Java Programming

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Java 415
Summer 2024

```java
class Lecture6 {

        // Object-Oriented Programming (OOP)

}

// Keywords:
class, new, this, static, null, extends, super, final, abstract,
interface, implements, protected, package, import, enum
```

Object & Class

- An object is an entity to maintain its own states in fields[1] and provide accessory methods (or actions) on fields.
- To create objects of this type, we define a new class as follows:
  - designate a name with the first letter capitalized;
  - declare data and function members in the class body.
- Note that a class is one way to create reference types.[2]
- In this sense, defining a new class is to define a new type!
  - You are building a new world!

[1] It is also called attributes, properties, and whatsoever.

[2] For example, we will visit more reference types, like interface and enum.

Example: Points

- For any 2D point, the class could look like the code snippet below:

```
1 public class Point {
2
3     // Data members.
4     double x, y;
5
6 }
```

- Then we manipulate two points in another class, shown in the next page.

```
1  public class PointDemo {
2
3      public static void main(String[] args) {
4
5          Point p1 = new Point();
6          p1.x = 10;
7          p1.y = 20;
8
9          Point p2 = new Point();
10         p2.x = 30;
11         p2.y = 40;
12
13         System.out.printf("(%.2f, %.2f)\n", p1.x, p1.y);
14         System.out.printf("(%.2f, %.2f)\n", p2.x, p2.y);
15
16     }
17
18 }
```

- Could you draw the current state of memory allocation when the program reaches Line 15?

Encapsulation

- Each member may have an access modifier, say public and private.
  - public: accessible by all classes.
  - private: accessible only within its own class.
- In OOP practice, the internals like data members should be isolated from the outside world.
- So all fields should be declared private.
- Note that the private modifier does not guarantee any information security.[3]
  - What private is good for maintainability and modularity.[4]

[3] Thanks to a lively discussion on January 23, 2017.

[4] Read this article Are private members really more "secure" in Java?

- We then expose the public methods which perform actions on these fields, if necessary.
- For example,
  - getters: return one specific field.
  - setters: assign new value to the field.
- For example, getX() and getY() are the getters; setX() and setY() are the setters of the **Point** class.

Example: Point (Encapsulated)

```
1  public class Point {
2
3      // Data members: fields or attributes
4      private double x, y;
5
6      // Function members: methods
7      public double getX() { return x; }
8      public double getY() { return y; }
9      public void setX(double a) { x = a; }
10     public void setY(double b) { y = b; }
11
12 }
```

Constructors

- To create an object of the type, its constructor is invoked by the new operator.
- You can define constructors with parameters if necessary.
  - For example, one can initialize the object during the creation.
- Note that a constructor has its name identical to the class name and has no return type. (Why?)
- If you don't define any explicit constructor, Java assumes a default constructor for you.
- Adding any explicit constructor disables it but you can recover it by youself.

Parameterized Constructors: Example

```
1  public class Point {
2      ...
3      // Default constructor
4      public Point() {
5          // Do something in common.
6      }
7
8      // Parameterized constructor
9      public Point(double a, double b) {
10         x = a;
11         y = b;
12     }
13     ...
14 }
```

- You can initialize an object when the object is allocated.

Self Reference: this

- You can refer to any (instance) member of the current object by using this, within its (instance) methods and constructors.
- The most common situation to use this is that a field is shadowed by method parameters.
  - It is a direct result of the shadow effect.
- You can also use this to call another constructor of the class, say this() calling the default constructor, if existing.

Example: Point (Revisited)

```
1  public class Point {
2      ...
3      public Point(double x, double y) {
4
5          this.x = x;
6          this.y = y;
7
8      }
9      ...
10 }
```

- However, the this operator cannot be used in static methods.

Instance Members v.s. Static Members

- Before this lecture, every method is declared with static.
  - For example, the first static method is the main method.
- Notice that all members of the class are declared without static since we start this lecture.
- These members are called instance members, available only after one object is created.
- Semantically, each object has its own states, associated with the accessory methods applying on.
- For example, getX() could be invoked and return the x value for some specific **Point** object.
- In other words, you cannot invoke getX() without an existing **Point** object.

Instance Members v.s. Static Members

- A static variable occupies only one space, shared among the class and its objects.
- You can refer to these static members by calling the class name in absence of any instance.
  - For example, **Math**.PI.
- In particular, static methods perform algorithms.
  - For example, **Math**.random() and **Arrays**.sort().
- However, static methods cannot access to instance members directly. (Why?)
- You may try static initialization blocks.[5]

[5] See https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html.

Example: Distance Between Points (1/2)

```
1  public class Point {
2
3      /* Ignore the previous part. */
4
5      public double getDistanceFrom(Point that) {
6          return Math.sqrt(Math.pow(this.x - that.x, 2)
7                           + Math.pow(this.y - that.y, 2));
8      }
9
10     public static double measure(Point first, Point second) {
11         return Math.sqrt(Math.pow(first.x - second.x, 2)
12                          + Math.pow(first.y - second.y, 2));
13
14     }
15
16 }
```

- Note that you cannot use this in static context.

Example: Distance Between Points (2/2)

```
1  public class PointDemo {
2
3      public static void main(String[] args) {
4
5          /* Ignore the previous part. */
6          System.out.println(p1.getDistanceFrom(p2));
7          System.out.println(Point.measure(p1, p2));
8
9      }
10 }
```

- Both methods produce the same result.
- It concludes that
  - if the object keeps its own states, then declare non-static variables for those;
  - one can deal with data with both static or non-static methods.

Digression: Design Patterns

- Design patterns are a collection of general reusable solutions to a commonly occurring problem in software design.[6]
- These patterns fulfill experience reuse instead of code reuse.
- To my personal experience, OOP syntax is structural skeleton; design patterns are flesh and blood.
- If you wonder why we need OOP and how to exploit it, I suggest that you could follow any textbook[7] or studying materials for design patterns.

[6] Gamma et al. (1994).

[7] For example, Freeman and Robson (2020): Head First Design Patterns.

Example: Singleton Pattern

- For some situations, you need only one object of this type in the system.

```
1  public class Singleton {
2
3      // Do not allow to invoke the constructor by others.
4      private Singleton() { }
5
6      // Will be ready as soon as the class is loaded.
7      private static Singleton instance = new Singleton();
8
9      // Only way to obtain this singleton by the outside world.
10     public static Singleton getInstance() {
11         return instance;
12     }
13
14 }
```

Object Elimination: Garbage Collection (GC)[8]

- JVM handles object deallocation by one daemon thread called GC.
- GC reclaims the memory space occud by the objects which are no longer being used (referenced) by the application.
  - To make the object unreferenced, simply assign null to the reference variable.
- You can invoke **System**.gc() to execute a deallocation procedure.
- However, frequent invocation of GC is time-consuming.

[8] See Java Garbage Collection Basics.

Design Tool: Unified Modeling Language (UML)[10]

- We could conduct one object-oriented analysis and design by using UML which specifies, visualizes, constructs, and documents the artifacts of software systems and business modeling.[9]

[9] You could try some UML softwares, say StarUML.

[10] See Design and UML Class Diagrams.

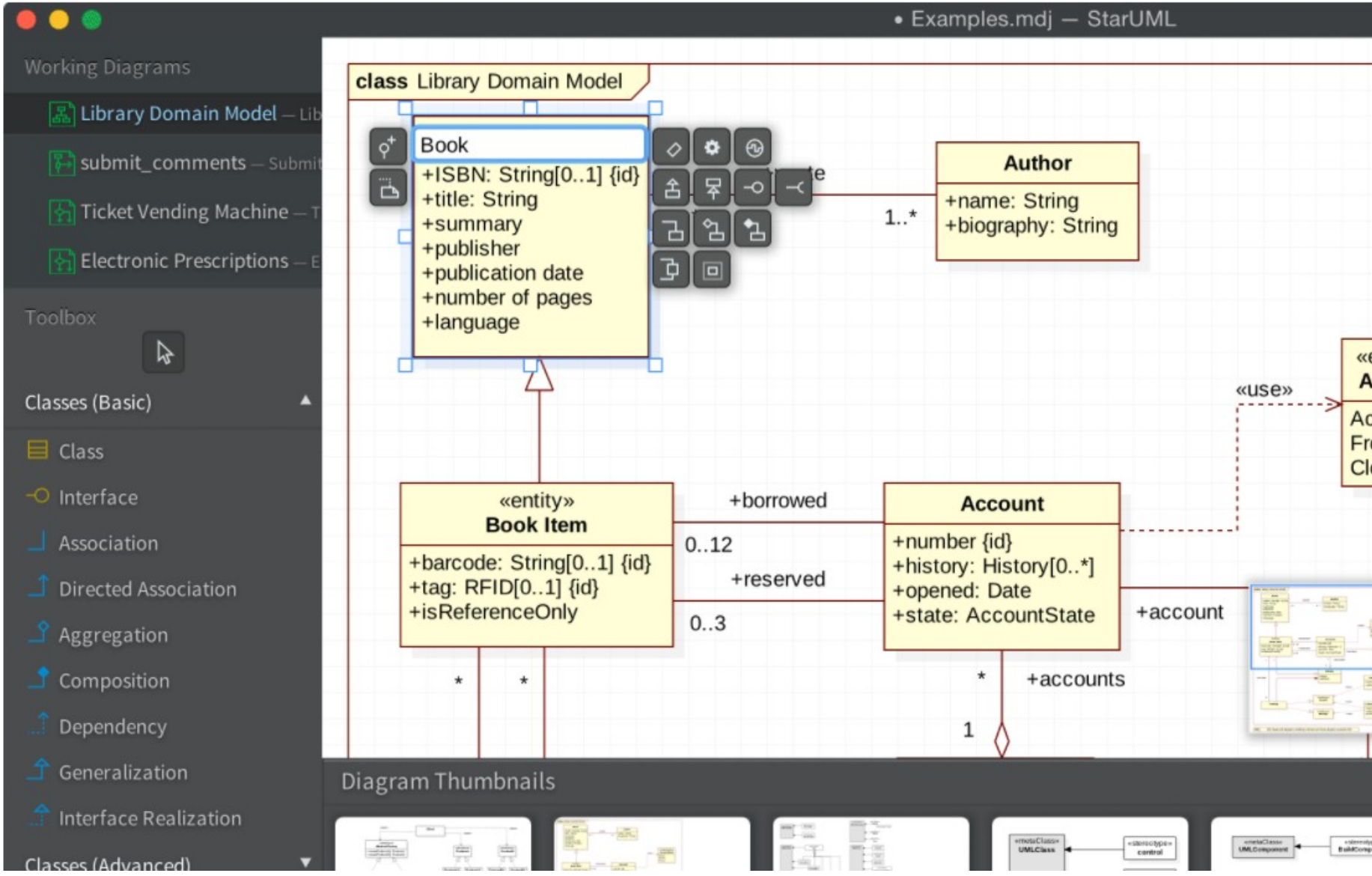
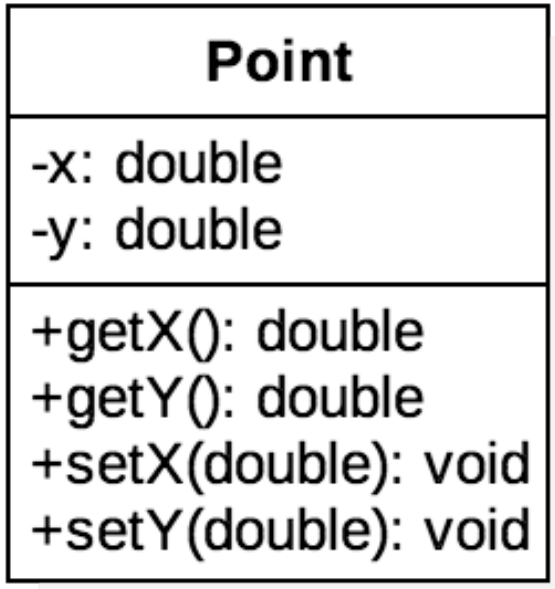


Photo credit: screenshot from http://staruml.io/.

