Java Programming 2

Quick Review on Object-Oriented Programming (OOP)

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

```java
class Lecture7 {

        // Object-Oriented Programming (OOP)

}

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

Object & Class

- An object keeps its own states in fields (attributes) and exposes its behaviors through associated methods.
- To create these objects, we collect all attributes associated with functions and put them in a new class.
- A class is the blueprint to create instances, aka runtime objects.
- A class acts as a derived type.
- Classes are the building blocks in Java.

Example: Points

- We define a new class as follows:
  - give a class name with the first letter capitalized, by convention;
  - declare data and function members in the class body.

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

- Now we use this class to create some points.

```
1  public class PointDemo {
2
3      public static void main(String[] args) {
4
5          Point p1 = new Point();
6          p1.x = 1;
7          p1.y = 2;
8
9          Point p2 = new Point();
10         p2.x = 3;
11         p2.y = 4;
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, we hide internal states and expose methods which perform actions on these fields.
- So all fields should be declared private.
- However, this private modifier does not guarantee any information security.[1]
  - What private is good for maintainability and modularity.[2]

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

[2] Read `http://stackoverflow.com/questions/9201603/are-private-members-really-more-secure-in-java`.

Function Members

- As said, the fields are hidden.
- So we provide getters and setters for an object, if necessary:
  - getter: return the state of the object.
  - setter: set a value to the state of the object.
- For example, getX() and getY() are getters; setX() and setY() are setters in the class **Point**.

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

- A constructor follows the new operator, acting like other methods.
- However, its name should be identical to the name of the class and it has no return type.
- A class may have several constructors if needed.
  - Recall method overloading.
- Note that constructors belong to the class but not objects.
  - In other words, constructors cannot be invoked by any object.
- If you don't define any explicit constructor, Java assumes a default constructor for you.
  - Moreover, adding any explicit constructor disables the default constructor.

Parameterized Constructors

- You can initialize an object when the object is ready.
- For 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 }
```

Self Reference

- You can refer to any (instance) member of the current object within methods and constructors by using this.
- The most common reason for using the this keyword is because a field is shadowed by method parameters.
  - Recall the variable scope.
- You can also use this to call another constructor in the same class, say this().

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

- Be aware that data members and function members are declared w/o static in this lecture.
- They are called instance members, which are 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 when a specific **Point** object is specified.

Example: Distance Measurement Between Points

```
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 }
```

- In OOP design, it is important to clarify the responsibility among objects of various types, aka single responsibility principle.[3]
  - High cohesion, low coupling.
  - The Hollywood principle: don't call us, we'll call you.

[3] Also see https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

Static Members

- Static members are ready once a class is loaded.
  - For example, main().
  - You may try static initialization blocks.[4]
- These members can be invoked directly by class name in absence of any instance.
  - For example, **Math**.PI.
- In particular, static methods perform algorithms.
  - For example, **Math**.random() and **Arrays**.sort().
- Note that a static method can access other static members. (Trivial.)
- However, static methods cannot access to instance members directly. (Why?)

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

Example

```java
public class Point {

    /* Ignore the previous part. */

    public static double measure(Point first, Point second) {
        // You cannot use this in static context.
        return Math.sqrt(Math.pow(first.x - second.x, 2)
                         + Math.pow(first.y - second.y, 2));

    }

}
```

```java
public class PointDemo {

    public static void main(String[] args) {

        /* Ignore the previous part. */
        System.out.println(Point.measure(p1, p2));

    }
}
```

Another Example: Singleton Pattern

- The singleton pattern is one of design patterns.[5]
- 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 }
```

[5] Design patterns are a collection of highly-reusable solutions to a commonly occurring problem within a given context in software design. The term "design pattern" is named by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, often referred to as the Gang of Four (GoF).

Object Elimination: Garbage Collection (GC)[6]

- Java handles object deallocation by GC.
  - Timing: preset period or when memory stress occurs.
- GC is a daemon thread, which searches for those unreferenced objects.
  - An object is unreferenced when it is no longer referenced by any part of your program. (How?)
  - To make the object unreferenced, simply assign null to the reference variable.
- Note that you may invoke **System**.gc() to execute a deallocation procedure.
  - However, frequent invocation of GC is time-consuming.

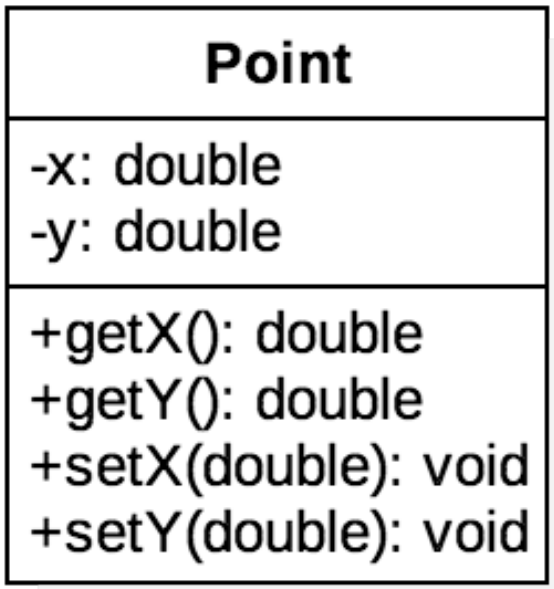
[6] `http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html`

Unified Modeling Language[7]

- Unified Modeling Language (UML) is a tool for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.
- Free software:
  - `http://staruml.io/` (available for all platforms)

[7] See `http://www.tutorialspoint.com/uml/` and `http://www.mitchellsoftwareengineering.com/IntroToUML.pdf`.

