Java™ Basics and Object-based Programming

Reading assignment

- Finish Chapter 1 and start on chapter 2!

Topics to be covered....

- So far
  - Classes and Objects
  - Methods
  - Primitive Types
  - Variables
  - References
  - Parameter passing
  - Arrays
  - Control Flow
- Today (and tomorrow ...)
  - Expressions, operators (see part2 notes)
  - Castings
  - Input and Output
  - Strings
  - Packages
  - Inheritance (intro)
  - Modifiers
  - Static members of a class
- Next (may start tomorrow)
  - Designing your programs
  - Object oriented design
  - Inheritance in Java
  - Interfaces
  - Exceptions

Casting

- We can take a variable of one type and **cast** it into a variable of another type
- Syntax:

  (<desired_type>)<variable>;

  e.g. double age = 0.0; (int)age;
- 2 types of casting (base types and wrt objects)

Interactive examples: Casting.java

Casting and base types

- double -> int (may lose precision, does not round up, but truncates)
- E.g.

```
double d1 = 3.2;
double d2 = 3.99;
int i1 = (int)d1;  // i1 = 3
int i2 = (int)d2;  // i2 = 3
double d3 = (double)d1; // d3 = 3.0
```

Casting with operators

- Must do the cast before the operator does its job

```
int i1 = 3; int i2 = 6; double dresult;
dresult = (double)i1/(double)i2; // dresult = 0.5
dresult = i1/i2;   // dresult = 0.0

// this last line performed an integer division
// which is then implicitly cast to a double result
```

Implicit Casting

- Need to be careful!

```
int result, i = 3;
double dresult, d = 3.2;

dresult = i/d;  // dresult?
iresult = i/d;  // iresult?

System.out.println("dresult is : " + dresult);
System.out.println("iresult is : " + iresult);
```

- **General rule: play it safe, explicitly cast!!!**

Implicit Casting with String Objects

- There is one situation in Java when **only** implicit casting is allowed
  - String concatenation!
  - Any time a string is concatenated with any object or base type, that object or base type is automatically converted to a string
  - Examples:

```
String s = (string)4.5;  // wrong!
String u = 22;              // u = “22”, correct.

String u = (int)22; // is this ok?

// But could do
String u = Integer.toString(22);
       // Or implicitly cast!
```

Input and output

- Java provides a rich set of classes for performing i/o
- Java provides classes for simple text i/o using a console window

```
import java.io.*;
```

- Java also provides i/o using a Graphical User Interface (GUI)

```
import java.awt.*; // for drawing
import javax.swing.*; // for widgets
```

Simple text Output

- Output to the console:
  - Very useful for debugging logical errors in your program! And just for understanding the different features in Java
  - **`System.out`** is a static object of type java.io.**`PrintStream`**
  - **`The PrintStream class defines methods for a buffered output stream where the characters are put in a temporary location called a buffer, which is then emptied into the Java console window`**

    - **`print()`**, and **`println()`** methods take the following arguments:
      - Any object (provided it has a **`toString( )`**) method
      - Any string or concatenated strings
      - Any base type (automatically cast to String)

Simple text Input

- Input from the console
  - must **import java.io.*;**
  - **System.in** is an object of type Input**Stream** (abstract class)
    - inputs bytes only (crude)
  - **InputStreamReader** translates bytes to characters.
    - API recommends wrapping an **InputStreamReader** within a **BufferedReader**

```
java.io.BufferedReader stndin;
stndin = new java.io.BufferedReader(new
  java.io.InputStreamReader(System.in));
String input = "";
  // to hold the user's reply to play again
input = stndin.readLine();
```

Text Input cont.

- readLine() – reads a string of characters up to a newline which is not included in the return String (if input is empty, it returns Null)
- read() – reads a single character, if input stream is at the end, it returns a '-1'
- See P. 35 for more details and try some examples!

