Java Programming 2

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Java2 306
Fall 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.
- For example, describe your cellphone.
  - Attributes: battery status, 4G signal strength, contact info in your phonebook, photos in albums, musics, clips, and so on.
  - Functions?
- Before creating the objects, we need to define a new class as their prototype (or concept).

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

- 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 OO paradigm, we hide internal states and expose methods which perform actions on these fields.
  - So all fields should be declared private.
  - This is so-called encapsulation.
- However, this private modifier does not quarantine any 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 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)

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

Exercise: Phonebook

```
1  public class Contact {
2      private String name;
3      private String phoneNumber;
4
5      public String getName() { return name; }
6      public String getPhoneNumber() { return phoneNumber; }
7
8      public void setName(String new_name) { name = new_name; }
9      public void setPhoneNumber(String new_phnNum) {
10         phoneNumber = new_phnNum;
11     }
12 }
```

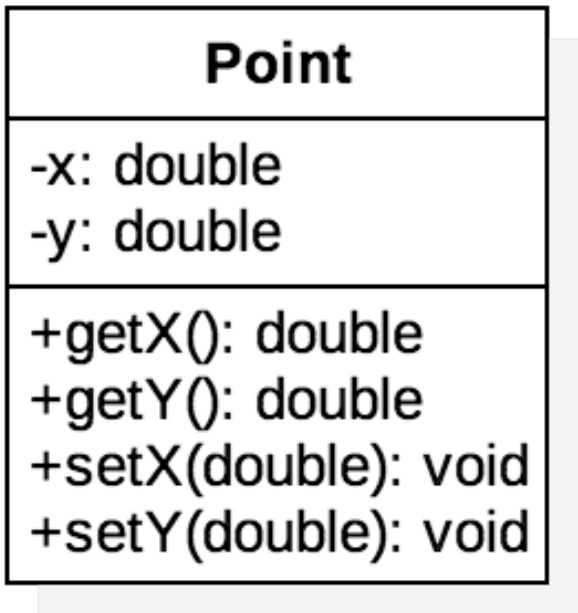
```java
public class PhonebookDemo {

    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[3]

- 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)

[3] 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, acting 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.
- 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 once the object is created.
- 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 new_x, double new_y) {
10         x = new_x;
11         y = new_y;