Example: Class Diagram

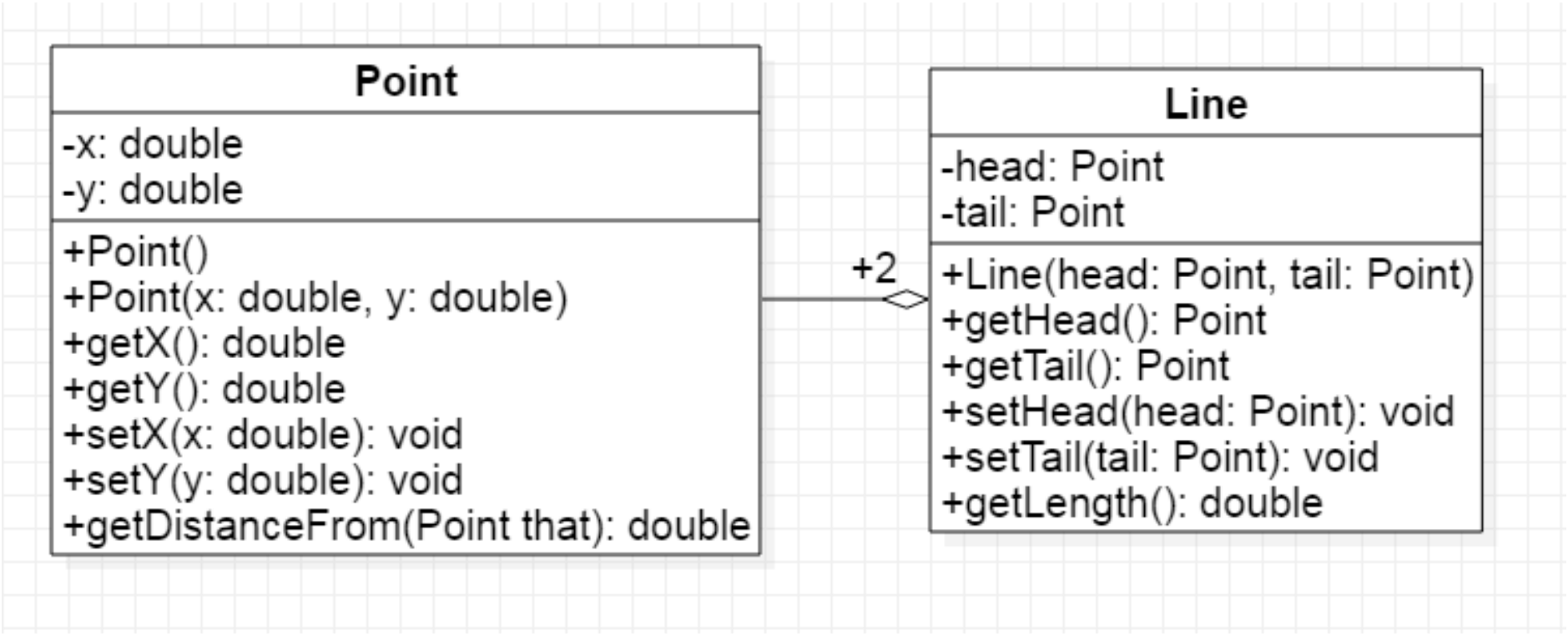


- $+$ refers to public.
- $-$ refers to private.

HAS-A Relationship

- Association is a weak relationship where all objects have their own lifetime and there is no ownership.
  - For example, teacher $\leftrightarrow$ student; doctor $\leftrightarrow$ patient.
- If A uses B, then it is an aggregation, stating that B exists independently from A.
  - For example, knight $\leftrightarrow$ sword; company $\leftrightarrow$ employee.
- If A owns B, then it is a composition, meaning that B has no meaning or purpose in the system without A. (We will see this later.)
  - For example, house $\leftrightarrow$ room.

Example: Lines (Aggregation)



- $+2$: one **Line** object uses two **Point** objects.

```
1  public class Line {
2
3      private Point head, tail;
4
5      public Line(Point p1, Point p2) {
6          head = p1;
7          tail = p2;
8      }
9
10     /* Ignore some methods. */
11
12     public double getLength() {
13         return head.getDistanceFrom(tail);
14     }
15
16 }
```

- In Line 13, we don't reinvent the wheel if the **Point** class is well-designed.

```
1  public class LineDemo {
2
3      public static void main(String[] args) {
4
5          Point p1 = new Point(1, 2);
6          Point p2 = new Point(3, 4);
7          Line l = new Line(p1, p2);
8
9          ...
10
11     }
12
13 }
```

- Make sure that you can make a sketch of the memory allocation for these three objects.

Exercise: Circles

```
1  public class Circle {
2
3      private Point center;
4      private double radius;
5
6      public Circle(Point c, double r) {
7          center = c;
8          radius = r;
9      }
10
11     public double getArea() {
12         return radius * radius * Math.PI;
13     }
14
15     public boolean isOverlapped(Circle that) {
16         return this.radius + that.radius >
17                this.center.getDistanceFrom(that.center);
18     }
19
20 }
```

First IS-A Relationship: Class Inheritance

- We can define new classes by inheriting states and behaviors commonly used in predefined classes (aka prototypes).
- A class is a subclass of some class, which is called the superclass, by using the extends keyword.
- For example,

```
1 // Superclass (or parent class)
2 class A {
3     void doAction() { } // A can run doAction().
4 }
5
6 // Subclass (or child class)
7 class B extends A { } // B can also run doAction().
```

- Note that Java allows single inheritance only.

Example: Human & Dog

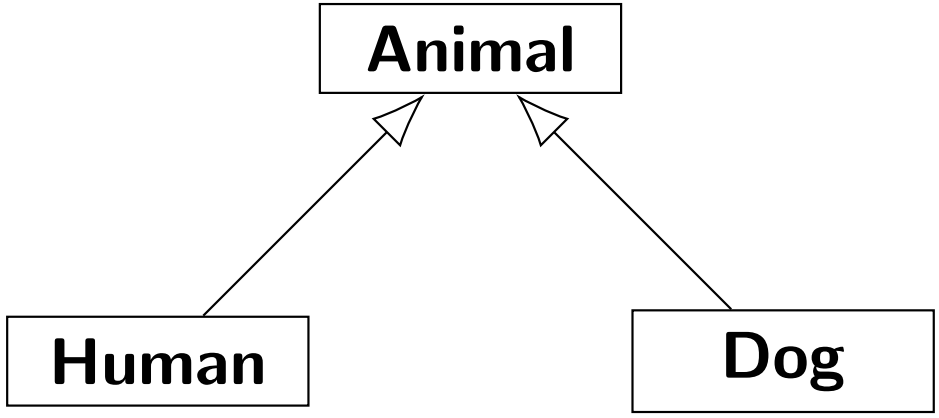
Photo credit: https://www.sunnyskyz.com/uploads/2016/12/nlf37-dog.jpg

Before Using Inheritance

```java
public class Human {

    public void eat() { }
    public void exercise() { }
    public void writeCode() { }

}
```

```java
public class Dog {

    public void eat() { }
    public void exercise() { }
    public void wagTail() { }

}
```

After Using Inheritance



- Extract the part shared between **Human** and **Dog** to another class, say **Animal**, as the superclass.

```java
public class Animal { // extends Object; implicitly.

    public void eat() { }
    public void exercise() { }

}
```

```java
public class Human extends Animal {

    public void writeCode() { }

}
```

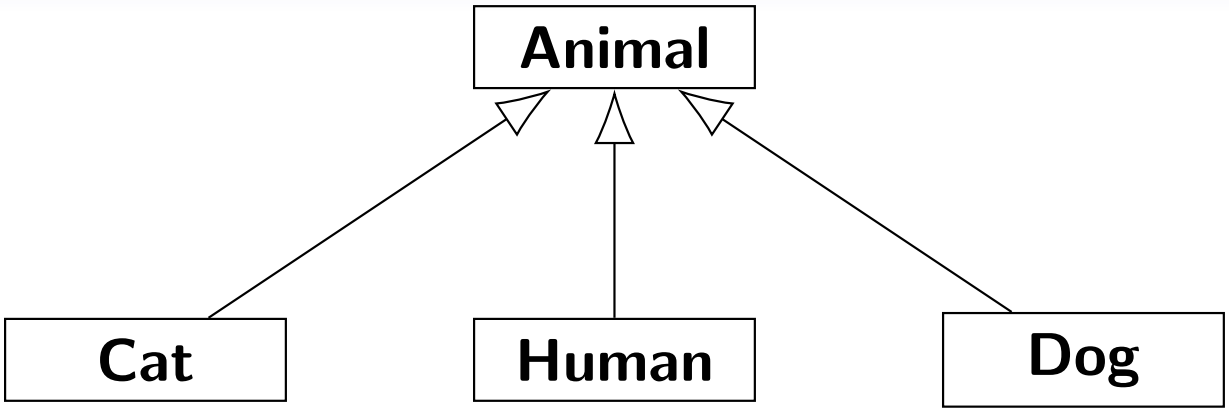
```java
public class Dog extends Animal {

    public void wagTail() { }

}
```

```java
public class InheritanceDemo {

    public static void main(String[] args) {

        Human arthur = new Human();
        arthur.eat();          // Arthur can eat.
        arthur.exercise();     // Arthur can do exericse.
        arthur.writeCode();    // Arthur can write code.
        arthur.wagTail();      // Oops. Arthur has no tail.

        Dog lucky = new Dog();
        lucky.eat();           // Lucky can eat.
        lucky.exercise();      // Lucky can do exercise.
        lucky.writeCode();     // Oops. Lucky cannot write code.
        lucky.wagTail();       // Lucky can wag its tail.

    }

}
```

