Java Programming 2

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Java2 297
Spring 2018

```java
class Lecture7 {

        // Object—Oriented Programming

}

// Key words:
class, new, this, static, null, extends, super, abstract, final,
     interface, implements, protected
```

Observation in Real World

- Look around.
- We can easily find many examples for real-world objects.
    - For example, a person with a bottle of water.
- Real-world objects all have states and behaviors.
    - What states can the object need?
    - What behaviors can the object perform on the states?
- Identifying these states and behaviors for real-world objects is a great way to begin thinking in object-oriented programming.
- From now, OO is a shorthand for “object-oriented.”

Objects

- An object keeps its states in fields (or attributes) and exposes its behaviors through methods.
- Plus, we hide internal states and expose methods which perform actions on the aforesaid states.
- This is so-call encapsulation, which is one of OO features.[1]
- Before we create the objects, we need to define a new class as their prototype (or concept).

[1]The rest of features in OO are inheritance and polymorphism, which we will see later.

Classes

- We often find many objects all of the same kind.
  - For example, student A and student B are two instances of "student".
  - Every student needs a name and a student ID.
  - Every student should do homework and pass the final exams.
- A class is the blueprint to create class instances which are runtime objects.
  - In the other word, an object is an instance of some associated class.
- In Java, classes are the building blocks in every program.
- Once the class is defined, we can use this class to create objects.

Example: Points in 2D Coordinate

```java
public class Point {
    // data members: so—called fields or attributes
    double x, y;
}
```

```java
public class PointDemo {
    public static void main(String[] args) {
         // now create a new instance of Point
        Point p1 = new Point();
        p1.x = 1;
        p1.y = 2;
        System.out.printf("(%d, %d)\n", p1.x, p1.y);

        // create another instance of Point
        Point p2 = new Point();
        p2.x = 3;
        p2.y = 4;
        System.out.printf("(%d, %d)\n", p2.x, p2.y);
    }
}
```

Class Definition

- First, give a class name with the first letter capitalized, by convention.
- The class body, surrounded by balanced curly braces {}, contains data members (fields) and function members (methods).

Data Members

- As mentioned earlier, these fields are the states of the object.
- Each field may have an access modifier, say public and private.
  - public: accessible by all classes
  - private: accessible only within its own class
- We can decide if these fields are accessible!
- In practice, all fields should be declared private to fulfill the concept of encapsulation.
- However, this private modifier does not quarantine any security.[2]
  - What private is good for maintainability and modularity.[3]

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

[3] 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 if necessary:
  - getters: return some 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)

```java
public class Point {
    // data members: fields or attributes
    private double x;
    private double y;

    // function members: methods
    public double getX() { return x; }
    public double getY() { return y; }

    public void setX(double new_x) { x = new_x; }
    public void setY(double new_y) { y = new_y; }
}
```

Exercise: Phonebook

```java
public class Contact {
    private String name;
    private String phoneNumber;

    public double getName() { return name; }
    public double getPhoneNumber() { return phoneNumber; }

    public void setName(String new_name) { name = new_name; }
    public void setPhoneNumber(String new_phnNum) {
        phoneNumber = new_phnNum;
    }
}
```

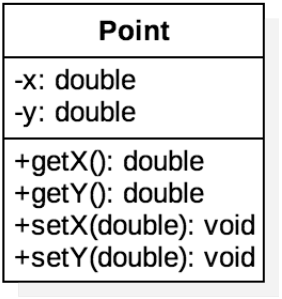
```java
public class PhonebookDemo {

    public static void main(String[] args) {
        Contact c1 = new Contact();
        c1.setName("Arthur");
        c1.setPhoneNumber("09xxnnnnnn");

        Contact c2 = new Contact();
        c1.setName("Emma");
        c1.setPhoneNumber("09xxnnnnnn");

        Contact[] phonebook = {c1, c2};

        for (Contact c: phonebook) {
            System.out.printf("%s: %s\n", c.getName(),
                                          c.getPhoneNumber());
        }
    }

}
```

Unified Modeling Language[4]

- 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)

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

Example: Class Diagram for Point



- Modifiers can be placed before both fields and methods:
  - $+$ for public
  - $-$ for private

Constructors

- A constructor follows the new operator.
- A constructor acts like other methods.
- However, its names 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.
- Constructors are used only during the objection creation.
  - 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 provide specific information to objects by using parameterized constructors.
- For example,

```java
public class Point {
    ...
    // default constructor
    public Point() {
        // do something in common
    }

    // parameterized constructor
    public Point(double new_x, double new_y) {
        x = new_x;
        y = new_y;
    }
    ...
}
```

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.
- You can also use this to call another constructor in the same class by invoking this().

