java, multiple inheritance is not allowed (only one implementation)

Interface



Interface

- A type definition (but not its implementation)
- A form of contract
- Contains only abstract methods
- Methods always **`public`**
- A class can extend only one other class, but implement many interfaces

Shop, ex

```
class ShopOverload {

   float sum = 0.00f;

   void reset () {
       sum = 0.00f;
   }
   float getTotal () {
       return sum;
   }
   void regSale ( Bulldozer itemSold ) {
       sum += itemSold.price();
   }
   void regSale ( Truck itemSold ) {
       sum += itemSold.price();
   }
   .
   .

}
```

- One method for each item type…
- Solution: Create a superclass! (or..?)

Shop, ex

```
public abstract class Vehicle {
   public abstract float prize();
}

public class Bulldozer extends Vehicle {
   public float prize() {
       return 1500000.0f;
   }
}

public class Truck extends Vehicle {
   public float prize() {
       return 1200000.0f;
   }
}
```


Shop, ex

```
class ShopOverload {

   float sum = 0.00f;

   void reset () {
       sum = 0.00f;
   }
   float getTotal () {
       return sum;
   }
   void regSale ( Vehicle itemSold ) {
       sum += itemSold.price();
   }

}
```

- What if we want to start selling apples?

Shop, ex

```
class ShopOverload {

   float sum = 0.00f;

   void reset () {
       sum = 0.00f;
   }
   float getTotal () {
       return sum;
   }
   void regSale ( Vehicle itemSold ) {
       sum += itemSold.price();
   }
   void regSale ( AppleBag itemSold ) {
       sum += itemSold.price();
   }

}
```

- Again one method for each item type…

Shop, ex

```
interface Valuable {
   public float price();
}
class Bulldozer extends Vehicle implements Valuable
{
   public float price() {
      return 1500000.00f;
   }
}
class Truck extends Vehicle implements Valuable {
   public float price() {
      return 1200000.00f;
   }
}
class AppleBag extends Fruit implements Valuable {
   static float PRICE_PER_MASSUNIT = 14.90f;
   float mass;
   AppleBag(float mass) { this.mass = mass; }
   public float price() {
      return mass * PRICE_PER_MASSUNIT;
   }
}
```


Shop, ex

```
class ShopInterface {

    int sum = 0;

    void reset () {
        sum = 0;
    }

    int getTotal () {
        return sum;
    }

    void regSale ( Valuable itemSold ) {
        sum += itemSold.price();
    }

}
```


Shop, ex

```
class Shop {

   public static void main(String[] args) {

      ShopOverload shop1 = new ShopOverload();
      shop1.regSale(new Bulldozer());
      shop1.regSale(new Truck());
      shop1.regSale(new AppleBag(1.2f));
      System.out.println(shop1.getTotal());


      ShopInterface shop2 = new ShopInterface();
      shop2.regSale(new Bulldozer());
      shop2.regSale(new Truck());
      shop2.regSale(new AppleBag(2.3f));
      System.out.println(shop2.getTotal());

   }

}
```


Femkamp, ex

```
interface Vapenvårdare{ void vårdaVapen(); }
interface Simmare{ void  simma(); }
interface Skytt extends Vapenvårdare{ void skjut(); }
interface Ryttare{ void rid(); }
interface Löpare{ void spring(); }
interface Fäktare extends Vapenvårdare{ void fäkta(); }

class FemKampare implements Simmare, Skytt, Ryttare, Löpare,
Fäktare{
    public void  simma(){System.out.println("simmar");}
    public void  skjut(){System.out.println("skjuter");}
    public void  rid(){System.out.println("rider");}
    public void  spring(){System.out.println("springer");}
    public void  fäkta(){System.out.println("fäktar");}
    public void vårdaVapen(){System.out.println("vårdar
vapen");}
}
```