Exercise: Add **Cat** to Animal Hierarchy[11]

https://cdn2.ettoday.net/images/2590/2590715.jpg

[11] See also https://en.wikipedia.org/wiki/Kneading_(cats) and https://petsmao.nownews.com/20170124-10587.




```
1 public class Cat extends Animal {
2
3     public void makeBiscuits() { }
4
5 }
```

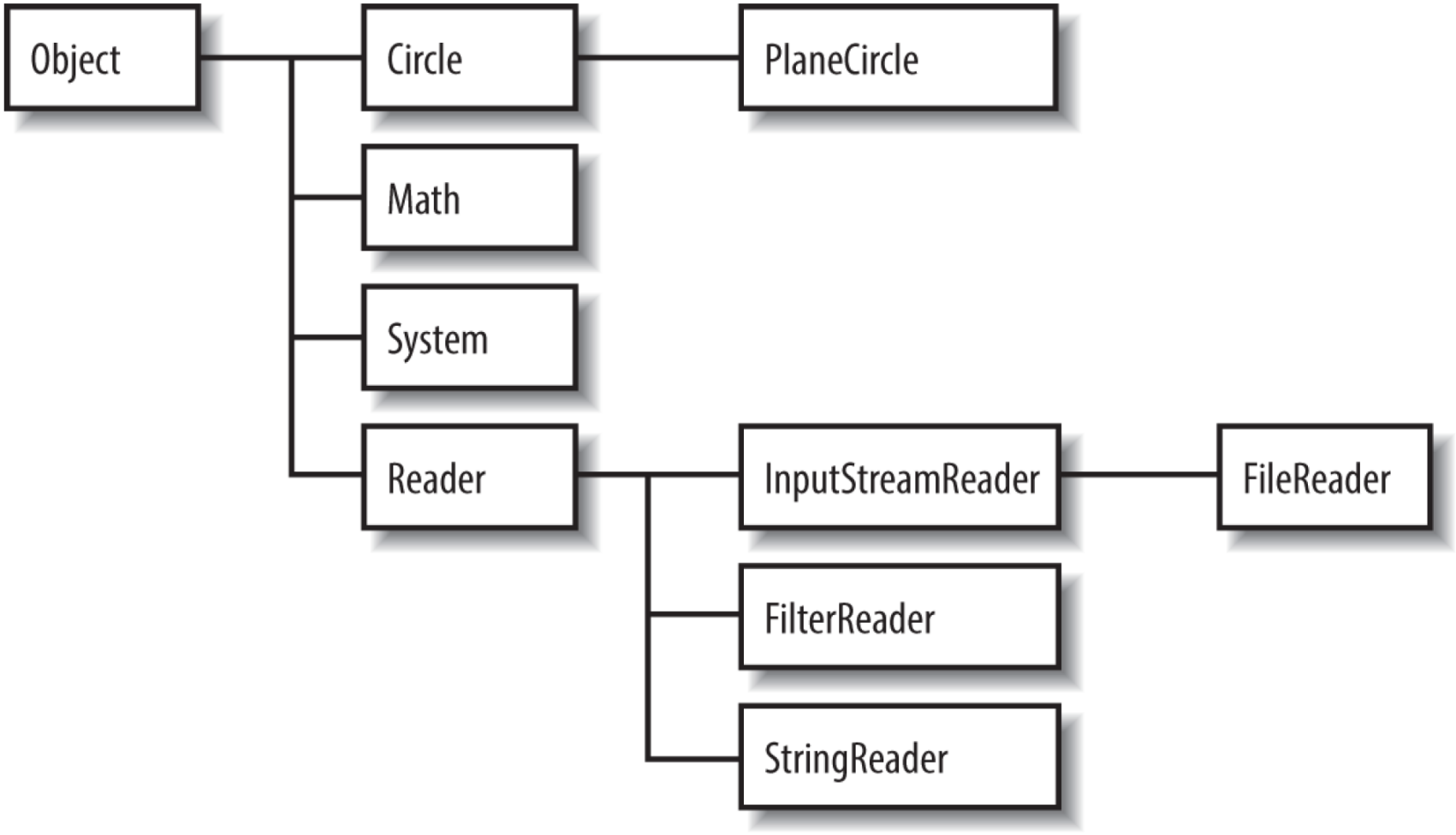
- You could add more kinds of animals by extending **Animal**!
- In this sense[12], we succeed to reuse the code.

[12] This is not the whole story. Stay tuned.

Constructor Chaining

- Once the constructor of the subclass is invoked, JVM will invoke the constructor of its superclass, recursively.
- So you might think that there will be a whole chain of constructors called, all the way back to the constructor of the class **Object**, the topmost class in Java.
- In this sense, we could say that every class is an immediate or a distant subclass of **Object**.

Illustration for Class Hierarchy[13]



[13] See Fig. 3-1 in p. 113 of Evans and Flanagan.

The super Operator

- Recall that this is used to refer to the object itself.
- You can use super to refer to (non-private) members of the superclass.
- Note that super() can be used to invoke the constructor of its superclass, just similar to this().

Method Overriding

- A subclass is supposed to re-implement the methods inherited from its superclass.[14]

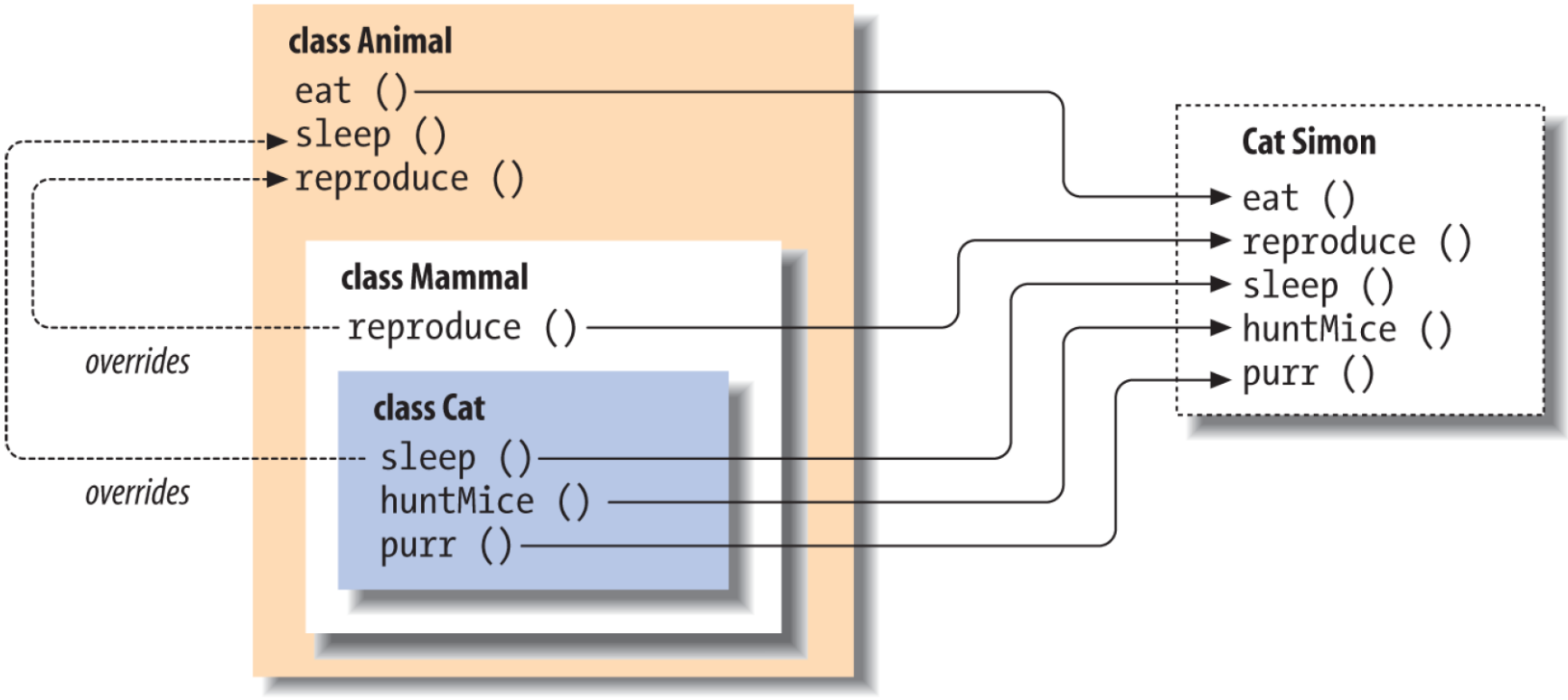
```
1 class B extends A {
2
3     @Override
4     void doAction() { /* New impl. w/o changing API. */ }
5
6 }
```

- Use @Override, which is one of annotations[15], to help check if the overriding works.
- Note that you cannot override the static methods.

[14] The overridden method has the signature identical to the parent one with the same return type. You cannot reduce its visibility, say from public to private.

[15] See https://docs.oracle.com/javase/tutorial/java/annotations/.

Conceptual Example



Example: Animals

```
1 public class Human extends Animal {
2     ...
3     @Override
4     public void eat() {
5         System.out.println("Eating with chopsticks...");
6     }
7     ...
8 }
```

```
1 public class Dog extends Animal {
2     ...
3     @Override
4     public void eat() {
5         System.out.println("Eating on the ground...");
6     }
7     ...
8 }
```

Example: Overriding toString()

- **Object** provides the method toString() which is deliberately designed to be invoked by **System**.out.println()!
- It returns a hashcode for this object as default.[16]
- Override this method to output a customized string.

```
1 public class Point {
2     ...
3     @Override
4     public String toString() {
5         return "(" + x + ", " + y + ")";
6     }
7     ...
8 }
```

[16] See https://en.wikipedia.org/wiki/Java_hashCode().