Example: Class Diagram for Point

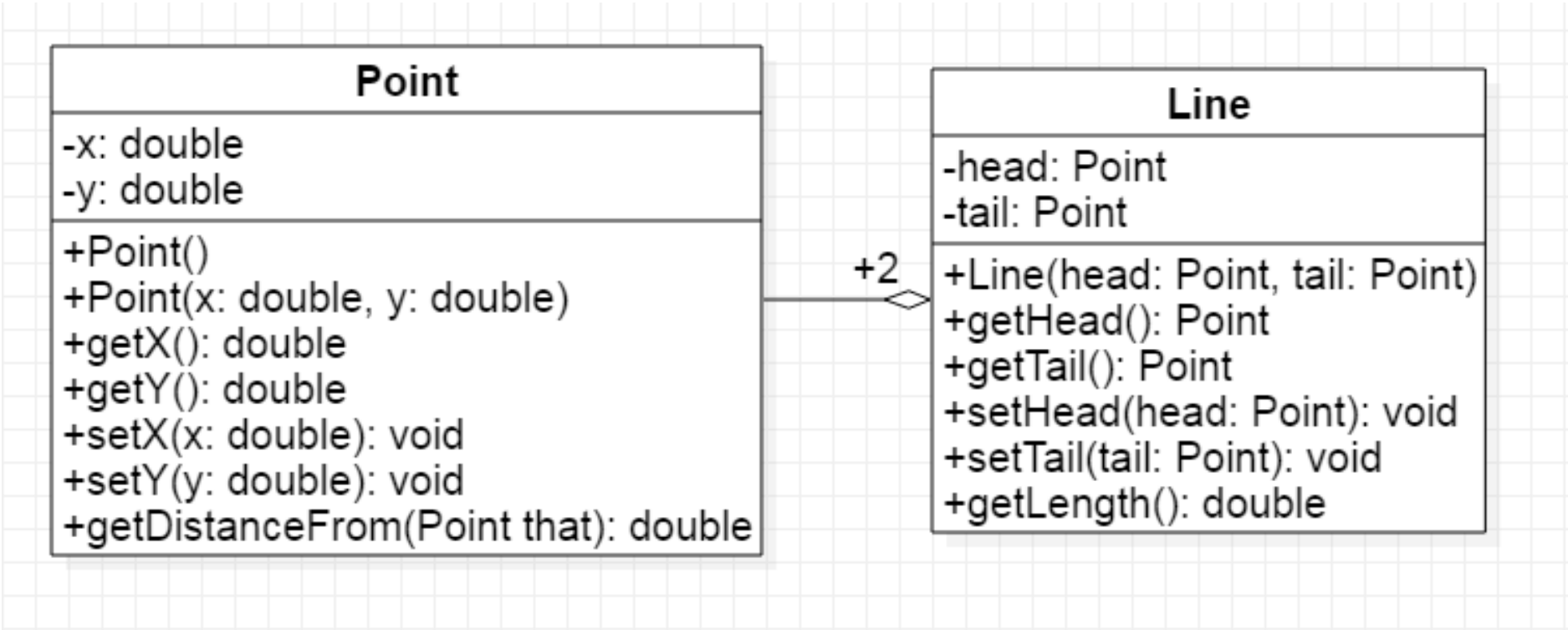


- $+$ refers to public.
- $-$ refers to private.

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)



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

```java
public class Line {

    private Point head, tail;

    public Line(Point p1, Point p2) {
        head = p1;
        tail = p2;
    }

    /* Ignore some methods. */

    public double getLength() {
        return head.getDistanceFrom(tail);
    }

}
```

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

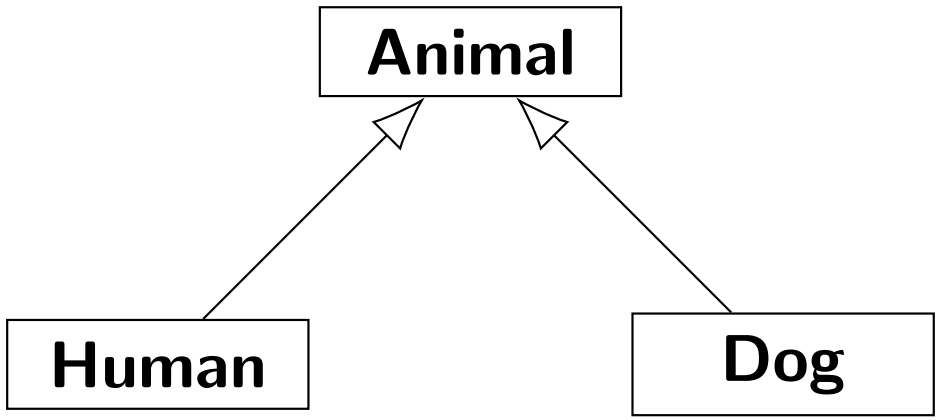
Photo credit: `https://www.sunnyskyz.com/uploads/2016/12/nlf37-dog.jpg`

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 wag() {}

}
```

After Using Inheritance



- Move the common part 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() {}

}
```

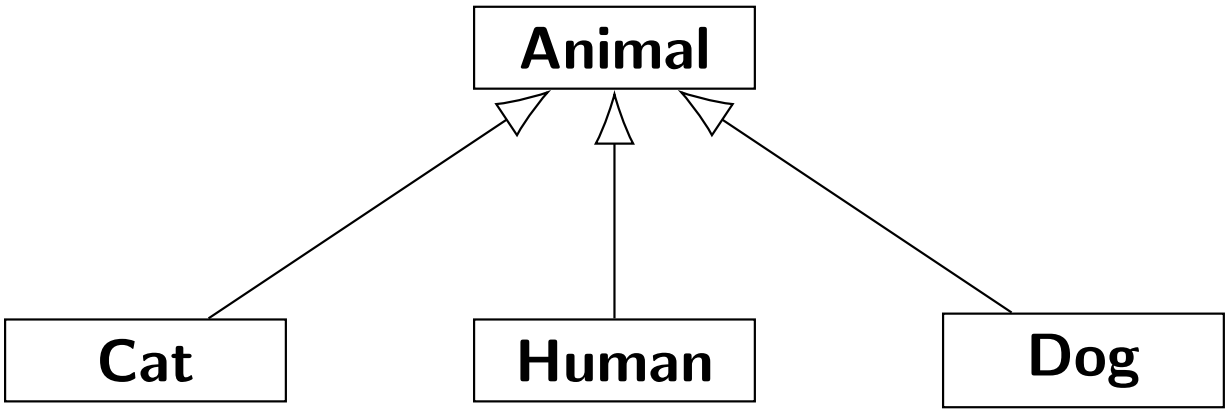
```java
public class Human extends Animal {

    public void writeCode() {}

}
```

```java
public class Dog extends Animal {

    public void wag() {}

}
```

Exercise: Add **Cat** to Animal Hierarchy[8]

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

[8] See https://petsmao.nownews.com/20170124-10587.