Femkamp, ex

```
class Spel{
   void a(Simmare s){s.simma();}
   void b(Skytt s){s.skjut();}
   void c(Ryttare r){r.rid();}
   void d(Löpare l){l.spring();}
   void e(Fäktare f){f.fäkta();}
   void f(Vapenvårdare vv){vv.vårdaVapen();}

   public static void main(String [] args){
      Spel spelet = new Spel();
      FemKampare fk = new FemKampare();
      spelet.a(fk); //fk behandlas som en Simmare
      spelet.b(fk); //fk behandlas som en Skytt
      spelet.c(fk); //fk behandlas som en Ryttare
      spelet.d(fk); //fk behandlas som en Löpare
      spelet.e(fk); //fk behandlas som en Fäktare
      spelet.f(fk); //fk behandlas som en Vapenvårdare
    }
}
```


static

- Variables and methods in a class can be **`static`**
- Cannot use instance variables in a **`static`** method
- Class specific, not object specific
- Kind of like “global”
- “Class-oriented” programming instead of object-oriented programming…
- Use **`static`** only when necessary!

static

```
public class Ball{

   private static int counter;
   private int ballNo;

   public Ball(){
      ballNo = counter++;
   }

   public int getBallNo(){
     return ballNo;
   }

   public static int getTotalNoOfBalls (){
     return counter;
   }

}
```


Exercise, bank 1

- Write a class **`Account`** representing a bank account
- The account should not be able to handle overdrafts
- Three constructors + methods for deposits and withdrawals
- Negative amounts not allowed (depositing or withdrawing)
- It must have a unique account number
- Write class **`TestBank`** to test **`Account`**

MODULE: Standard libraries

- java.lang
- java.lang.Object
- Wrapper-classes
- Strings

Package java.lang

- Contains the most common and fundamental functionality
- Wrappers for primitive types
- Automatically imported

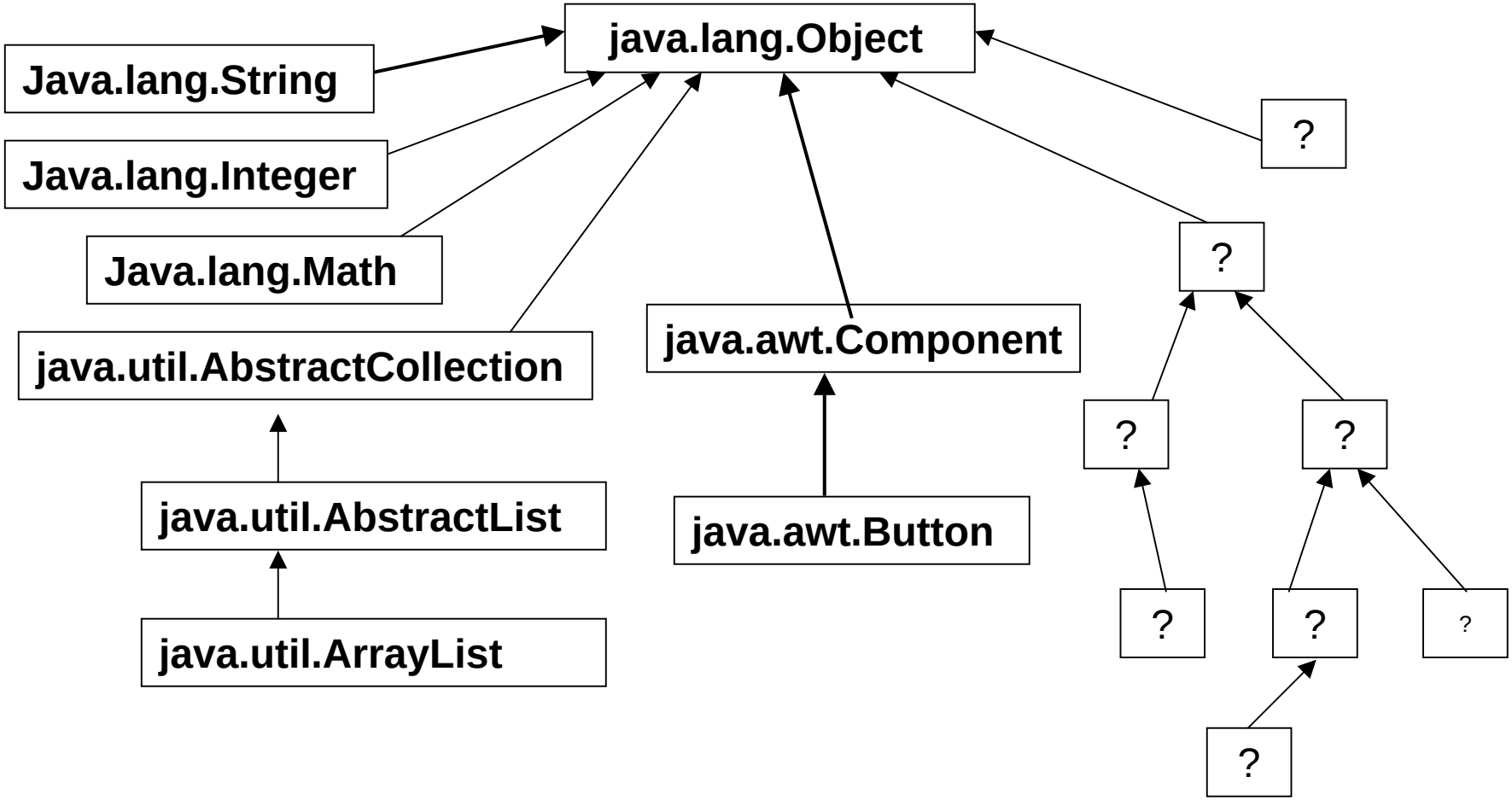
```
import java.lang.*; // not necessary
```