Example: List

```
1  import java.util.List;
2  import java.util.Arrays;
3
4  public class TestDemo {
5
6      public static void main(String[] args) {
7
8          List<String> lst = Arrays.asList("csie", "ntu", "tw");
9          System.out.println(lst); // Output [csie, ntu, tw].
10
11     }
12
13 }
```

- You may use **Arrays**.asList() to create a **List**[17] object.

[17] See https://docs.oracle.com/javase/8/docs/api/java/util/List.html.

Subtype Polymorphism[20]

- The term polymorphism literally means "many forms."
- One of OOP design rules is design by contract[18]: separate the interface from implementations and program to abstraction, not to implementation.[19]
- Subtype polymorphism fulfills this rule.
- How can a "single" interface be designed to accommodate different implementations?
  - Use the superclass of those types as the placeholder.

[18] Meyer (1986).

[19] See GoF (1994). The original statement is "program to interface, not to implementation."

[20] See also Java Polymorphism and its Types.

Example: Animals (Revisited)

```
1  public class AnimalDemo { // before decoupling
2
3      public static void goDinner(Human someone) { someone.eat(); }
4
5      public static void main(String[] args) {
6
7          Human arthur = new Human();
8          goDinner(arthur);
9          Dog lucky = new Dog();
10         goDinner(lucky); // Oops!
11
12     }
13
14 }
```

- You cannot pass a dog to the method goDinner(). (Why?)
- Instead, you need to write another method for dogs.
- How to decouple this dependency?

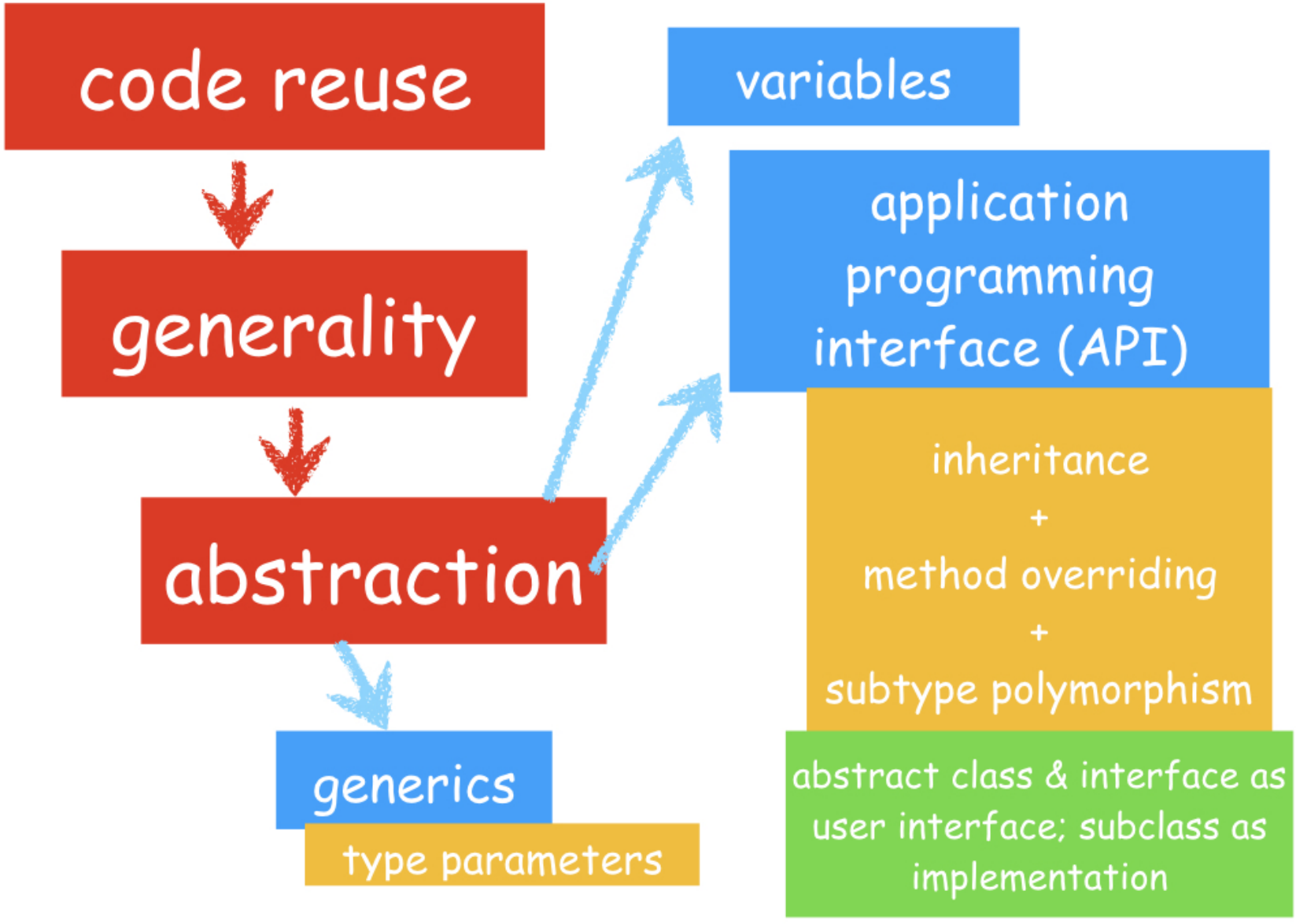
```
1  public class AnimalDemo { // after decoupling
2
3      public static void goDinner(Animal someone) { someone.eat(); }
4
5      public static void main(String[] args) {
6
7          Animal arthur = new Human();
8          goDinner(arthur);
9          Animal lucky = new Dog();
10         goDinner(lucky); // It works now!
11
12     }
13
14 }
```

- This example illustrates the analogy between the relationship of toString() and println().

Reflection: Big Picture of Why We Need OOP?[21]

- OOP is the solid foundation of modern (large-scale) software design.
- In particular, great reuse mechanism and abstraction are realized by these three concepts:
  - encapsulation isolates the internals (private members) from the externals, fulfilling the abstraction and providing the sufficient accessibility (public methods);
  - inheritance provides method overriding w/o changing method headers (return type + signatures);
  - polymorphism use superclass as a placeholder to manipulate the implementations (subtype objects).
- We use PIE as the shorthand for these three concepts.

[21] See https://en.wikipedia.org/wiki/Programming_paradigm


code reuse
generality
abstraction
generics
type parameters
variables
application programming interface (API)
inheritance + method overriding + subtype polymorphism
abstract class & interface as user interface; subclass as implementation

- This leads to the production of frameworks[22], which actually do most of the job, leaving the (application) programmer only with the job of customizing with business logic rules and providing hooks into it.
- This greatly reduces programming time and makes feasible the creation of larger and larger systems.
- In daily life, we often interact with objects at an abstract level.
  - We don't need to know the details to use them effectively, say using computers and cellphones, driving a car, and so on.

[22] See Spring Framework, especially Spring Boot for web applications. See also Android SDK for mobile applications.

Another Example

```
1  class Animal {
2      /* Ignore the previous part. */
3      void speak() { }
4  }
5
6  class Dog extends Animal {
7      @Override
8      void speak() { System.out.println("Woof! Woof!"); }
9  }
10
11 class Cat extends Animal {
12     @Override
13     void speak() { System.out.println("Meow~"); }
14 }
15
16 class Bird extends Animal {
17     @Override
18     void speak() { System.out.println("Tweet!"); }
19 }
```

```
1  public class PolymorphismDemo2 {
2
3      public static void main(String[] args) {
4
5          Animal[] animals = { new Dog(), new Cat(), new Bird() };
6
7          for (Animal animal: animals) {
8              animal.speak();
9          }
10
11     }
12
13 }
```

- Again, **Animal** is a placeholder for its three subclasses.

Liskov Substitution Principle[23]

- For convenience, let **U** be a subtype of **T**.
- We manipulate objects (right-hand side) via references (left-hand side)!
- Liskov states that **T**-type objects may be replaced with **U**-type objects without altering any of the desirable properties of **T** (correctness, task performed, etc.).

[23] See https://en.wikipedia.org/wiki/Liskov_substitution_principle.

Casting

- Upcasting[24] is to cast the **U** object/variable to the **T** variable.

```
1        U u1 = new U(); // Trivial.
2        T t1 = u1;      // OK.
3        T t2 = new U(); // OK.
```

- Downcasting[25] is to cast the **T** variable to a **U** variable.

```
1        U u2 = (U) t2;  // OK, but dangerous. Why?
2        U u3 = new T(); // Error! Why?
```

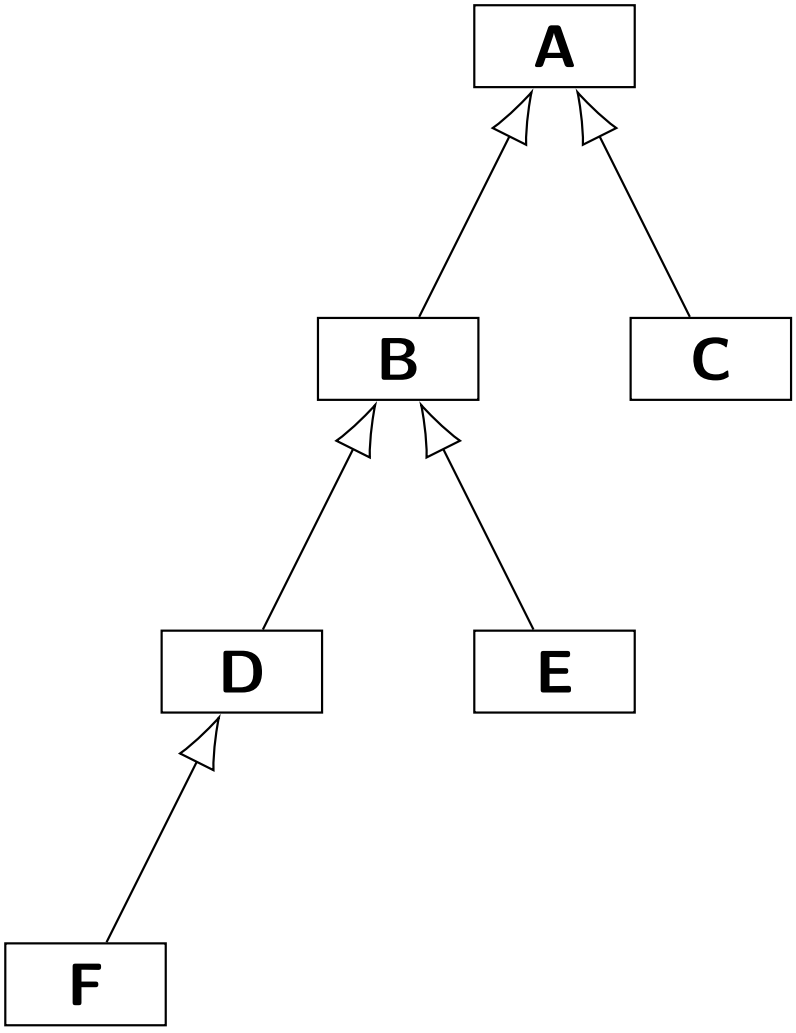
[24] Widening conversion; back compatibility.