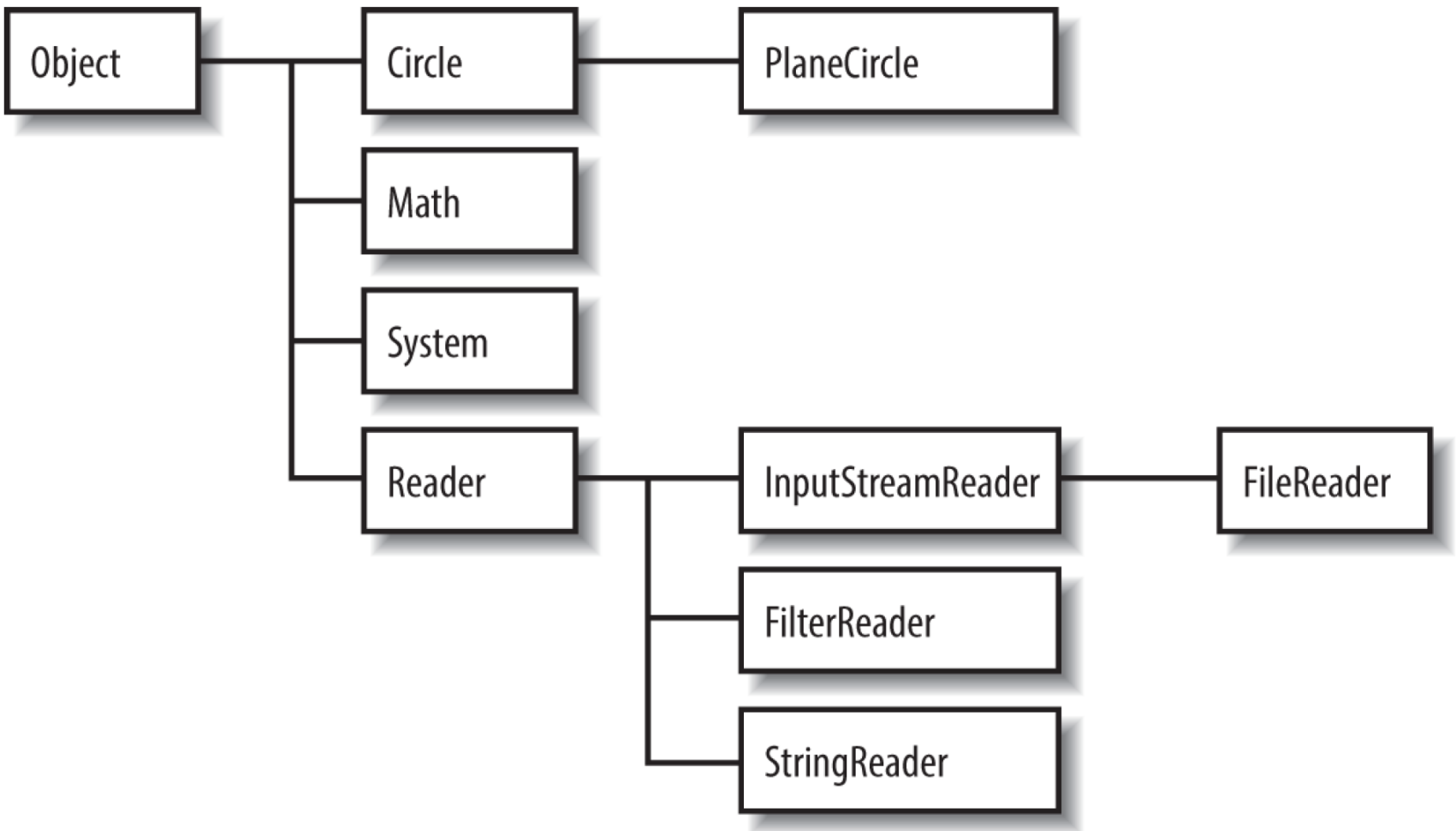

```
1 public class Cat extends Animal {
2
3     public void stepping() {}
4
5 }
```

- You could add more kinds of animals by extending **Animal**!
- Again, code reuse.

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[9]



[9] 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

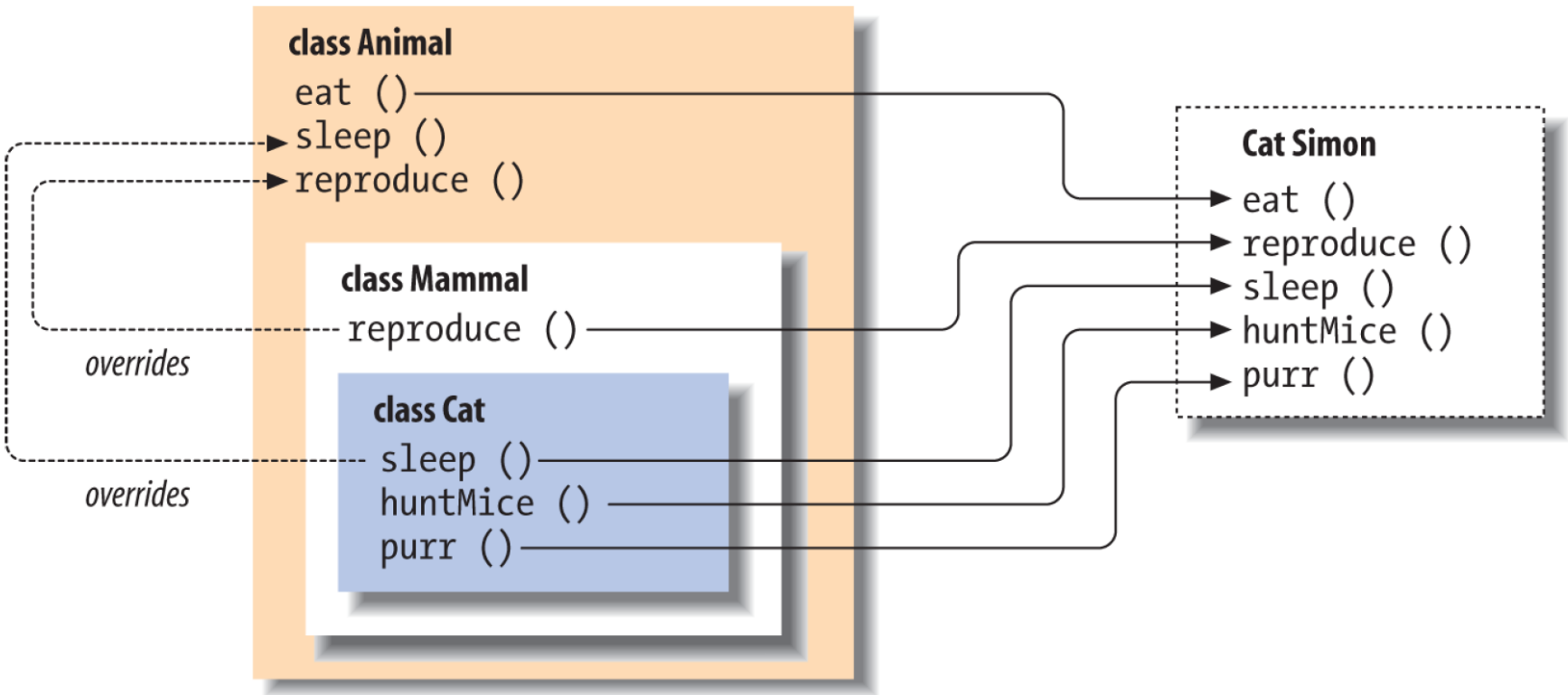
- A subclass is supposed to re-implement the methods inherited from its superclass.
- The requirement of method overriding is as follows:
  - method signature identical to the one of its superclass;
  - same return type;
  - non-reduced visibility relative to the one of its superclass.[10]
- Note that you cannot override the static methods.
- You should use the annotation[11] @Override to help you.

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

[10] For example, you cannot reduce the visibility from public to private.

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

Example


class Animal
eat ()
sleep ()
reproduce ()
class Mammal
reproduce ()
class Cat
sleep ()
huntMice ()
purr ()
overrides
overrides
Cat Simon
eat ()
reproduce ()
sleep ()
huntMice ()
purr ()

Example: Overriding toString()

- **Object** provides the method toString() which is deliberately designed to be invoked by **System**.out.println()!
- By default, it returns a hash code.[12]
- It could be overridden so that it returns an informative string.