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().

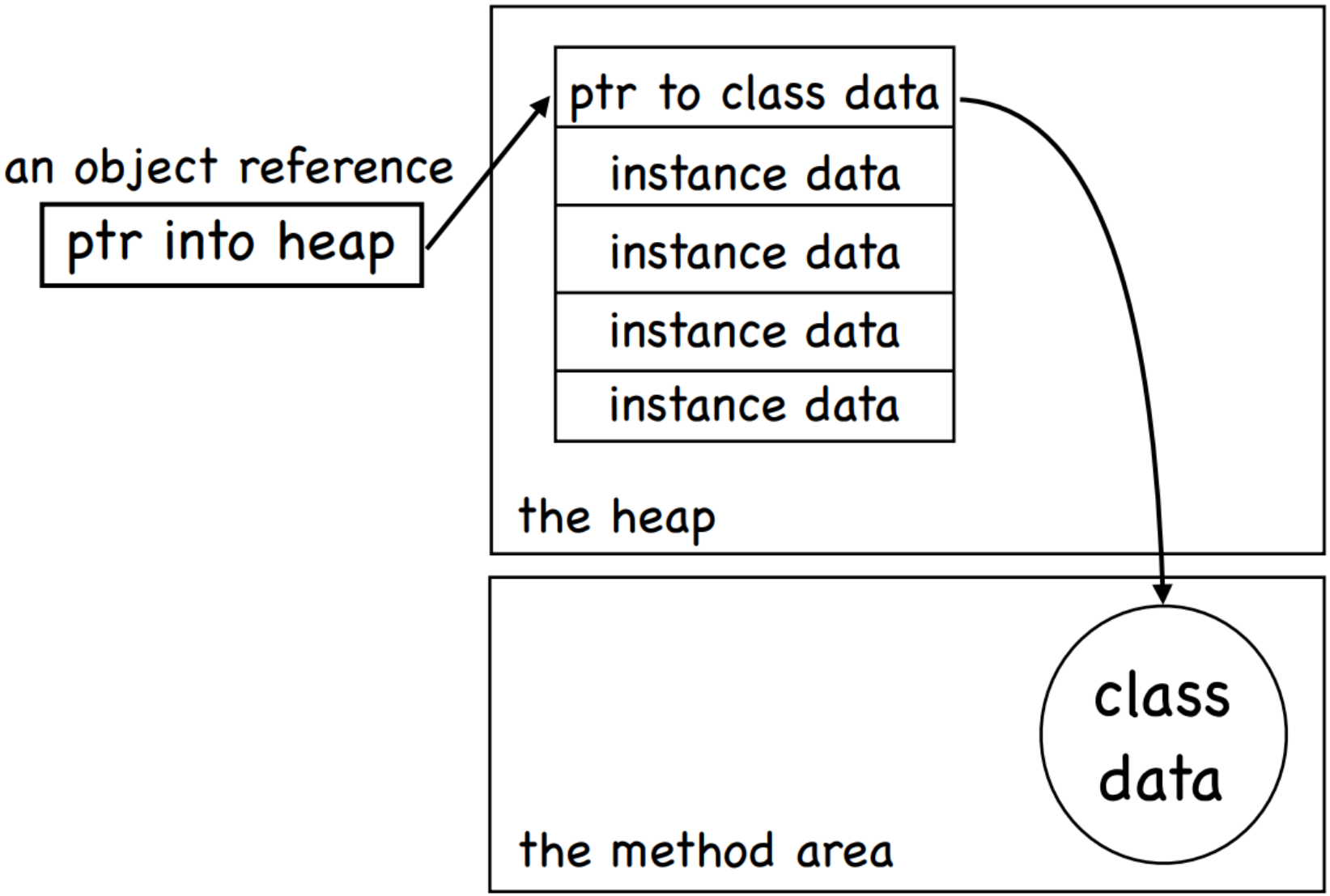
Example: Point (Revisited)

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

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

Instance Members

- Since this lecture, all members are declared w/o static, so-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

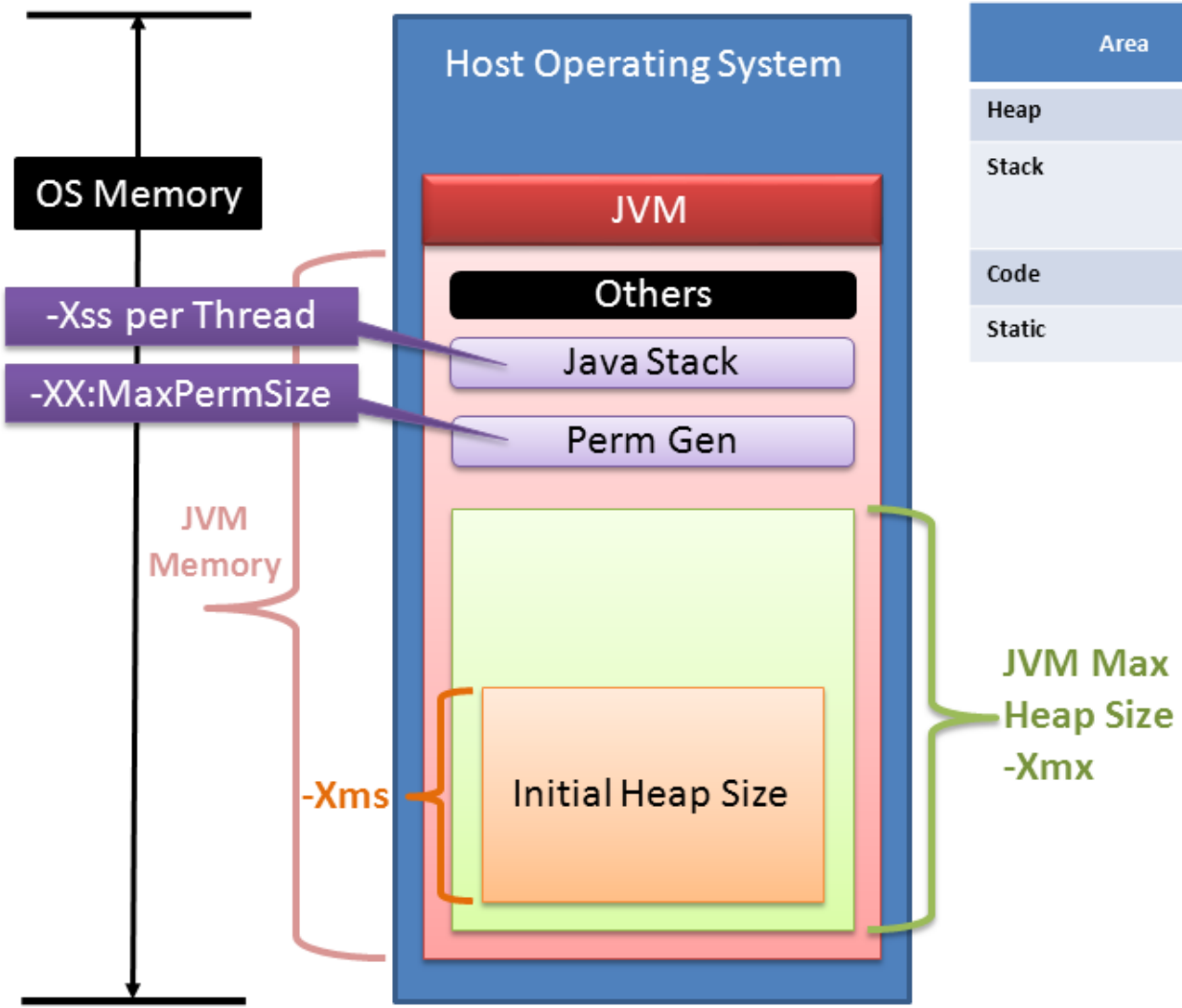
Static Members

- Static members belong to the class[4], and are shared between the instance objects.
- Those are ready once the class is loaded.
  - For example, the main methods.
- They can be invoked directly by the class name without any instance.
  - For example, Math.random() and Math.PI.
- Particularly useful for utility methods that perform work which is independent of instances.
  - For example, factory methods in design patterns.[5]

[4] Aka class members.

[5] "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,