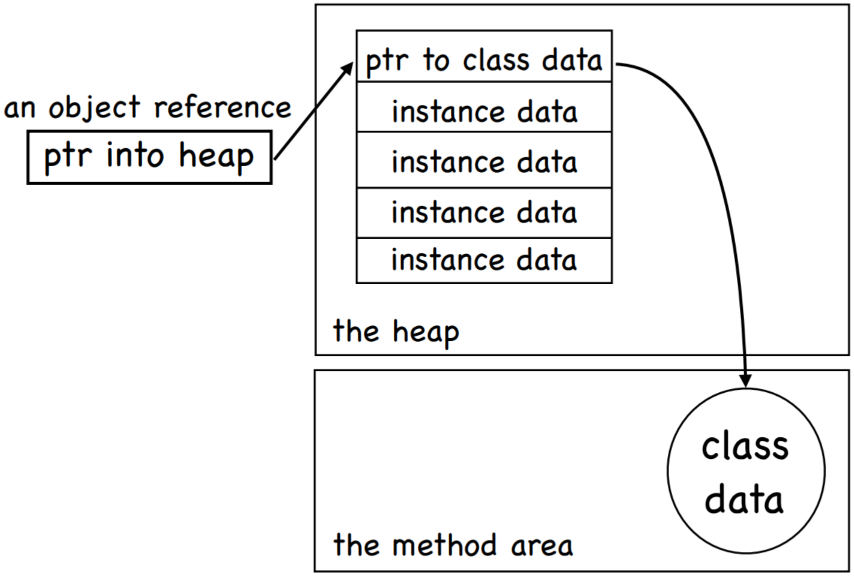
Example: Point (Revisited)

```
public class Point {
    ...
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

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

Instance Members

- You may notice that, until now, all members are declared w/o static.
- These members are called instance members.
- These instance members are available only after the object is created.
- This implies that each object has its own states and does some actions.


an object reference
ptr into heap
ptr to class data
instance data
instance data
instance data
instance data
the heap
class data
the method area

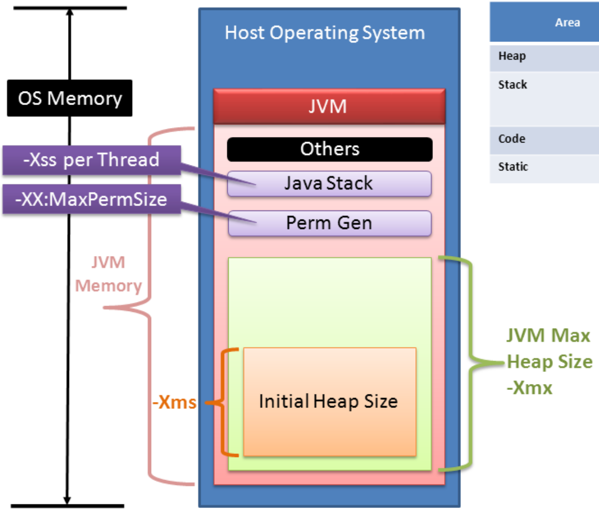
Static Members

- The static members belong to the class[5], and are shared between the instance objects.
- These members are ready once the class is loaded.
  - For example, the main method.
- They can be invoked directly by the class name without using any instance.
  - For example, **Math**.random() and **Math**.PI.
- They are particularly useful for utility methods that perform work that is independent of instances.
  - For example, factory methods in design patterns.[6]

[5] As known as class members.

[6] "Design pattern is a general reusable solution to a commonly occurring problem within a given context in software design." by Wikipedia.

Memory used by JVM



Area	Contains
Heap	Objects
Stack	- Methods - Local Variables - Reference Variables
Code	Byte Code
Static	Static Methods and Data

-Xmx : The JVM max Heap Size

-Xss : The Java Thread Stack Size, the default is OS and JVM dependent, and it can range 256k-to-1MB.The default should be tuned down to a range that doesn't cause StackOverflow. I often use 128k-192k. Since the default –Xss is high, tuning it down can help save on memory used and given back to the Guest OS.

Perm Size is an area additional to the –Xmx (Max Heap) value and is not GC'ed because it contains class-level information.

"other mem" is additional mem required for NIO buffers, JIT code cache, classloaders, Socket Buffers (receive/send), JNI, GC internal info

JVM Memory = JVM Max Heap (-Xmx value) + JVM Perm Size (-XX:MaxPermSize) + NumberOfConcurrentThreads * (-Xss value) + "other mem"

- A static method can access other static members. (Trivial.)
- However, static methods cannot access to instance members directly. (Why?)
- For example,

```java
...
    public double getDistanceFrom(Point that) {
        return Math.sqrt(Math.pow(this.x - that.x, 2)
                                   + Math.pow(this.y - that.y, 2));
    }

    public static double measure(Point first, Point second) {
        // You cannot use this.x and this.y here!
        return Math.sqrt(Math.pow(first.x - second.x, 2)
                                   + Math.pow(first.y - second.y, 2));
    }
...
```

Example: Count of Points

```java
public class Point {
    ...
    private static int numOfPoints = 0;

    public Point() {
        numOfPoints++;
    }

    public Point(int x, int y) {
        this(); // calling the constructor with no argument
                // should be placed in the first line
        this.x = x;
        this.y = y;
    }
    ...
}
```

Exercise: Singleton

- In some situations, you may create the only instance of the class.