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

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

Another Example: ArrayList (Revisited)

```
1  import java.util.Arrays;
2  import java.util.ArrayList;
3
4  public class ArrayListDemo2 {
5
6      public static void main(String[] args) {
7
8          String[] fx1 = {"TWD", "CAD", "JPY"};
9          ArrayList<String> fx2 =
10                             new ArrayList<>(Arrays.asList(fx1));
11         System.out.println(fx2); // Output [TWD, CAD, JPY].
12
13     }
14
15 }
```

- Use **Arrays**.asList() to convert arrays to **ArrayList** objects.
- Much better!!!

Subtype Polymorphism[14]

- The word polymorphism literally means "many forms."
- One of OOP design rules is to separate the interface from implementations and program to abstraction, not to implementation.[13]
- Subtype polymorphism fulfills this rule.
- How to make a "single" interface for different implementations?
  - Use the superclass of those types as the placeholder.

[13] GoF (1995). The original statement is "program to interface, not to implementation."

[14] Also read `http://www.javaworld.com/article/3033445/learn-java/java-101-polymorphism-in-java.html`.

Example: Dependency Reduction (Decoupling)

```
1  class HighSchoolStudent {
2
3      void doHomework() {}
4
5  }
6
7  class CollegeStudent {
8
9      void writeFinalReports() {}
10
11 }
```

- Now let these two kinds of students go study.

```java
public class PolymorphismDemo {

    public static void main(String[] args) {

        HighSchoolStudent Emma = new HighSchoolStudent();
        goStudy(Emma);

        CollegeStudent Richard = new CollegeStudent();
        goStudy(Richard);

    }

    public static void goStudy(HighSchoolStudent student) {
        student.doHomework();
    }

    public static void goStudy(CollegeStudent student) {
        student.writeFinalReports();
    }

    // What if the 3rd kind of students comes into the system?

}
```

Using Inheritance & Subtype Polymorphism

```
1  class Student {
2      void doMyJob() { /* Do not know the detail yet. */}
3  }
4
5  class HighSchoolStudent extends Student {
6
7      void doHomework() {}
8      @Override
9      void doMyJob() { doHomework(); }
10
11 }
12
13 class CollegeStudent extends Student {
14
15     void writeFinalReports() {}
16     @Override
17     void doMyJob() { writeFinalReports(); }
18
19 }
```

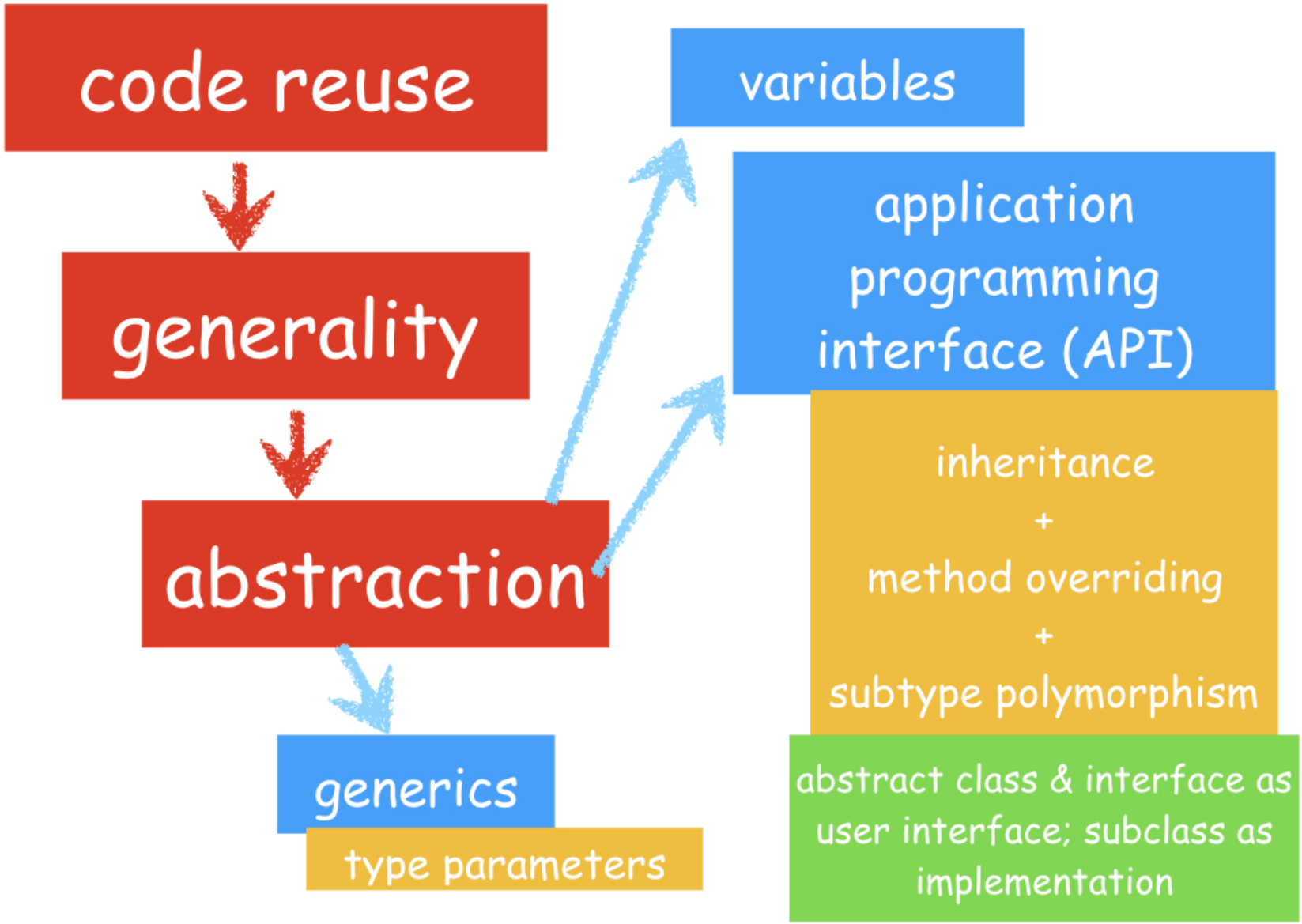
```java
public class PolymorphismDemo {

    public static void main(String[] args) {

        Student Emma = new HighSchoolStudent();
        goStudy(Emma);

        Student Richard = new CollegeStudent();
        goStudy(Richard);

    }

    // We can handle all kinds of students in this way!!!
    public static void goStudy(Student student) {
        student.doMyJob();
    }

}
```

- This example illustrates the mechanism between toString() and println().

Why OOP?[15]

- 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 the method signature;
  - polymorphism exploits the superclass as a placeholder to manipulate the implementations (subtype objects).
- We use PIE as the shorthand for these three concepts.

[15] 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[16], 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 analog, we often manipulate objects in an abstract level; we don't need to know the details when we use them.
  - For example, using computers and cellphones, driving a car, and so on.

[16] See `https://spring.io/`.

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 subtypes.

Liskov Substitution Principle[17]

- 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.).