[25] Narrow conversion; forward advance.

Solution: instanceof

- Upcasting is wanted and always allowed. (Why?)
- However, downcasting is not always possible even when you use cast operators.
  - In fact, type checking at compilation time becomes unsound if any cast operator is applied. (Why?)
  - **ClassCastException** is thrown for invalid casting or explicit conversion.
- In particular, a **T**-type variable, acting as a placeholder, can point to all siblings of **U**-type.
- We can use instanceof to check if the referenced object is compatible with the target type at runtime.

Example



- The class inheritance can be represented by a digraph (directed graph).
- For example, **D** is a subtype of **A** and **B**, which are both reachable from **D** on the digraph.

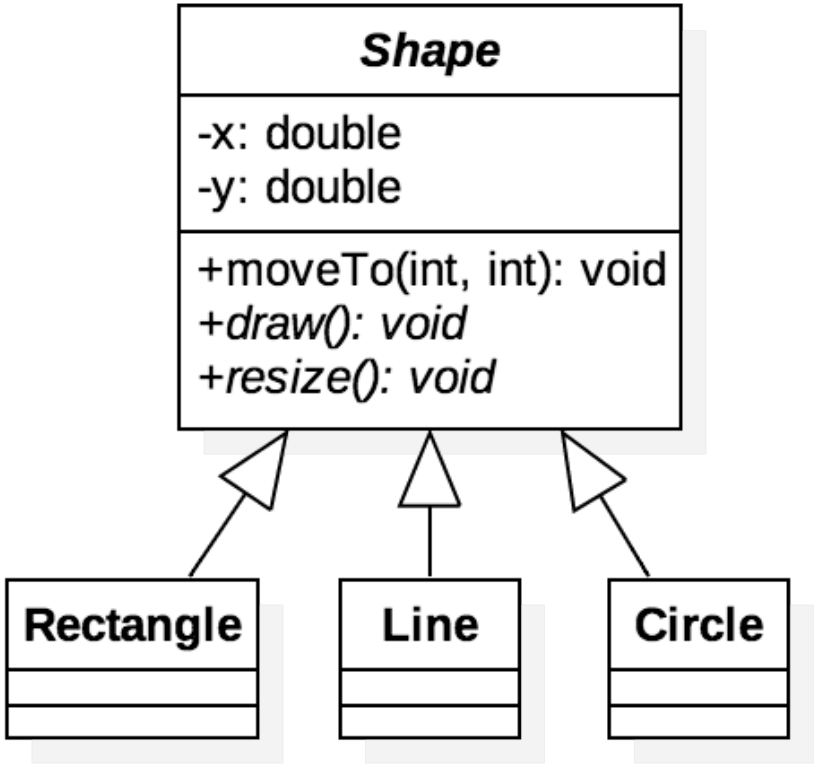
```java
class A { }
class B extends A { }
class C extends A { }
class D extends B { }
class E extends B { }
class F extends D { }

public class InstanceofDemo {

    public static void main(String[] args) {

        Object o = new D();

        System.out.println(o instanceof A); // Output true.
        System.out.println(o instanceof B); // Output true.
        System.out.println(o instanceof C); // Output false.
        System.out.println(o instanceof D); // Output true.
        System.out.println(o instanceof E); // Output false.
        System.out.println(o instanceof F); // Output false.

    }

}
```

Abstract Classes / Methods

- A method can be declared abstract without braces but ending with a semicolon.
- When one class has one or more abstract methods, the class itself must be declared abstract as well.[26]
- Typically, one abstract class sits at the top of one class hierarchy, acting as an placeholder.
- No abstract class can be instantiated directly. (Why not?)
- When inheriting an abstract class, Eclipse (or any IDE) could help you insert all abstract methods.

[26] You can also declare one abstract class which has no abstract method.

Example



- In UML, abstract methods and classes are in italic.
- The method *draw*() and *resize*() can be implemented when the specific shape is known.

The final Keyword[27]

- A final variable is a variable which can be initialized once and cannot be changed later.
  - The compiler makes sure that you can do it only once.
  - A final variable is often declared with static keyword and treated as a constant, for example, **Math**.PI.
- A final method is a method which cannot be overridden by subclasses.
  - Make a method final if its implementation should be preserved.
- A class that is declared final cannot be inherited.
  - For example, again, **Math**.

[27] In Java, the keyword const is reserved.

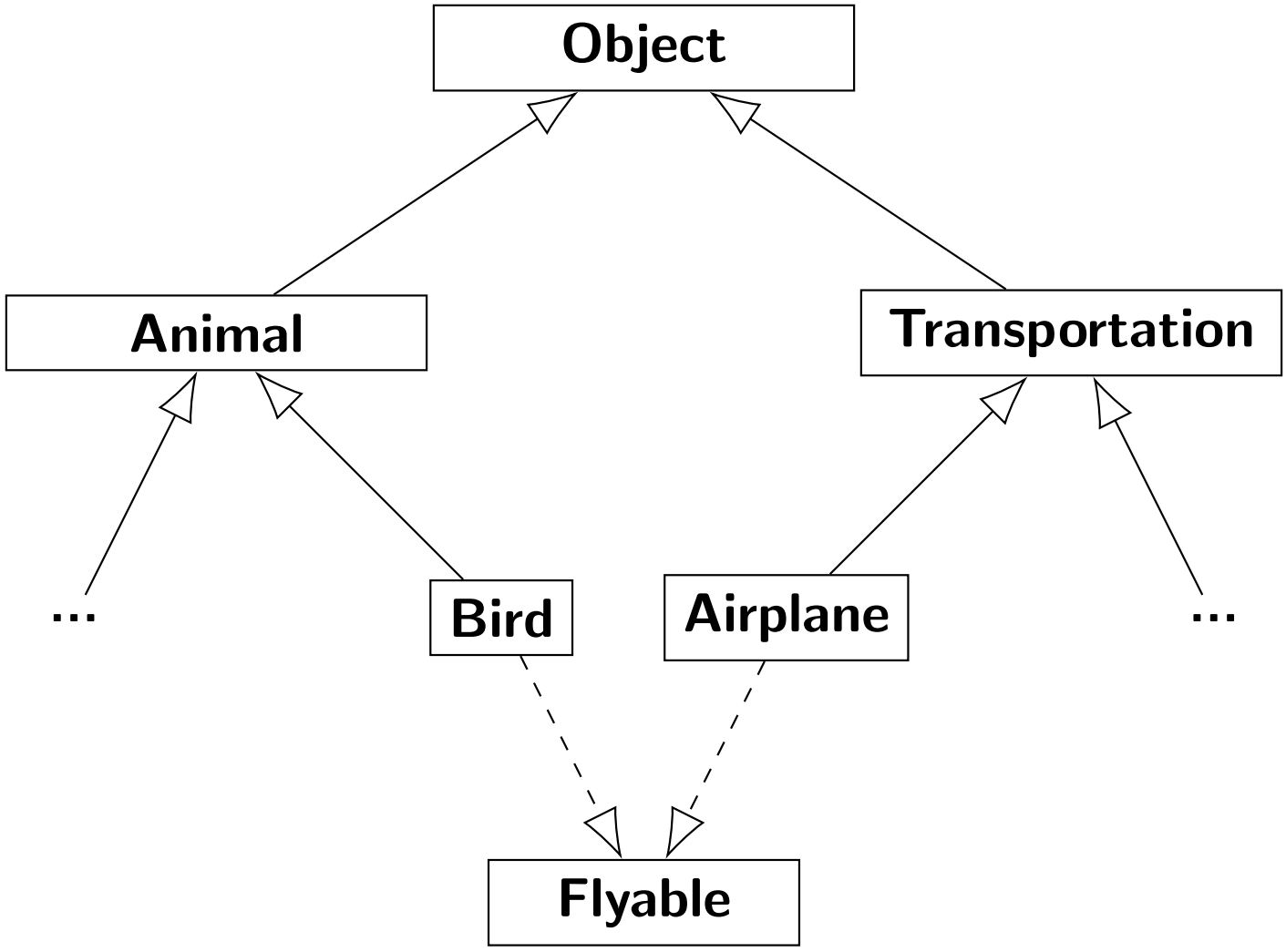
Second IS-A Relationship: Interface Inheritance

- Objects of different types often work together without a proper vertical relationship.[28]
- For example, consider **Bird** inherited from **Animal** and **Airplane** inherited from **Transportation**.
- Both **Bird** and **Airplane** are able to fly in the sky, say by calling the method fly().
- The Fly method should not be defined in each superclass. (Why?)

[28] Recall that Java allows single inheritance between classes.