```java
public class Singleton {

    // Do now allow to invoke the constructor by other classes.
    private Singleton() {}

    // Will be ready as soon as the class is loaded.
    private static Singleton INSTANCE = new Singleton();

    // Only way to obtain this singleton by the outside world.
    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Garbage Collection (GC)[8]

- Java handles deallocation[7] automatically.
  - Timing: preset period or when memory stress occurs.
- GC is the process of looking at the heap, identifying if the objects are in use, and deleting those unreferenced objects.
- An object is unreferenced if the object is no longer referenced by any part of your program. (How?)
  - Simply assign null to the reference to make the object unreferenced.
- Note that you may invoke **System**.**gc**() to execute the deallocation procedure.
  - However, frequent invocation of GC is time-consuming.

[7]Release the memory occupied by the unused objects.

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

finalize()

- The method **finalize**() conducts a specific task that will be executed right before the object is reclaimed by GC.
  - For example, closing files and terminating network connections.
- The **finalize**() method can be only invoked prior to GC.
- In practice, it must not rely on the **finalize**() method for normal operations. (Why?)

Example

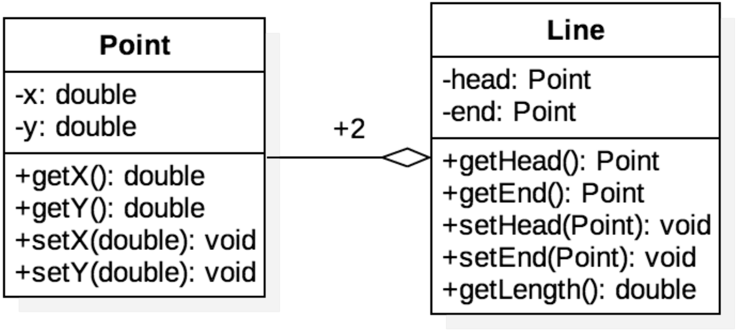
```java
public class Garbage {
    private static int numOfObjKilled = 0;

    public void finalize() {
        numOfObjKilled++;
    }

    public static void main(String[] args) {
        double n = 1e7;
        for (int i = 1; i <= n; i++)
            new Garbage(); // lots of unreferenced objects
        System.out.println(numOfObjKilled);
    }
}
```

- You may try different number for instance creation.
- The number of the objects reclaimed by GC is uncertain.

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.
  - For example, house $\leftrightarrow$ room.

Example: Lines



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

```java
public class Line {
    private Point head, end;

    public Line(Point h, Point e) {
        head = h;
        end = e;
    }

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

    public static double measureLength(Line line) {
        return line.getLength();
    }
}
```

More Examples

- **Circle**, **Triangle**, and **Polygon**.
- **Book** with **Author**s.
- **Lecturer** and **Student**s in the classroom.
- **Zoo** with many creatures, say **Dog**, **Cat**, and **Bird**.
- **Channel**s played on **TV**.
- More.

More About Objects

- Inheritance: passing down states and behaviors from the parents to their children.
- Interfaces: requiring objects for the demanding methods which are exposed to the outside world.
- Polymorphism
- Packages: grouping related types, and providing access controls and name space management.
- Immutability
- Enumeration types
- Inner classes

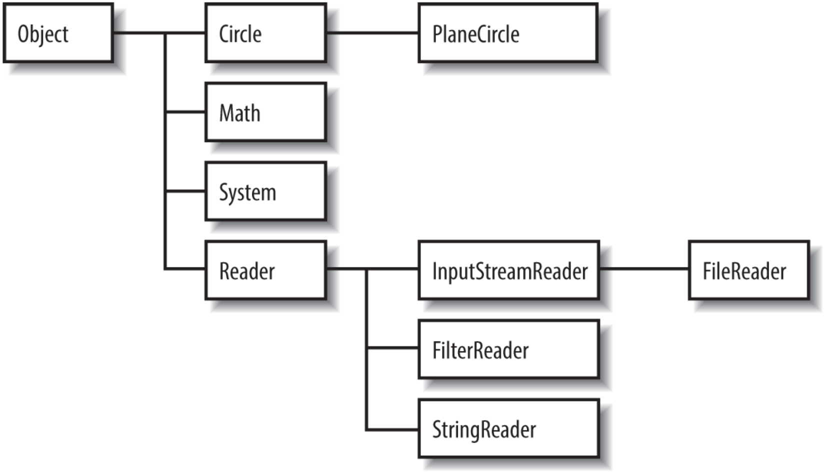
First IS-A Relationship: Inheritance

- The relationships among Java classes form class hierarchy.
- We can define new classes by inheriting commonly used states and behaviors from predefined classes.
- A class is a subclass of some class, which is so-called the superclass, by using the extends keyword.
  - For example, **B** extends **A**.
- In semantics, **B** is a special case of **A**, or we could say **B** specializes **A**.
  - For example, human and dog are two specific types of animals.
- When both **B** and **C** are subclasses of **A**, we say that **A** generalizes **B** and **C**. (Déjà vu.)
- Note that Java allows single inheritance only.