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

Casting

- Upcasting[18] 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[19] 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?
```

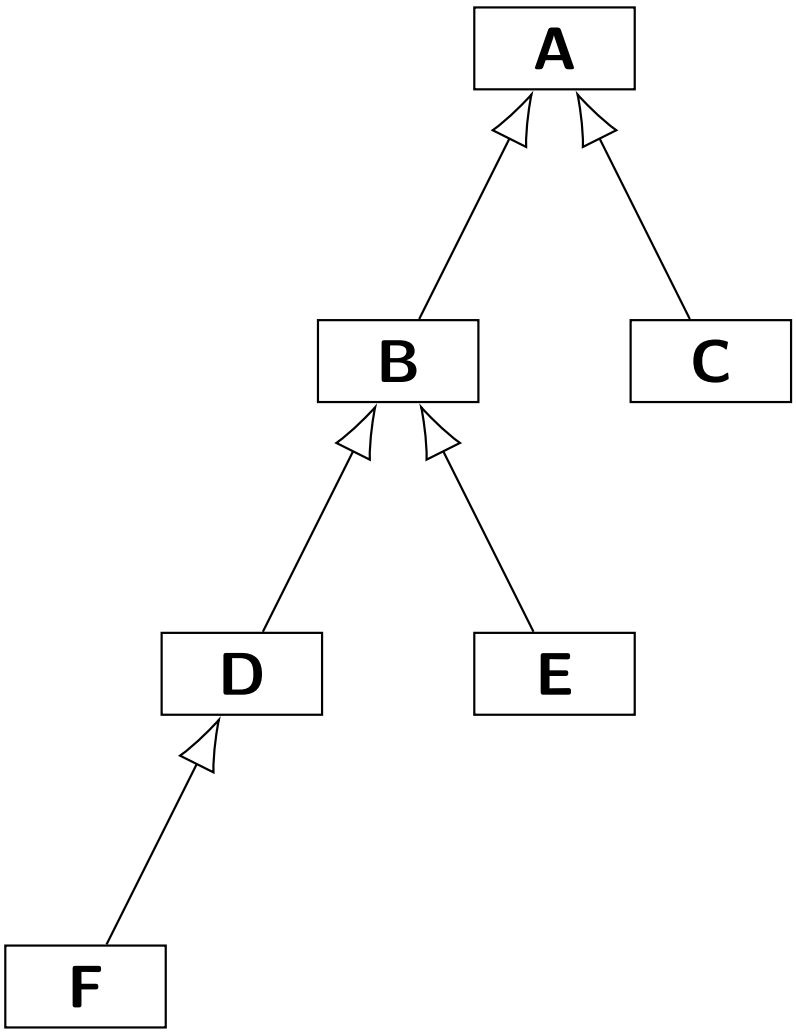
[18] A widening conversion; back compatibility.

[19] A narrow conversion; forward advance.

Solution: instanceof

- Upcasting is wanted and always allowed. (Why?)
- However, downcasting is not always true even when you use cast operators.
  - In fact, type checking at compile time becomes unsound if any cast operator is used. (Why?)
- Even worse, a **T**-type variable can point to all siblings of **U**-type.
  - Recall that a **T**-type variable works as a placeholder.
- Run-time type information (RTTI) is needed to resolve the error: **ClassCastException**.
- We can use instanceof to check if the referenced object is of 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.

    }

}
```

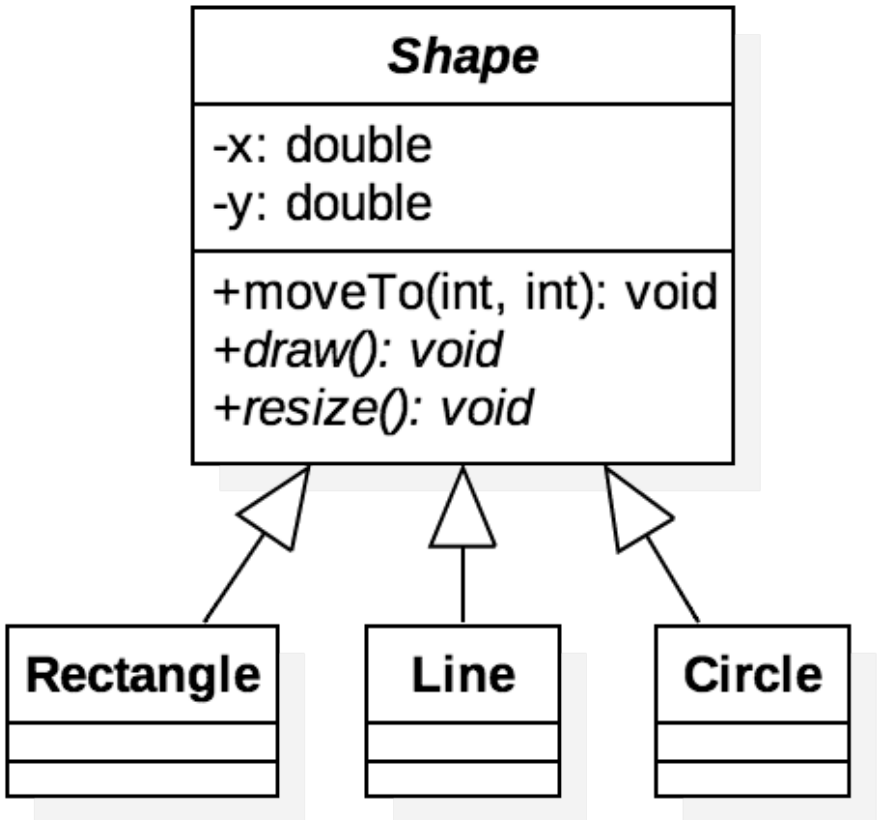
Abstract Classes

- An abstract class is a class declared abstract.
- Typically, abstract classes sit at the top of one class hierarchy, acting as placeholders.[20]
- The abstract classes may have some methods without implementation[21] and declared abstract.
  - They are abstract methods.
  - If a class has one or more abstract methods, then the class itself must be declared abstract.
- All abstract classes cannot be instantiated.
- When inheriting an abstract class, the editor could help you recall every abstract methods.

[20] For example, abstract factory pattern.

[21] The methods are declared without braces, and followed by a semicolon.

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[22]

- 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.
  - You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object.
- A class that is declared final cannot be inherited.
  - For example, again, **Math**.