- These methods also raise an error condition if an input error occurs
- For now use this code (we will discuss exceptions later!)

```
try {
answer =
  Integer.valueOf(stndin.readLine()).intValue();
} catch  (IOException e) {
}
ourCasino.playAllSlotMachines(answer);
```

The surroundings of a class

- Package
  - A class belongs to a named package or the default package

    ```
    package cscc115assignment1;
    ```
  - A class can import packages

    ```
    import javax.swing.*;
    import java.io.*;
    ```

- Inheritance
  - A class can extend another class (i.e., be a *subclass*)

    ```
    public class Manager extends Employee { … }
    public class Model extends Observable { … }
    ```
  - A class can be a *superclass* for another class

- Interfaces
  - A class can implement an interface

    ```
    public class TextView implements Observer { … }
    ```

Packages

- Large software systems have many more classes than lines of code per class. Thus, organizing classes is as important as programming individual classes.
- Java offers the notion of a package to aggregate related classes.
- Classes are assigned to a package using a **package** directive before the class declaration:

```
package packagename;
```

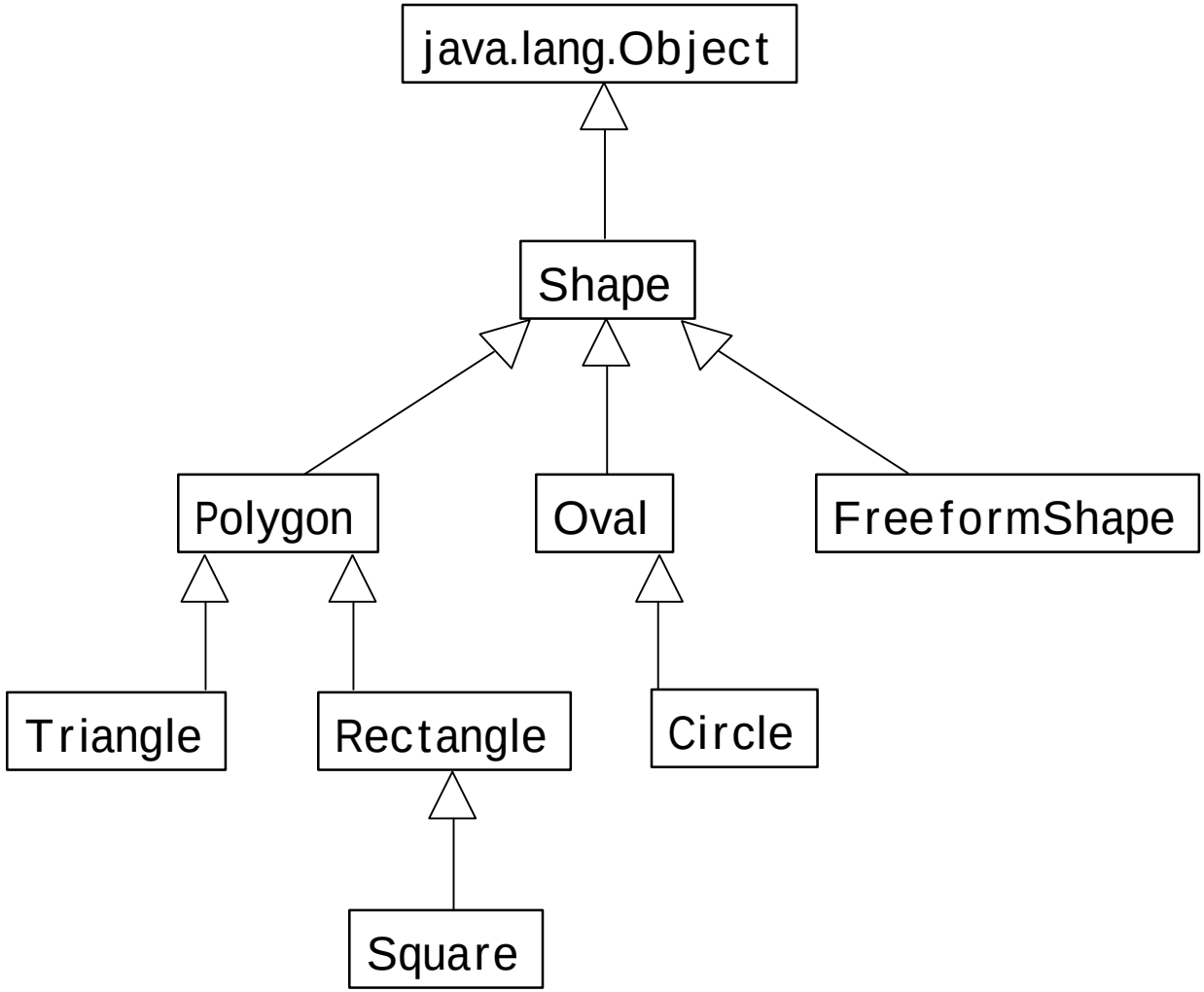
```
package assignment3;
```

- Package names are usually in all lower case.
- Using the **import** directive, packages can be imported (i.e., made available) to classes.

```
import packagename.*;
```

```
import assignment3.*;
```

Inheritance, Is-a, Class Hierarchy



Inheritance Relationship

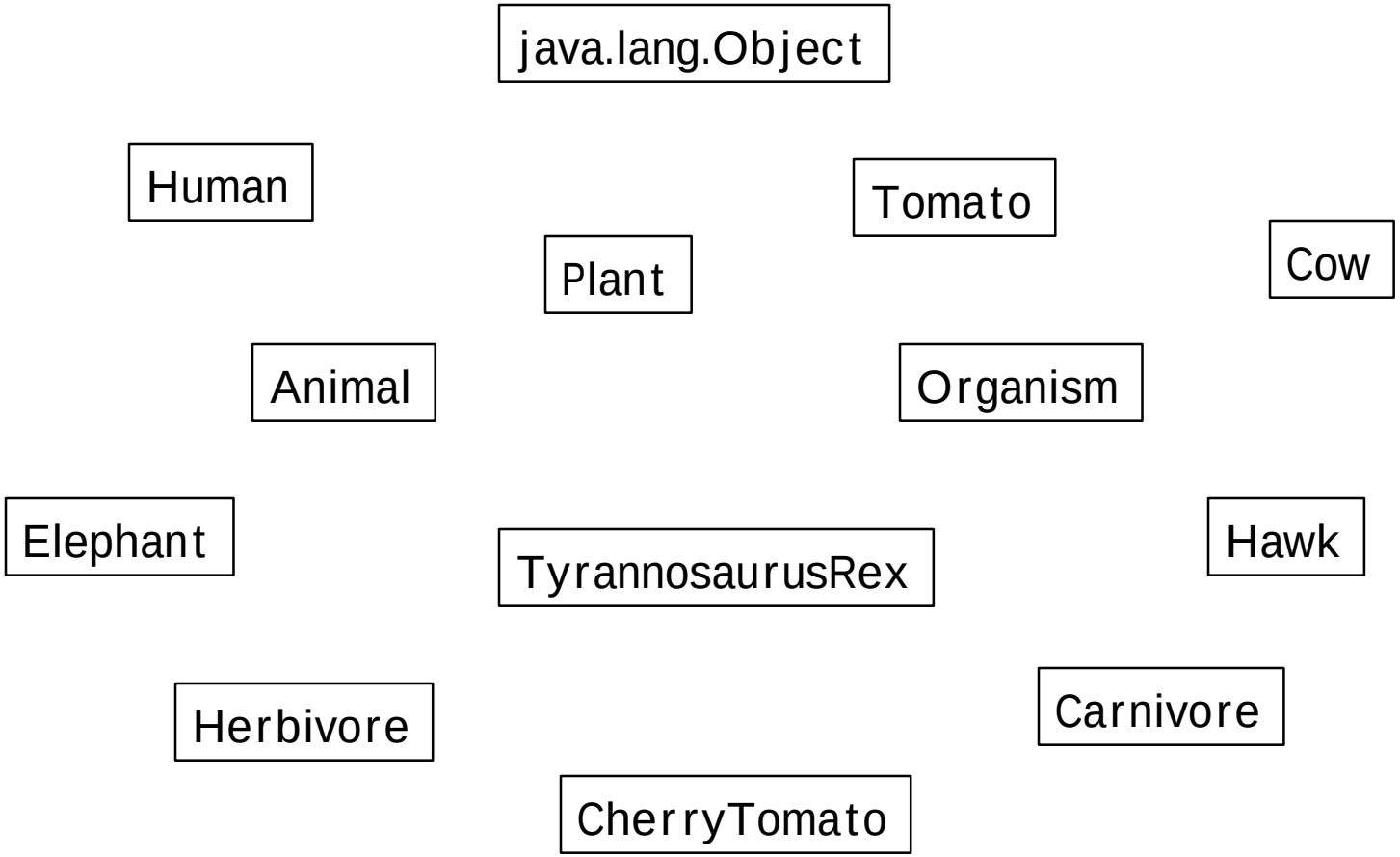
- Subclass
  - *extends* a *superclass* definition with new fields or methods
  - *inherits* the fields and methods of the superclass
  - modifies the meaning of the *superclass*
  - forms an *is-a* relationship with its superclass