Example

```java
class Animal {
    String name;
    int weight;

    Animal(String s, int w) { name = s; weight = w; }

    void eat() { weight++; }
    void exercise() { weight--; }
}

class Human extends Animal {
    Human(String s, int w) { super(s, w); }
    void writeCode() {}
}

class Dog extends Animal {
    Dog(String s, int w) { super(s, w); }
    void watchDoor() {}
}
```

Class Hierarchy[9]



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

super

- Recall that the keyword this is used to refer to the object itself.
- You can use the keyword 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().

Constructor Chaining

- As the constructor is invoked, the constructor of its superclass is invoked accordingly.
- 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.
- So every class is an immediate or a distant subclass of **Object**.
- Recall that the method **finalize**() and **toString**() are inherited from **Object**.
    - **toString**(): return a string which can be any information stored in the object.

Example

```java
class A {
    A() { System.out.println("A is creating..."); }
}

class B extends A {
    B() { System.out.println("B is creating..."); }
    // overriding toString()
    public String toString() { return "I am B."; }
}

public class ConstructorChainingDemo {
    public static void main(String[] args) {
        B b = new B();
        System.out.println(b);
    }
}
```

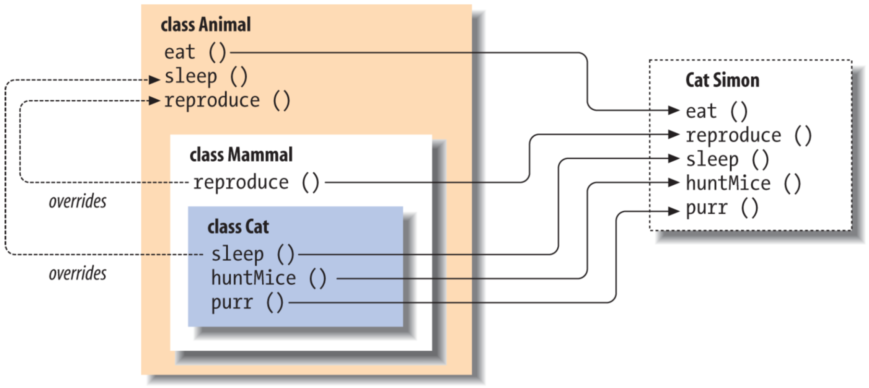
- The **println**() method (and similar methods) can take an object as input, and invoke **toString**() method implicitly.

Method Overriding

- The subclass is allowed to change the behavior inherited from its superclass, if needed.
- If one defines an instance method with its method name, parameters, and also return type, all identical to the previously defined method in its superclass, then we say this newly-defined method overrides the one in the superclass.[10]
    - Recall that method overloading occurs only in the same class.
- Note that you can invoke the overridden method through the use of the keyword super.

[10] Notice that the static methods do not follow this rule.

Example


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

Binding

- Association of the method definition to the method call is known as binding.
- The binding which can be resolved at the compilation time is known as static binding or early binding.
  - They are the static, private or final methods.[11]
- If the compiler is not able to resolve the binding, such binding is known as dynamic binding or late binding.
  - For example, method overriding.

[11] We will see the final keyword soon.

- When there are multiple implementations of the method in the inheritance hierarchy, the one in the "most derived" class (the furthest down the hierarchy) always overrides the others, even if we refer to the object through a reference variable of the superclass type.[12]
    - As you can see in Cat Simon.
- This is so-called subtype polymorphism.

[12] An overridden method in Java acts like a virtual function in C++.

Polymorphism[13]

- The word polymorphism literally means "many forms."
- Java allows 4 types of polymorphism:
    - coercion (casting)
    - ad hoc polymorphism (overloading)
    - subtype polymorphism
    - parametric polymorphism (generics)
- Modeling polymorphism in a programming language lets you create a uniform interface to different kinds of operands, arguments, and objects.

[13] Read `http://www.javaworld.com/article/3033445/learn-java/java-101-polymorphism-in-java.html`.

Example: Uniform Interface

```java
class Student {
    void doMyWork() { /* Do not know the detail. */}
}

class HighSchoolStudent extends Student {
    void writeHomework() {
        System.out.println("Write homework orz");
    }

    void doMyWork() { writeHomework(); }
}

class CollegeStudent extends Student {
    void writeReports() {
        System.out.println("Write reports qq");
    }

    void doMyWork() { writeReports(); }
}
```

```java
public class PolymorphismDemo {

    public static void main(String[] args) {
        HighSchoolStudent h = new HighSchoolStudent();
        goStudy(h);
        CollegeStudent c = new CollegeStudent();
        goStudy(c);
    }

    // uniform interface, multiple implementations
    // for future extension (scalability)
    public static void goStudy(Student s) {
        s.doMyWork();
    }

    /* no need to write these methods
    public static void goStudy(HighSchoolStudent s) {
        s.writeHomework();
    }

    public static void goStudy(CollegeStudent s) {
        s.writeReports();
    }
    */
}
```

Subtype Polymorphism

- For convenience, let **U** be a subtype of **T**.
- Liskov Substitution Principle 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.).[14,15]

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

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

Casting

- Upcasting (widening conversion) is to cast the **U** object to the **T** variable.

```java
T t = new U();
```

- Downcasting (narrow conversion) is to cast the **T** variable to a **U** variable.

```java
U u = (U) t; // t is T variable reference to a U object.
```

- Upcasting is always allowed, but downcasting is allowed only when a **U** object is passed to the **U**-type variable.
  - Java type system makes sure that the referenced object provides services adequate for **T** type.

instanceof

- However, type-checking in compilation time is unsound.
- The operator instanceof checks if an object reference is an instance of a type, and returns a boolean value.