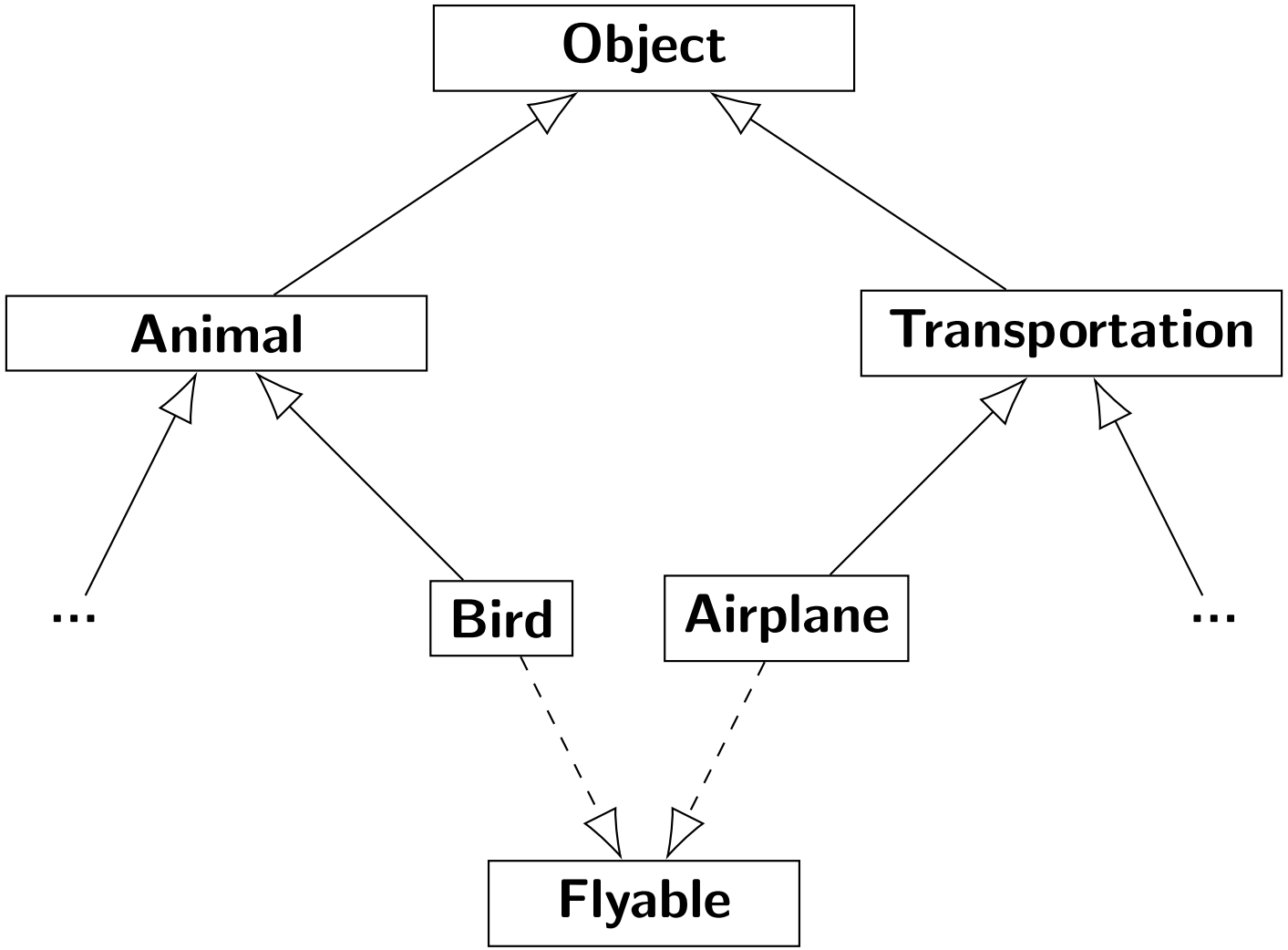
[22] In Java, the keyword const is reserved.

Another IS-A Relationship: Interface Inheritance

- Objects of different types are supposed to work together without a proper vertical relationship.
- 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().
- In semantics, the method fly() could not be defined in their superclasses. (Why?)

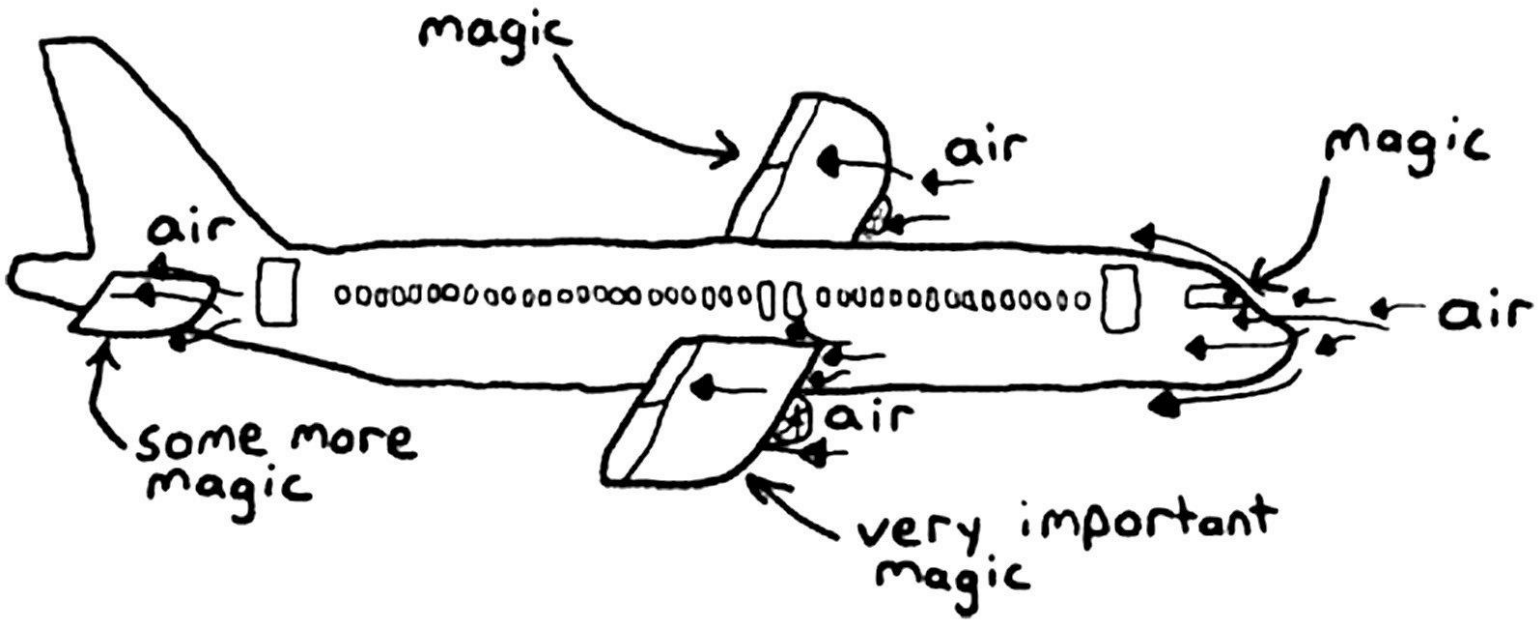
- We wish those flyable objects go flying by calling one API, just like the way of **Student**.
- Recall that **Object** is the superclass of everything.
- So, how about using **Object** as the placeholder?
  - Not really. (Why?)
- Clearly, we need a horizontal relationship: interface.