- Genealogical terminology
  - the *parent* of a class is its superclass
  - the *children* of a class are its immediate subclasses
  - the *ancestors* of a class are its parents, and their parents...
  - the *descendants* of a class are its children, and their children...

- Since each class has only *one* parent, this is *single inheritance*
- The classes form an *inheritance* or *is-a hierarchy*
- In Java, the Object class is the root of this hierarchy

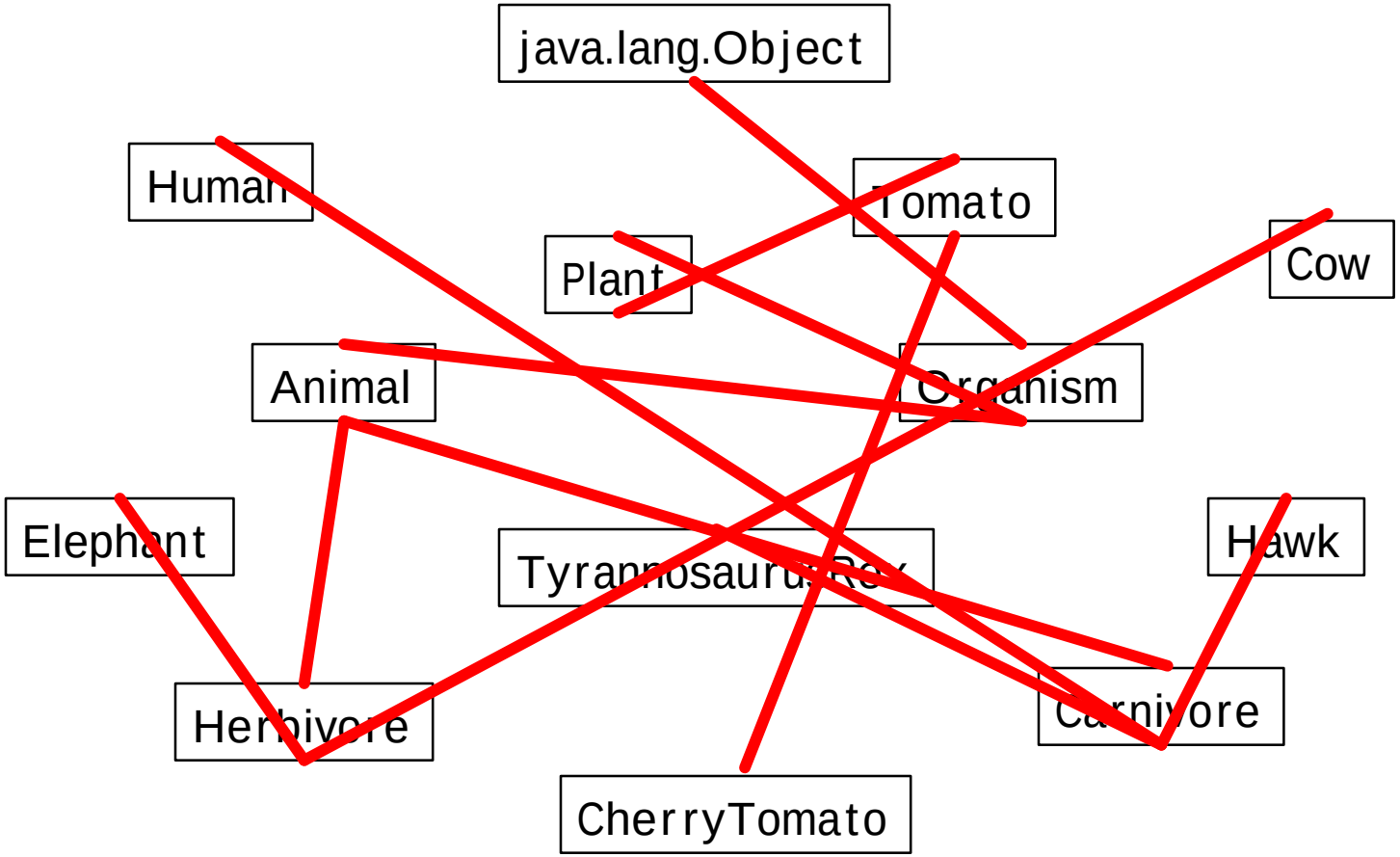
Data Modeling: Inheritance Quiz

- Arrange the classes below into an inheritance hierarchy



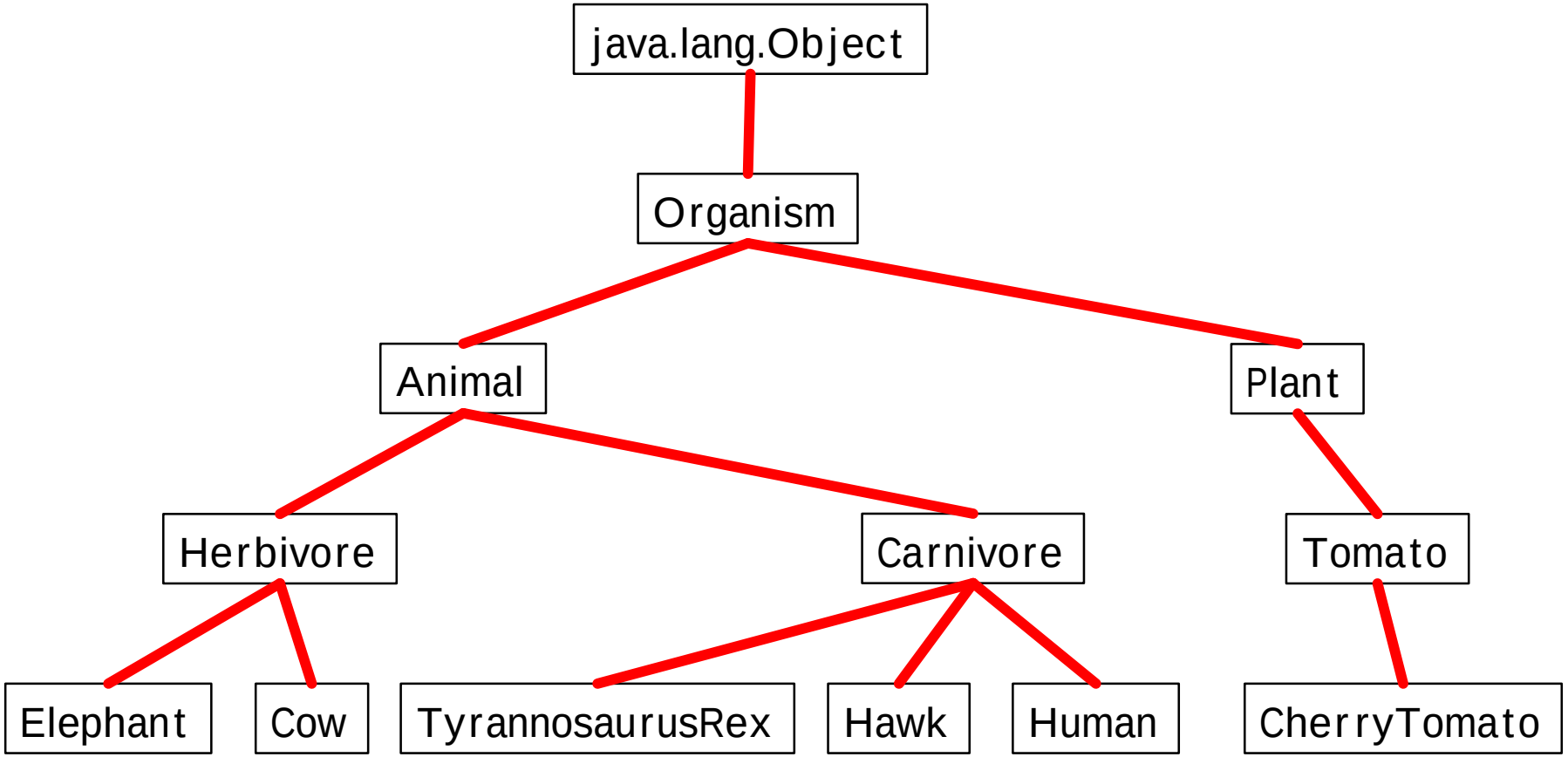
Data Modeling: Inheritance Quiz

- Arrange the classes below into an inheritance hierarchy



Data Modeling: Inheritance Quiz

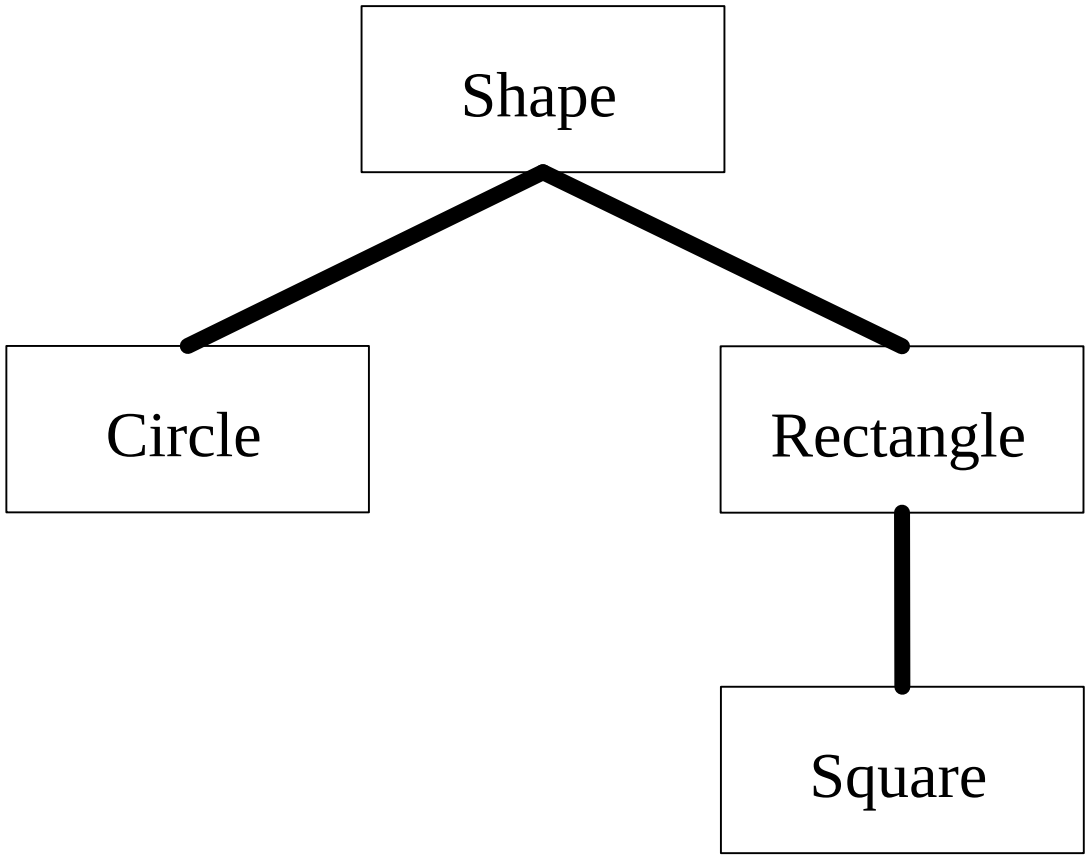
- Arrange the classes below into an inheritance hierarchy



Inheriting and Extending

- A subclass inherits both data (fields) and behavior (methods)
  - inherited members can be accessed as if they were present in the subclass itself
  - constructors and private members are not inherited
- **Overriding** a superclass method
  - A subclass can redefine a superclass method by using the same signature
- **Overloading** of a method
  - A method in the same class or a subclass with the same name but different signature