Example

```java
class T {}
class U extends T {}

public class InstanceofDemo {
    public static void main(String[] args) {
        T t1 = new T();

        System.out.println(t1 instanceof U); // output false
        System.out.println(t1 instanceof T); // output true

        T t2 = new U(); // upcasting

        System.out.println(t2 instanceof U); // output true
        System.out.println(t2 instanceof T); // output true

        U u = (U) t2; // downcasting; this is ok.

        u = (U) new T(); // pass the compilation; fail during
            execution!
    }
}
```

Abstraction, Method Overriding, and Polymorphism

- JVM invokes the appropriate method for the current object by looking up from the bottom of the class hierarchy to the top.
    - These methods are also called virtual methods.
- This preserves the behaviors of the subtype objects and the super-type variables play the role of placeholder.
- We often manipulate objects in an abstract level; we don't need to know the details when we use them.
    - For example, computers, cellphones, driving.

Exercise

- Imagine that we have a zoo with some animals.

```
class Animal {
    void speak() {}
}
class Dog extends Animal {
    void speak() { System.out.println("woof"); }
}
class Cat extends Animal {
    void speak() { System.out.println("meow"); }
}
class Bird extends Animal {
    void speak() { System.out.println("tweet"); }
}

public class PolymorphismDemo {
    public static void main(String[] args) {
        Animal[] zoo = {new Dog(), new Cat(), new Bird()};
        for (Animal a: zoo) a.speak();
    }
}
```

final

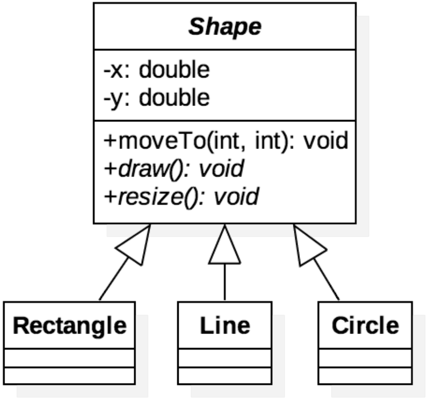
- A final variable is a variable which can be initialized once and cannot be changed later.
    - The compiler makes sure that you can do it only once.
    - A final variable is often declared with static keyword and treated as a constant, for example, **Math.PI**.
- A final method is a method which 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.

Abstract Classes

- An abstract class is a class declared abstract.
- The classes that sit at the top of an object hierarchy are typically abstract classes.[16]
- These abstract class may or may not have abstract methods, which are methods declared without implementation.
    - More explicitly, the methods are declared without braces, and followed by a semicolon.
    - If a class has one or more abstract methods, then the class itself must be declared abstract.
- All abstract classes cannot be instantiated.
- Moreover, abstract classes act as placeholders for the subclass objects.

[16] The classes that sit near the bottom of the hierarchy are called concrete classes.

Example



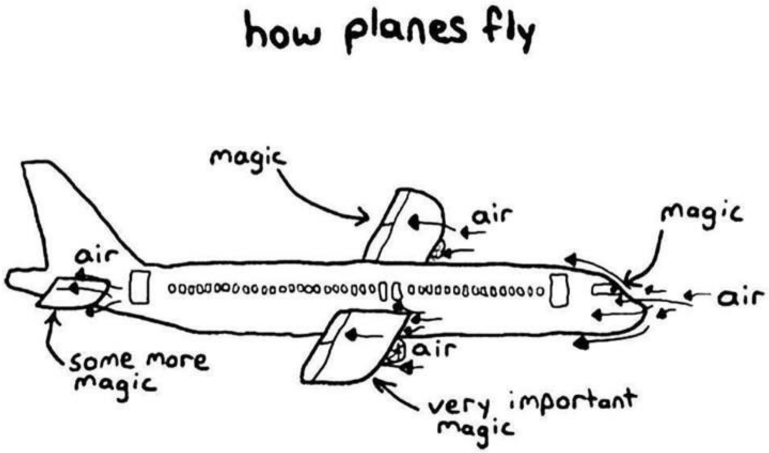
- Abstract methods and classes are in italic.
- In this example, the abstract method *draw*() and *resize*() should be implemented depending on the real shape.