- We want those flyable objects to go flying by calling one single, uniform API, just like the way of **Animal**.
- Recall that **Object** is the superclass of everything.
- So, how about using **Object** as the placeholder?
  - No good. (Why?)
- Clearly, we need an extra horizontal relationship: interface.

```
1 public interface Flyable {
2
3     void fly(); // Implicitly public and abstract.
4
5 }
```


Object
Animal
Transportation
...
Bird
Airplane
...
Flyable

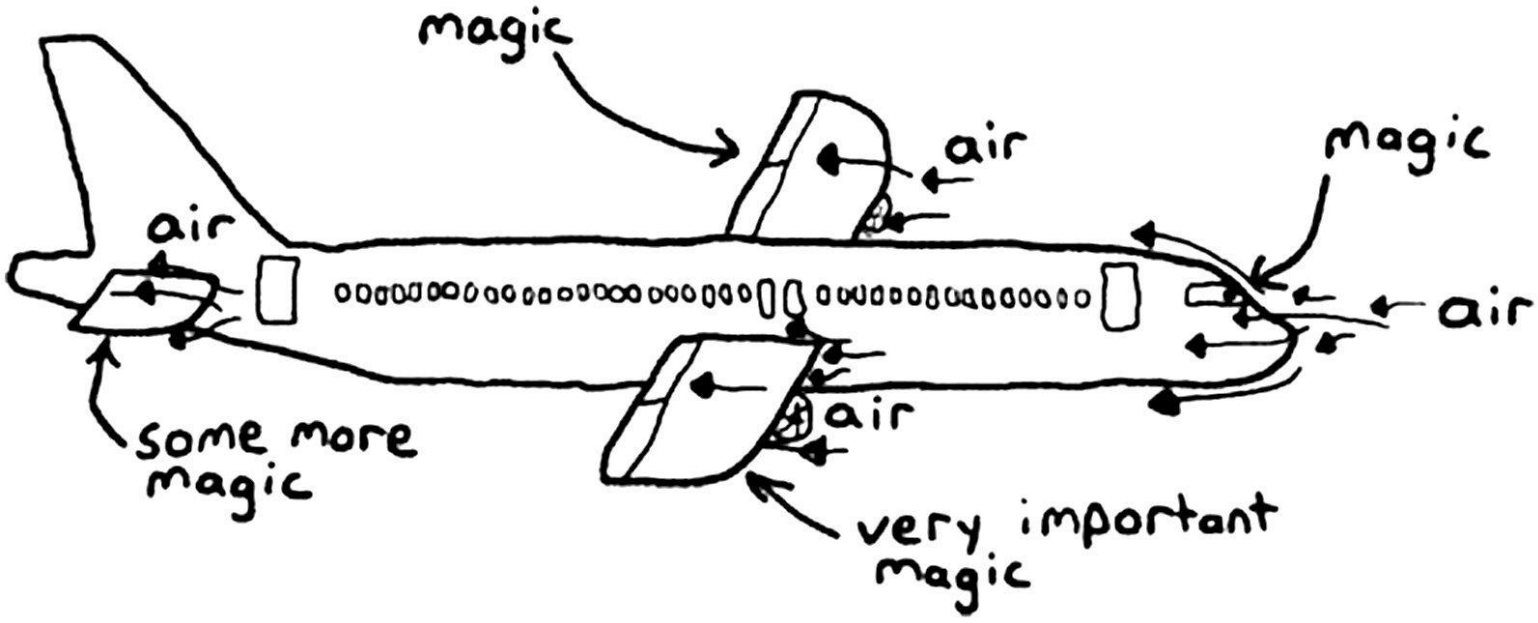
```java
public class Animal { }
```

```java
public class Bird extends Animal implements Flyable {

    public void flyByFlappingWings() {
        System.out.println("Flapping wings!");
    }

    @Override
    public void fly() { flyByFlappingWings(); }

}
```

```java
public class Transportation { }
```

```java
public class Airplane extends Transportation implements Flyable {

    public void flyByCastingMagic() {
        System.out.println("@#!^$%&#!$%@$");
    }

    @Override
    public void fly() { flyByCastingMagic(); }

}
```

how planes fly



https://i.imgur.com/y2bmNpz.jpg

```
1  public class InterfaceDemo {
2
3      public static void main(String[] args) {
4
5          Bird owl = new Bird();
6          goFly(owl);
7
8          Airplane a380 = new Airplane();
9          goFly(a380);
10
11     }
12
13     public static void goFly(Flyable flyableObj) {
14
15         flyableObj.fly();
16
17     }
18
19 }
```

- Again, a single interface allows multiple implementations!

A Deep Dive into Interfaces

- An interface defines behaviors for multiple types, acting like a contract between objects and clients.
- It could have abstract methods so that it cannot be instantiated (directly).
- Interfaces are also reference types, just like classes.
- Interfaces are stateless because they may not declare fields.
- A class can inherit multiple interfaces!
- Note that an interface can extend another interfaces, like a collection of contracts in some sense.

- We conventionally names interfaces using nouns and adjectives, often ending with "able."
  - For example, **Runnable**, **Callable**[29], **Serializable**[30], and **Comparable**[31].
- JDK8 introduces new features as follows:
  - Declare final static non-blank fields and methods;
  - Define default methods which are already implemented;
  - Use functional interfaces for lambda expressions (anonymous functions) which are widely used in the Stream framework.

[29] **Runnable** and **Callable** are related to Java multithreading.

[30] Used for an object which can be represented as a sequence of bytes. This is called object serialization.

[31] Use to define the ordering among objects. This is widely utilized in Java Collections, say sort and binary search.

Which to Use? Interfaces or Abstract Classes

- Use abstract classes when you want to:
  - share common code for a group of related classes, and
  - declare non-static members such as properties and methods.
- Use interfaces for any of situations as follows:
  - define a contract or a set of method signatures that classes must adhere to;
  - take advantage of multiple inheritance.

Exercise: RPG

MERCHANT

https://i.pinimg.com/originals/86/23/1e/86231eb0d17be0192765c38dd4afea89.jpg

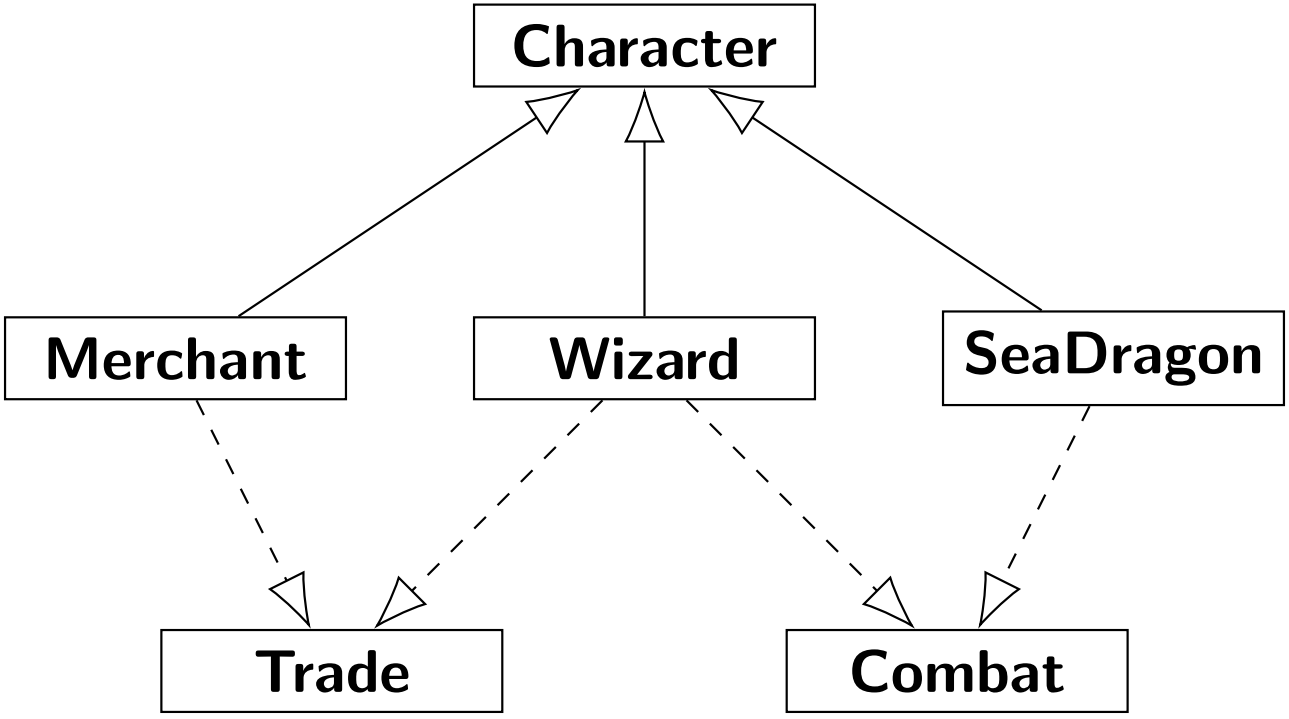
SEADRAGON

https://pm1.narvii.com/7075/794ae8b788ed9ae4c442e371ad5206911fbb2cber1-1950-1550v2_hq.jpg

WIZARD

https://s-media-cache-ak0.pinimg.com/originals/97/5b/81/975b8177e951f45c7f372de2dc08da6e.jpg

- First, **Wizard**, **SeaDragon**, and **Merchant** are three of **Character**s.
- In particular, **Wizard** fights with **SeaDragon** by invoking attack().
- **Wizard** buys and sells stuffs with **Merchant** by invoking buyAndSell().
- However, **SeaDragon** cannot buy and sell stuffs; **Merchant** cannot attack others.


Character
Merchant
Wizard
SeaDragon
Trade
Combat

```
1 abstract public class Character { }
```

```
1 public interface Combat {
2
3     void attack(Combat enemy);
4
5 }
```

```
1 public interface Trade {
2
3     void buyAndSell(Trade counterpart);
4
5 }
```

```java
public class Wizard extends Character implements Combat, Trade {

    @Override
    public void attack(Combat enemy) { }

    @Override
    public void buyAndSell(Trade counterpart) { }

}
```

```java
public class SeaDragon extends Character implements Combat {

    @Override
    public void attack(Combat enemy) { }

}
```