Package java.lang

- contains ca 100 classes and interfaces (subpackages included), e.g.:

Object	Number	Thread
String	Long	SecurityManager
StringBuffer	Class	ClassLoader
Integer	Error	System
Boolean	Exception	Math
Float	OutOfMemoryError	Runnable

java.lang.Object



java.lang.Object

- The base class for all other classes
- Has methods that are used or overridden by inheriting classes, e.g.:

```
clone()
hashCode()
equals()
toString()
```


Thing, ex

```
class Thing extends Object implements Cloneable{
   public String id;
   public Object clone() throws CloneNotSupportedException {
         return super.clone();
   }
   public boolean equals(Object obj){
      boolean result = false;
      if ((obj!=null) && obj instanceof Thing ){
         Thing t = (Thing) obj;
         if (id.equals(t.id)) result = true;
      }
      return result;
   }
   public int hashCode(){
      return id.hashCode();
   }
   public String toString() {
      return ”Thing is: ”+id;
   }
}
```


Thing, ex

```
Thing t1 = new Thing(), t2;

t1.id = “grej”;

t2 = t1; // t1 == t2 and t1.equals(t2)

t2 = (Thing) t1.clone(); // t2!=t1 but t1.equals(t2)

t2.id = “pryl”; // t2!=t1 and !t1.equals(t2)

Object obj = t2;

System.out.println(obj); //””Thing = pryl”
```


Wrapper classes

- Primtive types are effective and requires little memory
- Wrapper classes encapsulates primitive types in objects
- All primitive types have corresponding wrappers
- Wrapper objects are immutable
- Wrapper objects allow for primitives to be used in collections etc.

Wrapper classes

Primitiv	Wrapper Klass
boolean	Boolean
byte	Byte
char	Chararacter
short	Short
Int	Integer
long	Long
float	Float
double	Double