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


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

```
1  class Animal {}
2  class Bird extends Animal implements Flyable {
3
4      void flyByFlappingWings() {
5      System.out.println("Flapping wings!");
6      }
7
8      @Override
9      public void fly() { flyByFlappingWings(); }
10
11 }
12
13 class Transportation {}
14 class Airplane extends Transportation implements Flyable {
15
16     void flyByCastingMagic() {
17     System.out.println("#$%@$^@!#$!");
18     }
19
20     @Override
21     public void fly() { flyByCastingMagic(); }
22
23 }
```

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 uniform interface with multiple implementations!

A Deep Dive on Interfaces

- An interface is a contract between the object and the client.
- As shown, an interface is a reference type, just like classes.
- Unlike classes, interfaces are used to define methods without implementation so that they cannot be instantiated (directly).
- Also, interfaces are stateless.
- A class could implement multiple interfaces by providing method bodies for each predefined signature.

- Note that an interface can extend another interfaces!
  - Like a collection of contracts, in some sense.
- For example, **Runnable**, **Callable**[23], **Serializable**[24], and **Comparable**.
- In JDK8, we have new features as follows:
  - we can declare final static non-blank fields and methods;
  - we can also define default methods which are already implemented;
  - Java defines functional interfaces for lambdas which are widely used in the Stream framework. (Stay tuned in Java Programming 2!)

[23] Both are related to Java multithreading.

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

Timing for Interfaces & Abstract Classes

- Consider using abstract classes if you want to:
  - share code among several closely related classes, and
  - declare non-static or non-final fields.
- Consider using interfaces for any of situations as follows:
  - unrelated classes would implement your interface;
  - specify the behavior of a particular data type, but not concerned about who implements its behavior;
  - take advantage of multiple inheritance.

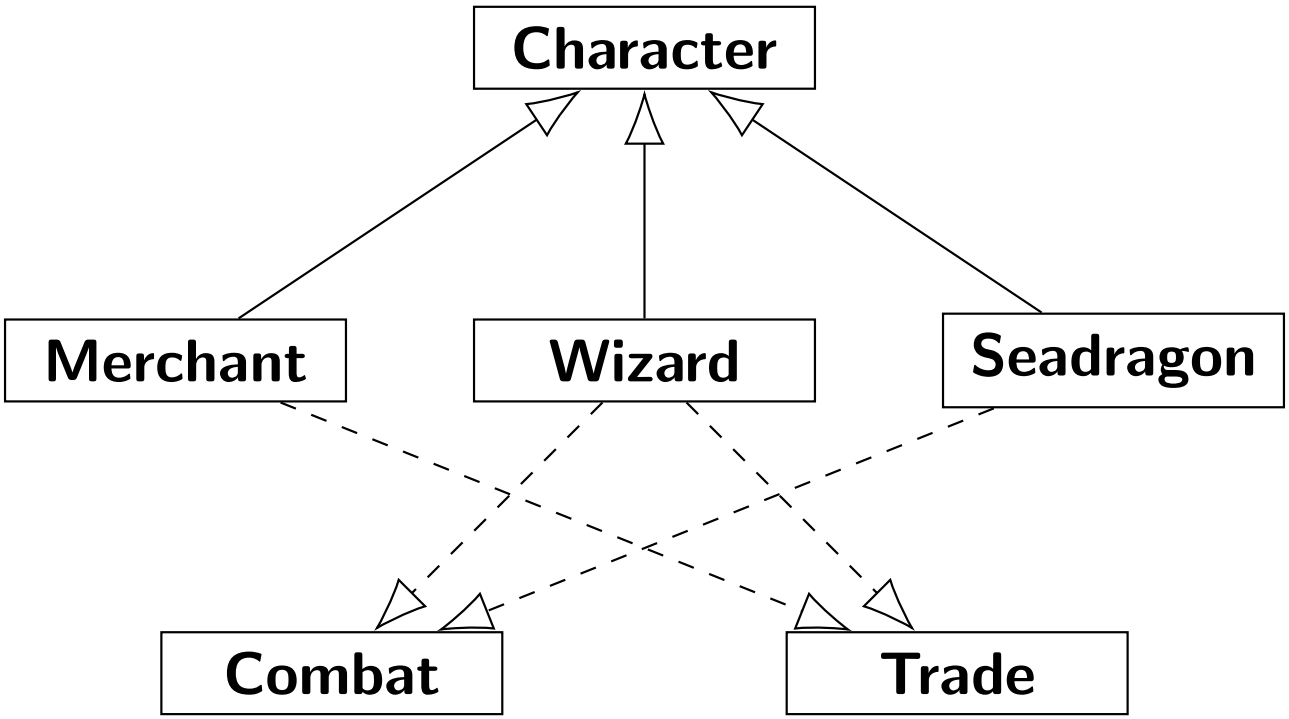
Exercise: RPG

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

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

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
Combat
Trade

```java
abstract public class Character {}
```

```java
public interface Combat {
    void attack(Combat enemy);
}
```

```java
public interface Trade {
    void buyAndSell(Trade counterpart);
}
```

```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) {}
}
```

HAS-A (Delegation) vs. IS-A (Inheritance)

- Class inheritance is a powerful way to achieve code reuse.
- However, class inheritance violates encapsulation!
- This is because a subclass depends on the implementation details of its superclass for its proper function.
- To solve this issue, we favor delegation over inheritance.[25]

[25] GoF (1995); Also see Item 18 in Bloch (2018).

Example: Strategy Pattern

- This pattern defines a family of algorithms by encapsulating each one, and making them interchangeable.
- It involves the following OO design principles:
  - encapsulate what varies;
  - code to an interface;
  - use delegation.

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.[26]
- Using immutable objects is a good practice when it comes to concurrent programming.[27]

[26] For you information, **StringBuffer** is the mutable version of **String** objects.

[27] 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.[28]

[28] 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 class PowerMachine {
2
3      private PowerState state;
4
5      public void setState(PowerState state) {
6          this.state = state;
7      }
8
9      public PowerState getState() { return state; }
10
11 }
12
13 enum PowerState {
14
15     ON("The power is on."), OFF("The power is off."),
16     SUSPEND("The power is low.");
17
18     private String status;
19     private PowerState(String str) { status = str; }
20
21 }
```

Behind enum?

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