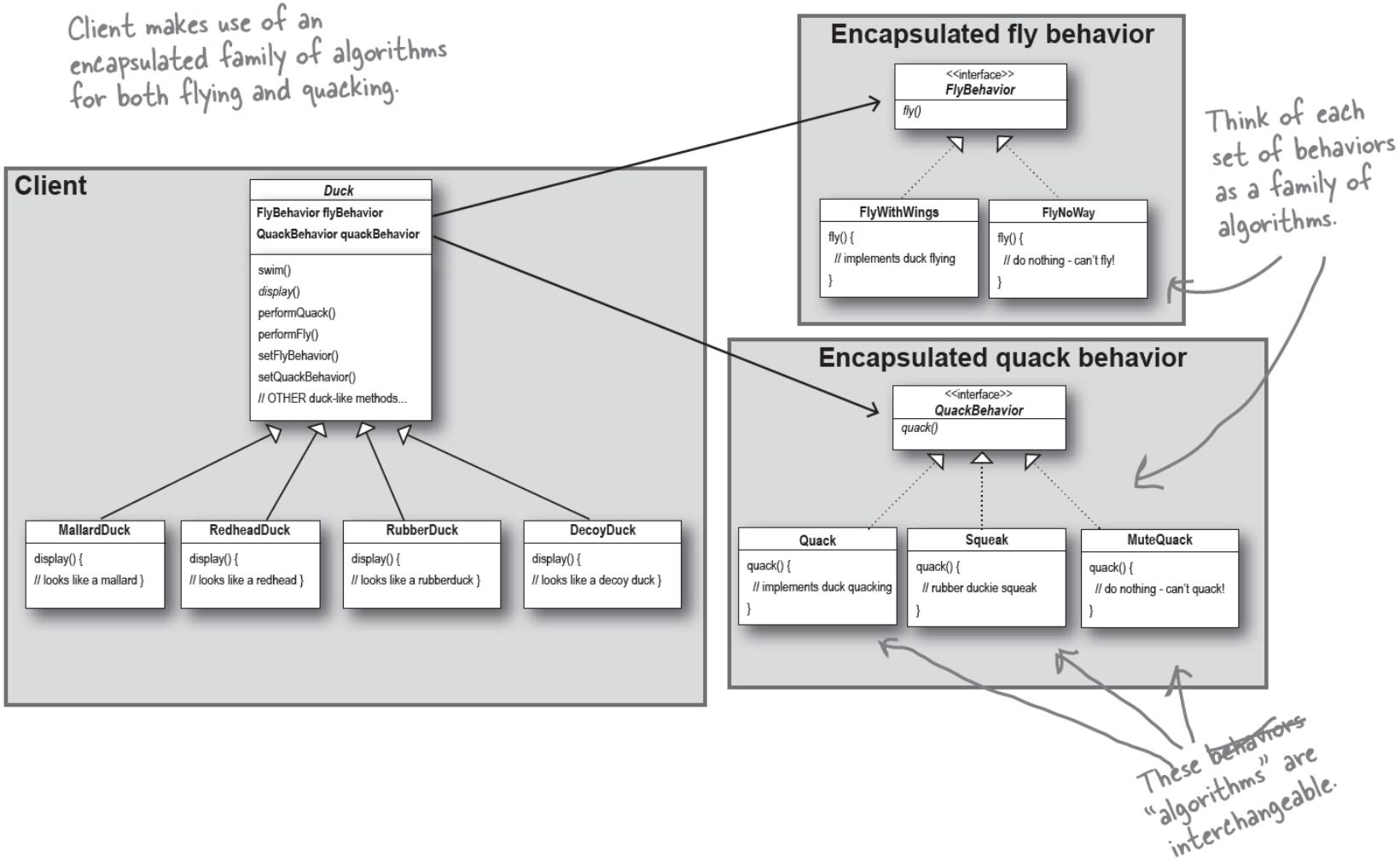
```java
public class Merchant extends Character implements Trade {

    @Override
    public void buyAndSell(Trade counterpart) { }

}
```

Delegation vs. Inheritance

- Class inheritance is a powerful way to achieve code reuse.
- However, class inheritance violates encapsulation!
- This is because a derived class depends on the implementation details of its base class for its proper function.
- To solve this issue, we favor delegation over inheritance.[32]

[32] See GoF (1995); See also Item 18 in Bloch (2018).

Example: Strategy Pattern[33]



[33] See Freeman and Robson (2020).

```
1  interface FlyBehavior { void fly(); }
2  interface QuackBehavior { void quack(); }
3
4  class FlyWithWings implements FlyBehavior
5  {
6      public void fly() { /* ... */ }
7  }
8  class CannotFly implements FlyBehavior
9  {
10     public void fly() { /* ... */ }
11 }
12 class Silence implements QuackBehavior
13 {
14     public void quack() { /* ... */ }
15 }
16 class SimpleQuack implements QuackBehavior
17 {
18     public void quack() { /* ... */ }
19 }
20 class Squeak implements QuackBehavior
21 {
22     public void quack() { /* ... */ }
23 }
```

```
1  class Duck {
2
3      private FlyBehavior flyBehavior;
4      private QuackBehavior quackBehavior;
5
6      public void setFlyBehavior(FlyBehavior flyBehavior) {
7          this.flyBehavior = flyBehavior;
8      }
9
10     public void setQuackBehavior(QuackBehavior quackBehavior) {
11         this.quackBehavior = quackBehavior;
12     }
13     public void performFly() {
14         flyBehavior.fly();
15     }
16     public void performQuack() {
17         quackBehavior.quack();
18     }
19 }
```

```java
class MalladDuck extends Duck { /* ... */ }
class RedHeadDuck extends Duck { /* ... */ }
class RubberDuck extends Duck { /* ... */ }
class DecoyDuck extends Duck { /* ... */ }

public class DuckDriver {

    public static void processDuck(Duck duck) {
        duck.performFly();
        duck.performQuack();
    }

    public static void main(String[] args) {
        Duck duck = new MalladDuck();
        duck.setFlyBehavior(new FlyWithWings());
        duck.setQuackBehavior(new SimpleQuack());
        processDuck(duck);

        duck.setFlyBehavior(new CannotFly()); // Injured duck.
        processDuck(duck);
    }
}
```

Special Issue: Wrapper Classes

Primitive	Wrapper
void	java.lang.Void
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

Autoboxing and Unboxing of Primitives

- The Java compiler automatically wraps the primitives in corresponding type, and unwraps them where appropriate.

```
1   ...
2   Integer i = 1; // Autoboxing.
3   Integer j = 2;
4   Integer k = i + 1; // Autounboxing and then autoboxing.
5
6   System.out.println(k); // Output 2.
7   System.out.println(k == j); // Output true.
8
9   Integer m = new Integer(i);
10  System.out.println(m == i); // Output false?
11  System.out.println(m.equals(i)); // Output true!?
12  ...
```

Immutable Objects

- An object is considered immutable if its state cannot change after it is constructed.
- Often used for value objects.
- Imagine that there is a pool for immutable objects.
- After the value object is first created, this value object is reused if needed.
- This implies that another object is created when we operate on the immutable object.
  - Another example is **String** objects.[34]
- Using immutable objects is a good practice when it comes to concurrent programming.[35]

[34] For you information, **StringBuffer** is the mutable version of **String** objects.
[35] See http://www.javapractices.com/topic/TopicAction.do?Id=29.

```
1         ...
2         String str1 = "NTU";
3         String str2 = "ntu";
4
5         System.out.println("str1 = " + str1.toLowerCase());
6         System.out.println("str1 = " + str1);
7
8         str1 = str1.toLowerCase();
9         System.out.println("str1 = " + str1);
10        System.out.println(str1 == str2); // False?!
11        System.out.println(str1.equals(str2)); // True!
12        System.out.println(str1.intern() == str2); // True!!
13        ...
```

- You can use equals() to check if the text is identical to the other.
- You may use intern() to check the **String** pool containing the **String** object whose text is identical to the other.[36]

[36] See the Interning Pattern in GoF (1995).

Special Issue: Enumeration

- An enum type is a special type for a set of predefined options.
- You can use a static method values() to enumerate all options.
- This mechanism enhances type safety and makes the source code more readable!

Example: Colors

```
1  public enum Color {
2
3      RED, BLUE, GREEN;
4
5      public static Color random() {
6
7          Color[] colors = values();
8          return colors[(int) (Math.random() * colors.length)];
9
10     }
11
12 }
```

- **Color** is indeed a subclass of **Enum** with three final and static references to **Color** objects corresponding to the enumerated values.
- We could also equip the enum type with static methods.

```
1  public class EnumDemo {
2
3      public static void main(String[] args) {
4
5          Color crayon_color = Color.RED;
6          Color tshirt_color = Color.random();
7          System.out.println(crayon_color == tshirt_color);
8
9      }
10
11 }
```

Exercise

```
1  public enum PowerState {
2
3      ON("The power is on."),
4      OFF("The power is off."),
5      SUSPEND("The power is low.");
6
7      private String status;
8      private PowerState(String msg) { status = msg; }
9
10     @Override
11     public String toString() { return msg; }
12
13 }
```

```
1  public class PowerMachine {
2
3      private PowerState state = PowerState.OFF;
4
5      public PowerState getState() {
6          return state;
7      }
8
9      public void turnOn() {
10         state = PowerState.ON;
11     }
12
13     public void turnOff() {
14         state = PowerState.OFF;
15     }
16
17     public void sleep() {
18         state = PowerState.SUSPEND;
19     }
20
21 }
```

```
1  public class PowerMachineDemo {
2
3      public static void main(String[] args) {
4
5          PowerMachine p = new PowerMachine();
6          System.out.println(p.getState());
7          p.turnOn();
8          System.out.println(p.getState());
9          p.sleep();
10         System.out.println(p.getState());
11         p.turnOff();
12
13     }
14
15 }
```

- Try to illustrate the memory allocation of this program.

Discussion: What behind enum?

```java
public enum Action { PLAY, WORK, SLEEP, EAT }
```

```java
public class Action {

    public final static Action PLAY = new Action("PLAY");
    public final static Action WORK = new Action("WORK");
    public final static Action SLEEP = new Action("SLEEP");
    public final static Action EAT = new Action("EAT");

    public static Action[] values() {
        return new Action[] { PLAY, WORK, SLEEP, EAT };
    }

    private final String text;
    private Action(String str) { text = str; }

    // Some functionalities are not listed explicitly.
    // Check java.lang.Enum.

}
```

Special Issue: Packages, Imports, and Access Control

- The first statement, other than comments, in a Java source file, must be a package declaration, if there exists.
- A package is a grouping of related types providing access protection (shown below) and namespace management.

Scope \ Modifier	private	(package)	protected	public
Within the class	✓	✓	✓	✓
Within the package	x	✓	✓	✓
Inherited classes	x	x	✓	✓
Out of package	x	x	x	✓

Example

```
1  package www.csie.ntu.edu.tw;
2
3  public class Util {
4
5      void doAction1() { }
6      public void doAction2() { }
7      protected void doAction3() { }
8      public static void doAction4() { }
9
10 }
```

- Use package to indicate the package the class belongs to.
- The package is implemented by folders.

```java
import www.csie.ntu.edu.tw.Greeting;

public class ImportDemo {

    public static void main(String[] args) {

        Util util = new Util();
        util.doAction1(); // Error!
        util.doAction2(); // OK!
        util.doAction3(); // Error!!
        Util.doAction4(); // OK!!

    }

}
```

- As you can see, doAction1() is not visible. (Why?)
- Note that protected members are visible under inheritance, even if separated in different packages.