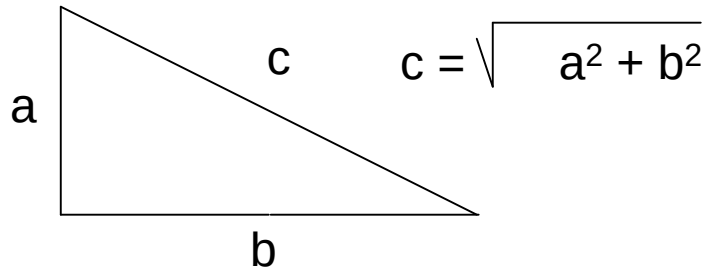
```
int iPrim = 10;
Integer iObj = new Integer(iPrim);
System.out.println(
                iObj.toHexString());

Integer iElem = new Integer(”123”);
int i = iElem.intValue();

Vector v = new Vector();
v.add(iElem);
```


java.lang.Math

- Class Math contains mathematical constants and methods for fundamental mathematical functions, e.g. exponential, squareroot and trigonometric functions
- All in Math is **`static`**..

```
static double PI;
static double E
static int abs(int a)
static int max(int a, int b)
static double sqrt(double a)
static double random()
static long round(double a)
etc.
```


java.lang.Math, ex.




```
double a=5,b=10,c;
c = Math.sqrt(Math.pow(a,2)+Math.pow(b,2));
System.out.println("c is "+Math.round(c)); // ”c is 11”
```

java.lang.String

- String is a class, not a primitive type
- immutable – cannot be modified
- Overloaded operators = and +
- The statement

```
String s = new String(”hej”);
```

  is equivalent with:

```
char chrArr[] = {’h’,’e’,’j’};
String s2 = new String(chrArr);
```


java.lang.String

Assignment, string literals

```
String s1 = ”Hej!”;
String s2 = new String(”Hej!”);
```

Concatenation

```
String s1 = ”1”;
String s2 = s1+”2”+3;  // → ”123”
```

Is never written over..

```
String s = new String(”1”);
s = ”2”; // new String created!
```


java.lang.String, concatenation

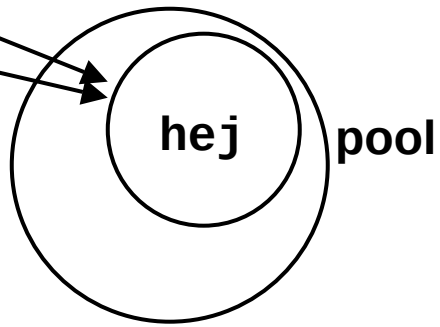
```
s1 = s1 + s1 + 123 + ”hejdå” ;

String title = "Dr:";
String fname = "Inge";
String ename = "Glad";
System.out.println("title: " + title + "\n" + "fname: "
  + fname + "\n” + "ename: " + ename);
```


java.lang.String, assignment

```
String s1 = "hej";
String s2 = "hej";
String s3 = new String("hej");
String s4 = new String("hej");

// which ones will be written?
if (s1==s2) System.out.println("s1==s2");
if (s3==s4) System.out.println("s3==s4");
if (s1.equals(s2)) System.out.println("s1.equals(s2)");
if (s3.equals(s4)) System.out.println("s3.equals(s4)");
```


java.lang.String, assignment

```
String s1 = ”hej”;
String s2 = ”hej”;
```




```
String s1 = new String(”hej”);
```

java.lang.String, methods

- `char charAt(int index)`
- `String concat(String string)`
- `boolean equals(Object anObject)`
- `String replace(char oldChar, char newChar)`
- `String toLowerCase()`
- `String toUpperCase()`

java.lang.StringBuffer

- represents a sequence of characters
- Is mutable (can be modified)
- More effective when working with dynamic strings

```
StringBuffer sBuf = new StringBuffer();
sBuf.append(“abc”);
sBuf.append(“def”);
```


StringBuffer, ex

```
StringBuffer bufStr = new StringBuffer(); // empty

bufStr.append("bcda!e"); // contains “bcda!e”

char c = bufStr.charAt(3); // extracts ‘a’

bufStr.delete(3,5); // now contains “bcde”

bufStr.insert(0,c); // now contains “abcde”

String str = bufStr.toString(); // extract the string

System.out.println(str); // write it to console using String

System.out.println(bufStr); // write it to console directly
```


Exercise, toString()

- Write a toString() method for a subclass

Exercise, bank 2

- Override the **`toString`** method in class **`Account`**
- Type check the name string. If it contains two names, each should start with a capital letter. Implement the methods:
  - **`isValidName`**
  - **`formattedName`**