```
1  public class Action {
2
3      public final static Action PLAY = new Action("PLAY");
4      public final static Action WORK = new Action("WORK");
5      public final static Action SLEEP = new Action("SLEEP");
6      public final static Action EAT = new Action("EAT");
7
8      private final String text;
9
10     public static Action[] values() {
11         return new Action[] {PLAY, WORK, SLEEP, EAT};
12     }
13
14     private Action(String str) { text = str;}
15
16     // Some functionalities are not listed explicitly.
17     // Check java.lang.Enum.
18
19 }
```

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

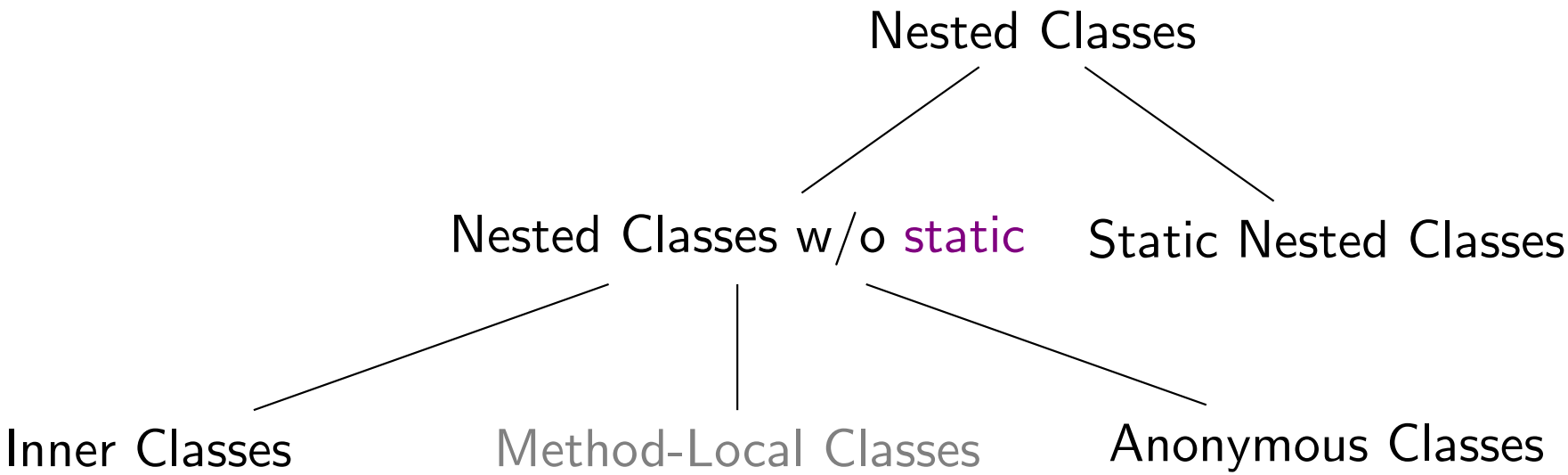
```
1  import www.csie.ntu.edu.tw.*; // Import all classes.
2  import static www.csie.ntu.edu.tw.Util.doAction4;
3
4  public class GreetingDemo {
5
6      public static void main(String[] args) {
7
8          Util util = new Util();
9          util.doAction2(); // ok!
10         Util.doAction4(); // 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.

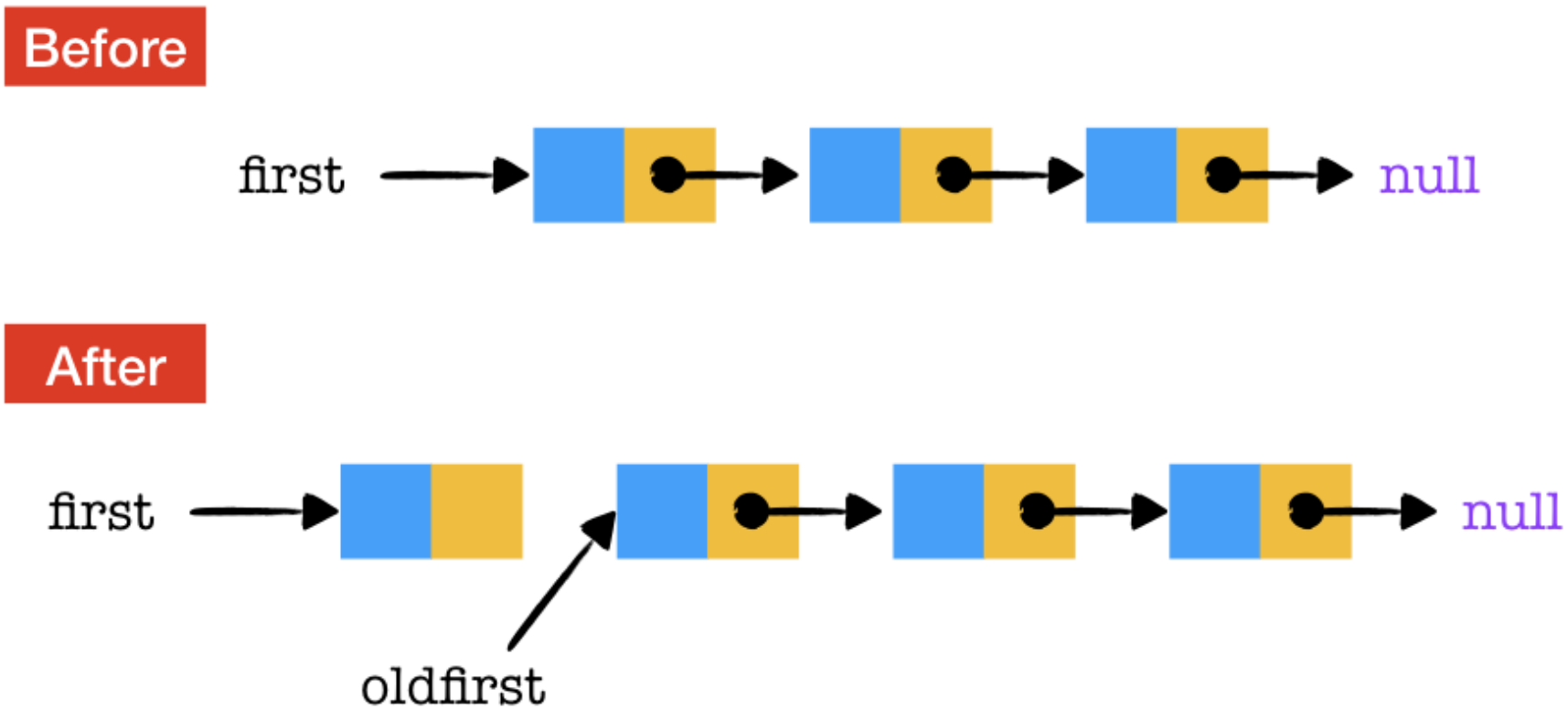
Special Issue: Nested Classes

- 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      abstract void onClicked();
3  }
4
5  public class AnonymousClassDemo1 {
6
7      public static void main(String[] args) {
8
9          Button btnOK = new Button() {
10             @Override
11             public void onClicked() {
12                 System.out.println("OK");
13             }
14         };
15
16         btnOK.onClicked();
17     }
18 }
```

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 }
```

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

Special Issue: Iterator Patterns

- An iterator is a simple and standard interface to enumerate elements in the data structure.
- In Java, we now proceed to reveal the mechanism of for-each loops:
  - One class implementing the interface **Iterable** should provide the detail of the method iterator().
  - The method iterator() should return an iterator defined by the interface **Iterator**, which has two unimplemented methods: hasNext() and next().
- Now your data structure could be compatible with for-each loops!

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() { return ptr < items.length; }
12             public String next() { return items[ptr++]; }
13         };
14
15     }
16 }
```

```java
public class IteratorDemo {
    public static void main(String[] args) {

        Box books = new Box();

        // for-each loop
        /*
        for (String book: books) {
            System.out.println(book);
        }
        */

        Iterator iter = books.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

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         System.out.println(new StaticClassDemo.Greeting());
14     }
15
16 }
```