Classic shape inheritance hierarchy



Interactive examples: Shape.java, Square.java

Class modifiers

- Class modifiers are *optional* keywords preceding the **class** keyword.
- **abstract**
  - The class has at least one **abstract** method (an abstract method has no method body and is preceded by abstract modifier).
  - A class with only **final** instance variables and only **abstract** methods is called an **interface**
- **final**
  - A final class cannot be subclassed
- **public**
  - A **public** class can be instantiated or **extended** by anything in the same package or anything that **imports** this class.
  - Each public class is declared in a separate file; downloadable component.
- friendly (default – if no modifier is specified)
  - Can be used and instantiated by all classes in the same package

Interactive examples: FriendlyClasses.java, Final.java

public, **protected**, **private**, and package modifiers

- These modifiers apply to *both* fields and methods
- **public**
  - Any method can access **public** members
- **protected**
  - Only methods of the same package or subclasses can access protected members
- **private**
  - Only methods of the same class can access **private** members (not even methods in subclasses can access private members!)
- friendly – default (no modifier)
  - Members, which are not **public**, **protected**, or **private**, are called **package members**
  - Only methods in the same package can access package members

Interactive examples: Protected.java, TestingPrivateMethods.java

Constructors -- revisited

- The **abstract**, **static**, and **final** modifiers are not allowed for constructors
- A **public**, no-argument constructor is provided by the Java run-time environment if the class does not define one

Usage modifiers -- fields

- **static**
  - A **static** variable is associated with its class, is shared by all objects of its class, and its storage exists once (i.e., with the class rather than all the objects)
- **final**
  - A **final** variable must be initialized and is read-only after initialization (i.e., it is constant)
  - **final** variables are usually also declared **static** so that storage is allocated only once for an entire class
  - The naming convention for **final** variables is all upper case
  - **final** variables are often declared in interfaces
  - **final** variables that point to objects will always point to the same object (even if it changes state)