Another IS-A Relationship

- Not all classes share a vertical relationship.
- Instead, some are supposed to perform the specific methods without a vertical relationship.
    - Consider the class **Bird** inherited from **Animal** and **Airplane** inherited from **Transportation**.
    - Both **Bird** and **Airplane** are able to be in the sky.
    - So they should perform the method canFly(), for example.
- By semantics, the method canFly() could not be defined in their superclasses.
- We need a horizontal relationship.

Example

```java
interface Flyable {
    void fly(); // implicitly public, abstract
}

class Animal {}

class Bird extends Animal implements Flyable {
    void flyByFlappingWings() {
        System.out.println("flapping wings");
    }

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

class Transportation {}

class Airplane extends Transportation implements Flyable {
    void flyByMagic() {
        System.out.println("flying with magicsssss");
    }

    public void fly() { flyByMagic(); }
}
```


how planes fly
magic
air
magic
air
air
air
some more magic
air
very important magic

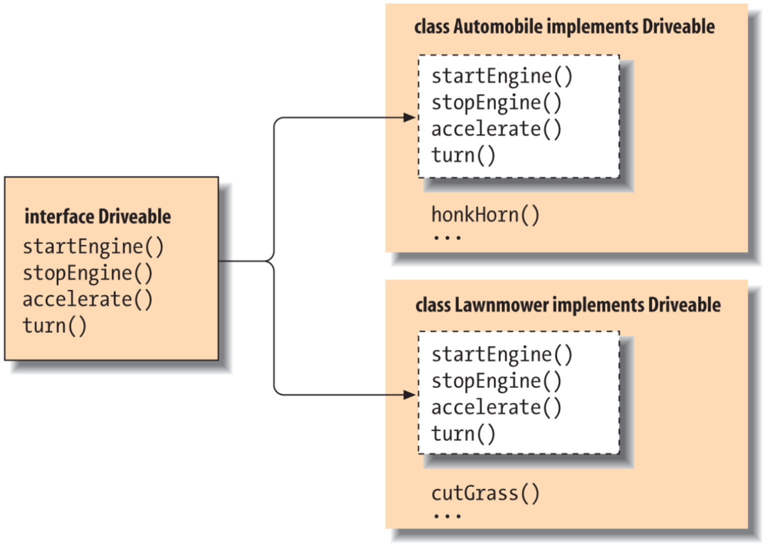
```java
public class InterfaceDemo {
    public static void main(String[] args) {
        Bird b = new Bird();
        goFly(b);

        Airplane a = new Airplane();
        goFly(a);
    }

    static void goFly(Flyable f) {
        f.fly();
    }
}
```

Interfaces

- An interface forms a contract between the object and the outside world.
    - For example, the buttons on the television set are the interface between you and the electrical wiring on the other side of its plastic casing.
- An interface is also a reference type, just like classes, in which only method signatures are defined.
- So they can be the types of reference variables!

- Note that interfaces cannot be instantiated (directly).
- A class implements one or multiple interfaces by providing method bodies for each predefined signature.
- This requires an object providing a different set of services.
- For example, combatants in RPG can also buy and sell stuffs in the market.

Example


interface Driveable
startEngine()
stopEngine()
accelerate()
turn()
class Automobile implements Driveable
startEngine()
stopEngine()
accelerate()
turn()
honkHorn()
...
class Lawnmower implements Driveable
startEngine()
stopEngine()
accelerate()
turn()
cutGrass()
...

Properties of Interfaces

- The methods of an interface are implicitly public.
- In most cases, the class which implements the interface should implement all the methods defined in the interface.
    - Otherwise, the class should be abstract.
- An interface can declare only fields which are static and final.
- You can also define static methods in the interface.
- An interface can extend another interface, just like a class which can extend another class.
    - In contrast with classes, an interface can extend many interfaces.

- Common interfaces are **Runnable**[17] and **Serializable**[18].
- A new feature since Java SE 8 allows to define the methods with implementation in the interface.
  - A method with implementation in the interface is declared default.

[17]See Java Multithread.

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

Timing for Interfaces and Abstract Classes

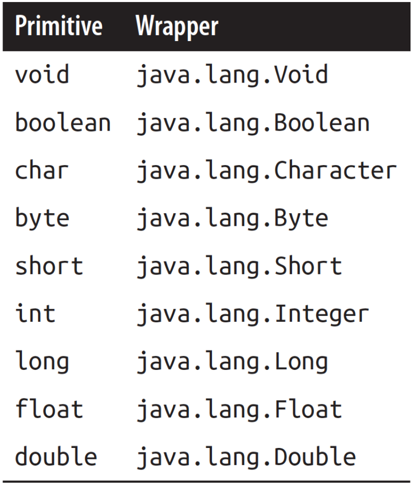
- Consider using abstract classes if you want to:
  - share code among several closely related classes
  - declare non-static or non-final fields
- Consider using interfaces for any of situations as follows:
  - unrelated classes would implement your interface
  - specify the behavior of a particular data type, but not concerned about who implements its behavior
  - take advantage of multiple inheritance
- Program to abstraction, not to implementation.[19]

[19] See software engineering or object-oriented analysis and design.