Example: More about Imports

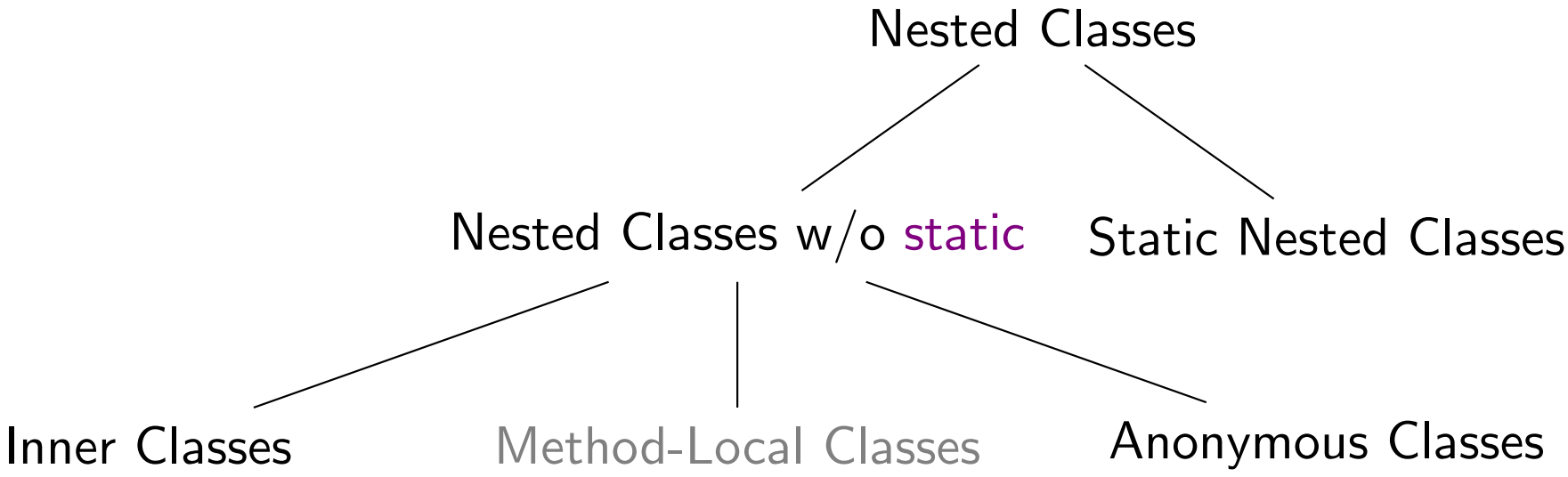
```
1  import www.csie.ntu.edu.tw.*; // Import all classes.
2  import static www.csie.ntu.edu.tw.Util.doAction4;
3
4  public class GreetingDemo {
5
6      public static void main(String[] args) {
7
8          Util util = new Util();
9          util.doAction2(); // ok!
10         Util.doAction4(); // ok!!
11
12         doAction4(); // No need to indicate the class name.
13     }
14
15 }
```

- Use the wildcard (*) to import all classes within the package.
- We could also import static members in the package only.

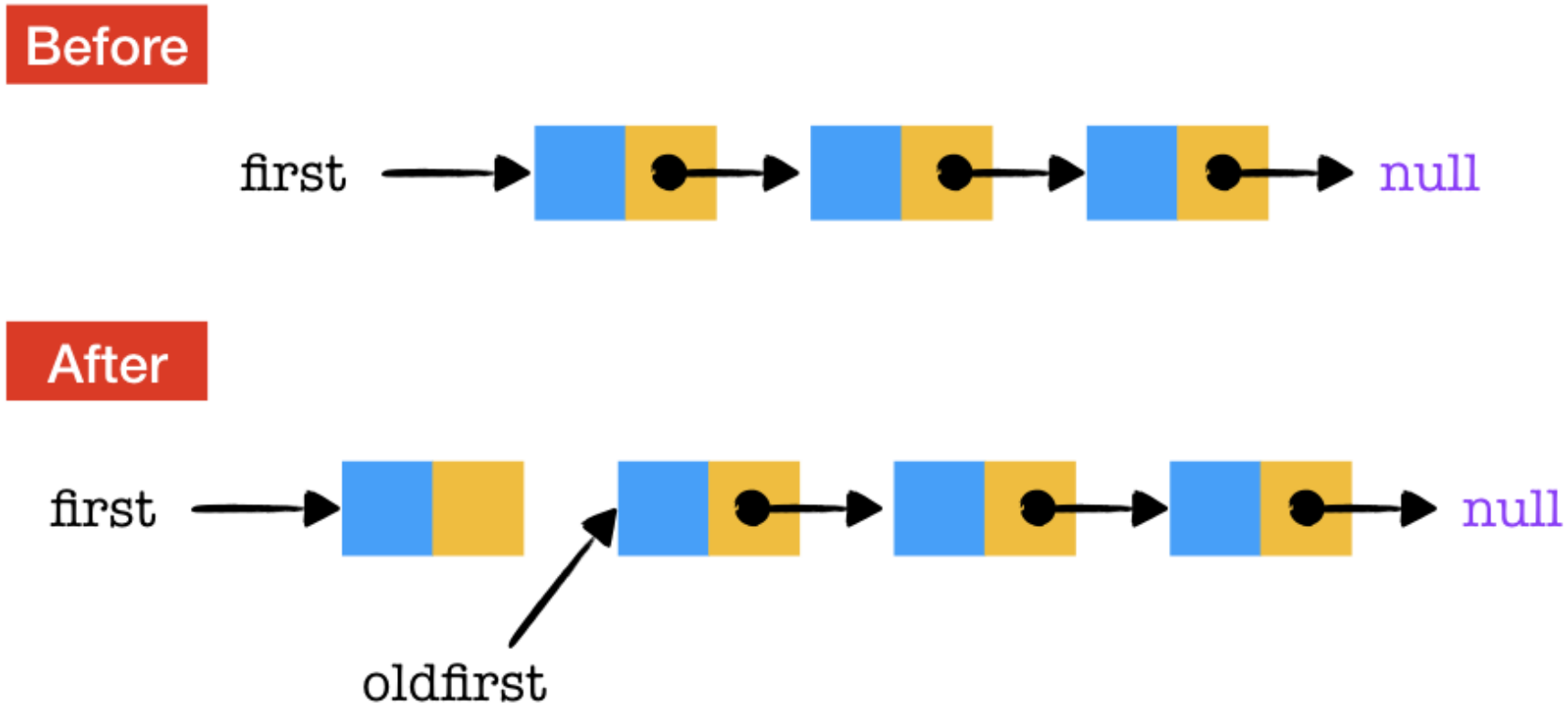
Special Issue: Nested Classes

- A nested class is a member of its enclosing class.
- Nesting classes increases encapsulation and also leads to more readable and maintainable code.
- Especially, it is a good practice to seal classes which are only used in one place.

Family of Nested Classes



Example: Stack by Linked List



```java
public class LinkedListStack {

    private Node first; // Trait of linked list!

    private class Node {
        String item;
        Node next;
    }

    public String pop() {
        String item = first.item;
        first = first.next; // Deja vu?
        return item;
    }

    public void push(String item) {
        oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

}
```

```java
public class LinkedListStackDemo {

    public static void main(String[] args) {

        LinkedListStack langs = new LinkedListStack();
        langs.push("Java");
        langs.push("C++");
        langs.push("Python");

        System.out.println(langs.pop()); // Output Python.
        System.out.println(langs.pop()); // Output C++.
        System.out.println(langs.pop()); // Output Java.

    }

}
```

- Note that the method push() and pop() run in $O(1)$ time!
- The output shows the FILO (first-in last-out) property of stack.

Exercise: House & Rooms

```java
import java.util.ArrayList;

public class House {

    private ArrayList<Room> rooms = new ArrayList<>();

    private class Room {
        String name;
        @Override
        public String toString() { return name; }
    }

    public void add(String name) {
        Room room = new Room();
        room.name = name;
        rooms.add(room);
    }

    @Override
    public String toString() { return rooms.toString(); }

}
```

```java
public class HouseDemo {

    public static void main(String[] args) {

        House home = new House();
        home.add("Living room");
        home.add("Bedroom");
        home.add("Bathroom");
        home.add("Kitchen");
        home.add("Storeroom");

        System.out.println(home);

    }

}
```

Anonymous Class

- Anonymous classes enable you to declare and instantiate the class at the same time.
- They are like inner classes except that they don't have a name.
- Use anonymous class if you need only one instance of the inner class.

Example: Button

```
1  abstract class Button {
2
3      abstract void onClicked();
4
5  }
```

```
1  public class AnonymousClassDemo1 {
2
3      public static void main(String[] args) {
4
5          Button btnOK = new Button() {
6              @Override
7              public void onClicked() { System.out.println("OK"); }
8          };
9
10         btnOK.onClicked();
11
12     }
13
14 }
```

Exercise: Fly Again

```
1  public class AnonymousClassDemo2 {
2
3      public static void main(String[] args) {
4
5          Flyable butterfly = new Flyable() {
6              @Override
7              public void fly() { /* ... */ }
8          };
9
10         butterfly.fly();
11
12     }
13
14 }
```

- We can instantiate objects for one interface by using anonymous classes.

Special Issue: Iterator Patterns

- An iterator is the standard interface to enumerate elements of the data structure in the for-each loop:
  - One class implementing the interface **Iterable** should provide the detail of the method iterator().
  - The iterator() method should produce an iterator defined by the interface **Iterator**, which has two unimplemented methods: hasNext() and next().
- For example, you has a box containing 3 strings (shown next page) and make it iterable.
- Then the box could be iterated in the for-each loop!

Example

```
1  import java.util.Iterator;
2
3  class Box implements Iterable<String> {
4
5      String[] items = { "Java", "C++", "Python" };
6
7      public Iterator<String> iterator() {
8
9          return new Iterator<String>() {
10             private int ptr = 0;
11             public boolean hasNext() {
12                 return ptr < items.length;
13             }
14             public String next() {
15                 return items[ptr++];
16             }
17         }; // anonymous class
18
19     }
20 }
```

```
1  public class IteratorDemo {
2
3      public static void main(String[] args) {
4
5          Box books = new Box();
6
7          // for-each loop
8          /*
9          for (String book : books) {
10             System.out.println(book);
11         }
12         */
13
14         Iterator iter = books.iterator();
15         while (iter.hasNext())
16             System.out.println(iter.next());
17
18     }
19
20 }
```

Static Nested Class

- A static nested class is an enclosed class declared static.
- Note that only nested class can be static.
- As a static member, it can access to other static members without instantiating the enclosing class.
- In particular, a static nested class can be instantiated directly, without instantiating the enclosing class object first; it acts like a minipackage.

Example

```
1  public class StaticClassDemo {
2
3      public static class Greeting {
4
5          @Override
6          public String toString() {
7              return "This is a static class.";
8          }
9
10     }
11
12     public static void main(String[] args) {
13
14         System.out.println(new StaticClassDemo.Greeting());
15
16     }
17
18 }
```