Interactive examples: UsageModifiers.java

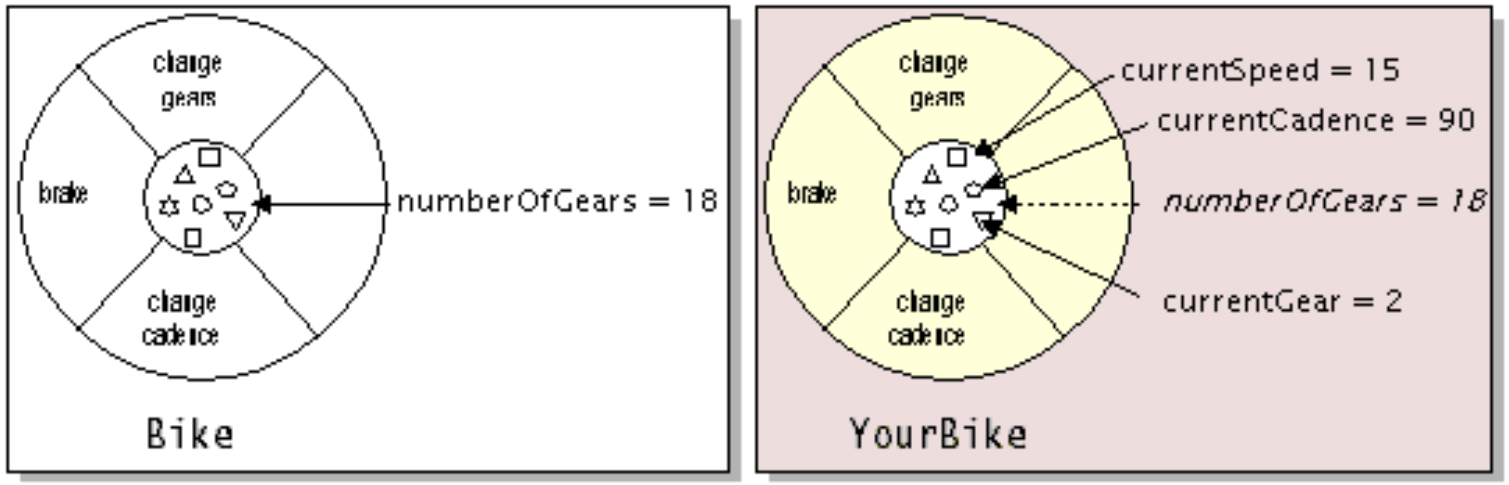
Usage modifiers -- methods

- **static**
  - A **static** method is associated with its class and is shared by all objects of its class (i.e., with the class rather than all the objects)
  - The **static** fields should only be changed by **static** methods (as long as the fields are not declared as **final**)
- **final**
  - A **final** method cannot be overridden by a subclass.

Interactive examples: UsageModifiers.java

Classes and Instances of Classes

- Class variables (e.g. numberOfGears) are used to hold state common to all instances
- Class methods can be invoked from a class, but instance methods must be called using a particular instance (see these later)



Class **Instance of a Class**

Interactive examples: StaticMethod.java

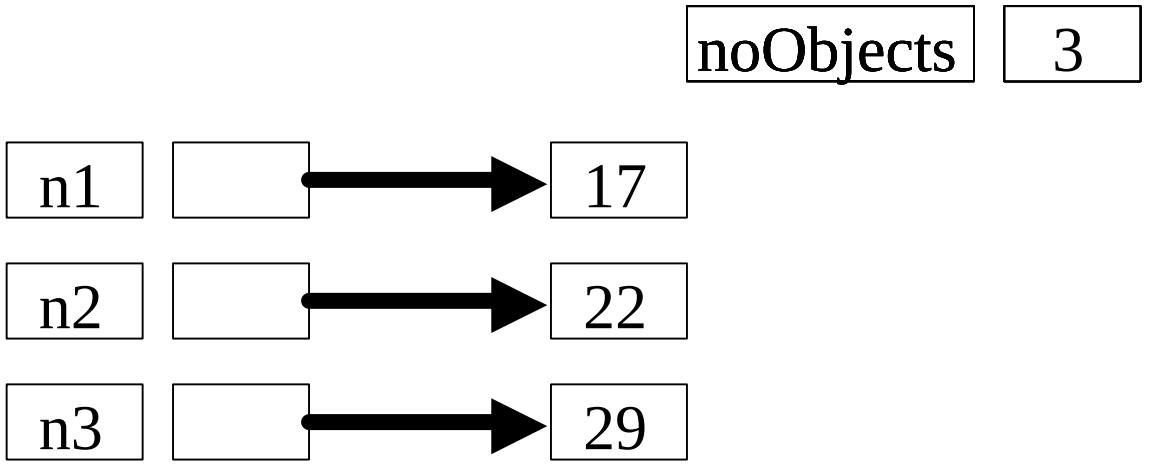
static members

- Example

```
public class Node {
   private static int noObjects = 0;
   private int id;
   public Node(int k) { id=k; noObjects++ }
   public static int getNoObjects() {
      return noObjects;
   }
}
System.out.println(Node.noObjects()); // 0
Node n1 = new Node(17);
System.out.println(Node.noObjects()); // 1
Node n2 = new Node(22);
System.out.println(Node.noObjects()); // 2
Node n3 = new Node(29);
System.out.println(Node.noObjects()); // 3
```

static field

- Execute program



Notes….

- We can both static and final modifiers
- But only one scope modifier (can't say it will be both public and private, but we can say a member will be both final and static)

Usage modifiers -- methods

- **abstract**
  - An **abstract** method has no body.
  - The parameter list is followed by a semicolon to terminate the **abstract** method declaration.
  - **abstract** methods may only appear within an **abstract** class.
  - **abstract** methods are typically overridden by subclasses.

Interactive examples: Abstractmethods.java

Modifiers (Quiz)

access modifiers

Modifier	Can be applied to		
	Classes	Fields	Methods
public			
protected			
(default)			
private			
static			
final			
abstract			

Other Modifiers

- synchronized, native, transient, volatile, strictfp

Accessing members -- revisited

- Dot notation
- Accessing instance members

```
objectName.classMember
objectName.field
objectName.method()
```

- Accessing static members

```
className.classMember
className.field
className.method()
```