Wrapper Classes

- To treat values as objects, Java supplies standard wrapper classes for each primitive type.
- For example, you can construct a wrapper object from a primitive value or from a string representation of the value.

```
...
        Double pi = new Double("3.14");
...
```



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 their wrapper types, and unwraps them where appropriate.

```
...
        Integer i = 1; // autoboxing
        Integer j = 2;
        Integer k = i + 1; // autounboxing and then autoboxing
        System.out.println(k); // output 2

        System.out.println(k == j); // output true
        System.out.println(k.equals(j)); // output true
...
```

- The method **equals**() inherited from **Object** is used to compare two objects.
    - You may override this method if necessary.

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.

- For example,

```
...
        k = new Integer(1);
        System.out.println(i == k); // output false (why?)
        System.out.println(k.equals(i)); // output true
...
```

- Good practice when it comes to concurrent programming.[20]
- Another example is String objects.

[20]See http://www.javapractices.com/topic/TopicAction.do?Id=29.

enum Types[21]

- An enum type is an reference type limited to an explicit set of values.
- An order among these values is defined by their order of declaration.
- There exists a correspondence with string names identical to the name declared.

[21] The keyword enum is a shorthand for enumeration.

Example: Colors

```java
enum Color {
    RED, GREEN, BLUE; // three options

    static Color random() {
        Color[] colors = values();
        return colors[(int) (Math.random() * colors.length)];
    }
}
```

- Note that **Color** is indeed a subclass of enum type with 3 static and final references to 3 Color objects corresponding to the enumerated values.
- This mechanism enhances type safety and makes the source code more readable!

```
Class Pen {
    Color color;
    Pen(Color color) { this.color = color; }
}

Class Clothes {
    Color color;
    T_Shirt(Color color) { this.color = color; }
    void setColor(Color new_color) { this.color = new_color; }
}

public class EnumDemo {
    public static void main(String[] args) {
        Pen crayon = new Pen(Color.RED);
        Clothes T_shirt = new Clothes(Color.random());
        System.out.println(crayon.color == T_shirt.color);
    }
}
```

Exercise: Directions

```java
enum Direction {UP, DOWN, LEFT, RIGHT}

/* equivalence
class Direction {
    final static Direction UP = new Direction("UP");
    final static Direction DOWN = new Direction("DOWN");
    final static Direction LEFT = new Direction("LEFT");
    final static Direction RIGHT = new Direction("RIGHT");

    private final String name;

    static Direction[] values() {
        return new Direction[] {UP, DOWN, LEFT, RIGHT};
    }

    private Direction(String str) {
        this.name = str;
    }
}
*/
```

Packages

- We organize related types into packages for the following purposes:
  - To make types easier to find and use
  - To avoid naming conflicts
  - To control access
- For example, fundamental classes are in **java.lang** and classes for I/O are in **java.io**.

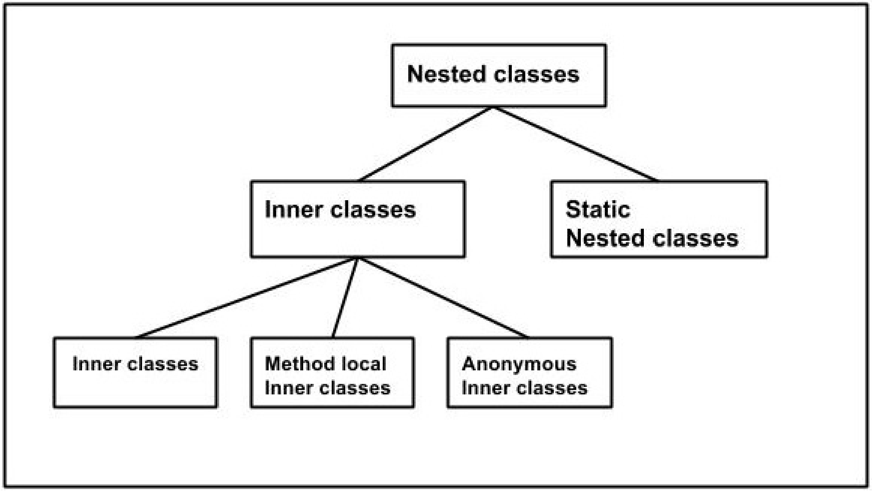
Access Control

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

Nested Classes

- A nested class is a member of its enclosing class.
- Non-static nested classes have access to other members of the enclosing class, even if they are declared private.
- Instead, static nested classes do not have access to other instance members of the enclosing class.
- We use nested classes when it needs to
  - logically group classes that are only used in one place
  - increase encapsulation
  - lead to more readable and maintainable code

Family of Nested Classes



Non-Static Nested Classes

- Depending on how and where you define them, they can be further divided in three types:
  - inner classes
  - method-local inner classes
  - anonymous inner classes
- Unlike a normal class, an inner class can be declared private.
- Note that the creation of inner-type objects is available after the outer-type object is created.
  - In other words, you cannot invoke the constructor of the inner type without having the outer type object.
- For static members in the inner classes,
  - you can declare a static variable which is supposed to be final;
  - however, static methods can only be declared in a static or top level type.

Example: Inner Class

```java
class OuterClass {
    private int x = 1;
    InnerClass innerObject = new InnerClass();

    class InnerClass {
        public void print() {
            System.out.println(x); // ok!
        }
    }
}

public class InnerClassDemo {
    public static void main(String[] args) {
        OuterClass outerObject = new OuterClass();
        outerObject.innerObject.print(); // output 1

        // you cannot do below
        InnerClass innerObject = new InnerClass();
    }
}
```

Example: Method-Local Inner Class

```java
class OuterClass {
    private int x = 1;

    void doSomething() {
        class LocalClass { // should be in the beginning
            int y = 2;
            static int z = 3; // implicitly final

            void print() {
                System.out.println(x);
                System.out.println(y);
                System.out.println(z);
            }
        }

        LocalClass w = new LocalClass();
        w.print();
    }
}

public class InnerClassDemo {
    ...
}
```

Anonymous Inner Class

- Anonymous inner classes are an extension of the syntax of the new operation, enabling you to declare and instantiate a class at the same time.
    - However, these do not have a name.
- Use them when you need to use these types only once.

Example

```java
abstract class A {
    abstract void foo();
}

public class AnonymousClassDemoOne {
    public static void main(String[] args) {
        A a = new A() {
            public void foo() { /* different implementation */ }
            void helper() { /* a subroutine for foo */ }
        };

        a.foo();
    }
}
```

- You may invoke a.foo() but not a.helper() because helper() is not defined in class A.

Exercise

```java
interface B {
    void foo();
}

public class AnonymousClassDemoTwo {
    public static void main(String[] args) {
        B b = new B() {
            public void foo() { /* different implementation */ }
        };

        b.foo();
    }
}
```

- An interface can be used to instantiate an object indirectly by anonymous classes with implementing the abstract methods.

One of Adapters: Iterators

- An important use of inner classes is to define an adapter class as a helper object.
- Using adapter classes, we can write classes more naturally, without having to anticipate every conceivable user's needs in advance.
- Instead, you provide adapter classes that marry your class to a particular interface.
- For example, an iterator is a simple and standard interface to enumerate elements in data structures.
  - The class which implements the interface **Iterable** has the responsibility to provide an iterator.
  - An iterator is defined in the interface **Iterator** with two uninplemented methods: hasNext() and next().

Example

```java
import java.util.Iterator;

class Box implements Iterable<Integer> {

    int[] items = {10, 20, 30};

    public Iterator iterator() {
        return new Iterator() {
            private int ptr = 0;

            public boolean hasNext() {
                return ptr < items.length;
            }

            public Integer next() {
                return items[ptr++];
            }
        };
    }
}
```

```java
public class IteratorDemo {
    public static void main(String[] args) {
        Box myBox = new Box();

        // for—each loop
        for (Integer item: myBox) {
            System.out.println(item);
        }

        // equivalence
        Iterator iterOfMyBox = myBox.iterator();
        while (iterOfMyBox.hasNext())
            System.out.println(iterOfMyBox.next());
    }
}
```

Static Nested Class

- A static inner class is a nested class declared static.
  - Similar to the static members, they can access to other static members without instantiating the outer class.
  - Also, a static nested class does not have access to the instance members of the outer class.
- In particular, the static nested class can be instantiated directly, without instantiating the outer class object first.
  - Static nested classes act something like a minipackage.

Example

```java
class OuterClass {
    static int x = 1;
    int y = 2;

    static class StaticClass {
        int z = 3;
        void doSomething() {
            System.out.println(x);
            System.out.println(y); // cannot do this
            System.out.println(z);
        }
    }
}

public class StaticNestedClassDemo {
    public static void main(String[] args) {
        OuterClass.StaticClass x = new OuterClass.StaticClass();
        x.doSomething();
    }
}
```

Classpath[22]

- The variable **classpath** is an environment variable for the Java compiler to specify the location of user-defined classes and packages.
  - By default, only the packages of the JDK standard API and extension packages are accessible without needing to set where to find them.
- The path for all user-defined packages and libraries must be set in the command-line (or in the Manifest associated with the JAR file containing the classes).

[22] https://en.wikipedia.org/wiki/Classpath_(Java)

Usage of Classpath

- You may use the following command in any terminal:

  java -cp [the absolute path of the classes or packages] [the full name of the application to run]

- For Windows users, try

  ```
  java -cp c:\workspace\project train.java.HelloWorld
  ```

- On Linux/Unix/Mac OS users, try

  ```
  java -cp /workspace/project train.java.HelloWorld
  ```

Java Archive (JAR)[24]

- JAR is a packed format typically used to aggregate many Java class files, associated metadata[23] and resources (text, images, etc.) into one file to distribute the application software or libraries running on the Java platform.
  - Try an executable JAR!

[23] Metadata refers data of data.

[24] See `https://docs.oracle.com/javase/tutorial/deployment/jar/`.