```
1  ...
2      public double getDistanceFrom(Point that) {
3          return Math.sqrt(Math.pow(this.x - that.x, 2)
4                           + Math.pow(this.y - that.y, 2));
5      }
6
7      public static double measure(Point first, Point second) {
8          // You cannot use this.x and this.y here!
9          return Math.sqrt(Math.pow(first.x - second.x, 2)
10                          + Math.pow(first.y - second.y, 2));
11     }
12 ...
```

Example: Count of Points

```
1  public class Point {
2      ...
3      private static int numOfPoints = 0;
4
5      public Point() {
6          numOfPoints++;
7      }
8
9      public Point(int x, int y) {
10         this(); // calling Line 5
11         this.x = x;
12         this.y = y;
13     }
14     ...
15 }
```

- Note that invoking constructors (like Line 10) should be placed in the first statement in one constructor.

Exercise: Singleton

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

```
1  public class Singleton {
2
3      // Do now allow to invoke the constructor by other classes.
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 }
```

Garbage Collection (GC)[7]

- Java handles deallocation[6] 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.

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

[7] `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

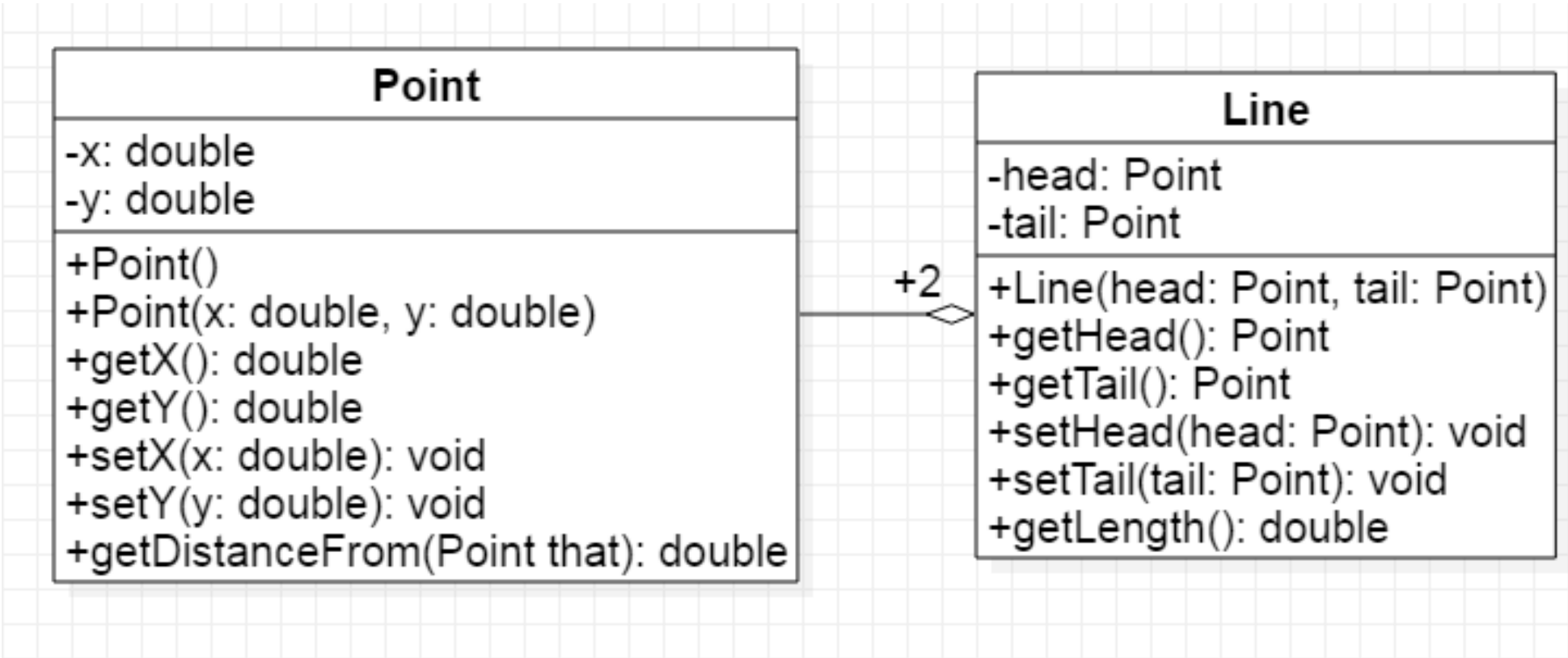
```
1  public class Garbage {
2      private static int numOfObjKilled = 0;
3
4      public void finalize() {
5          numOfObjKilled++;
6      }
7
8      public static void main(String[] args) {
9          double n = 1e7;
10         for (int i = 1; i <= n; i++)
11             new Garbage(); // lots of unreferenced objects
12         System.out.println(numOfObjKilled);
13     }
14 }
```

- 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, tail;

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

    /* ignore some methods */

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

    public static double measure(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

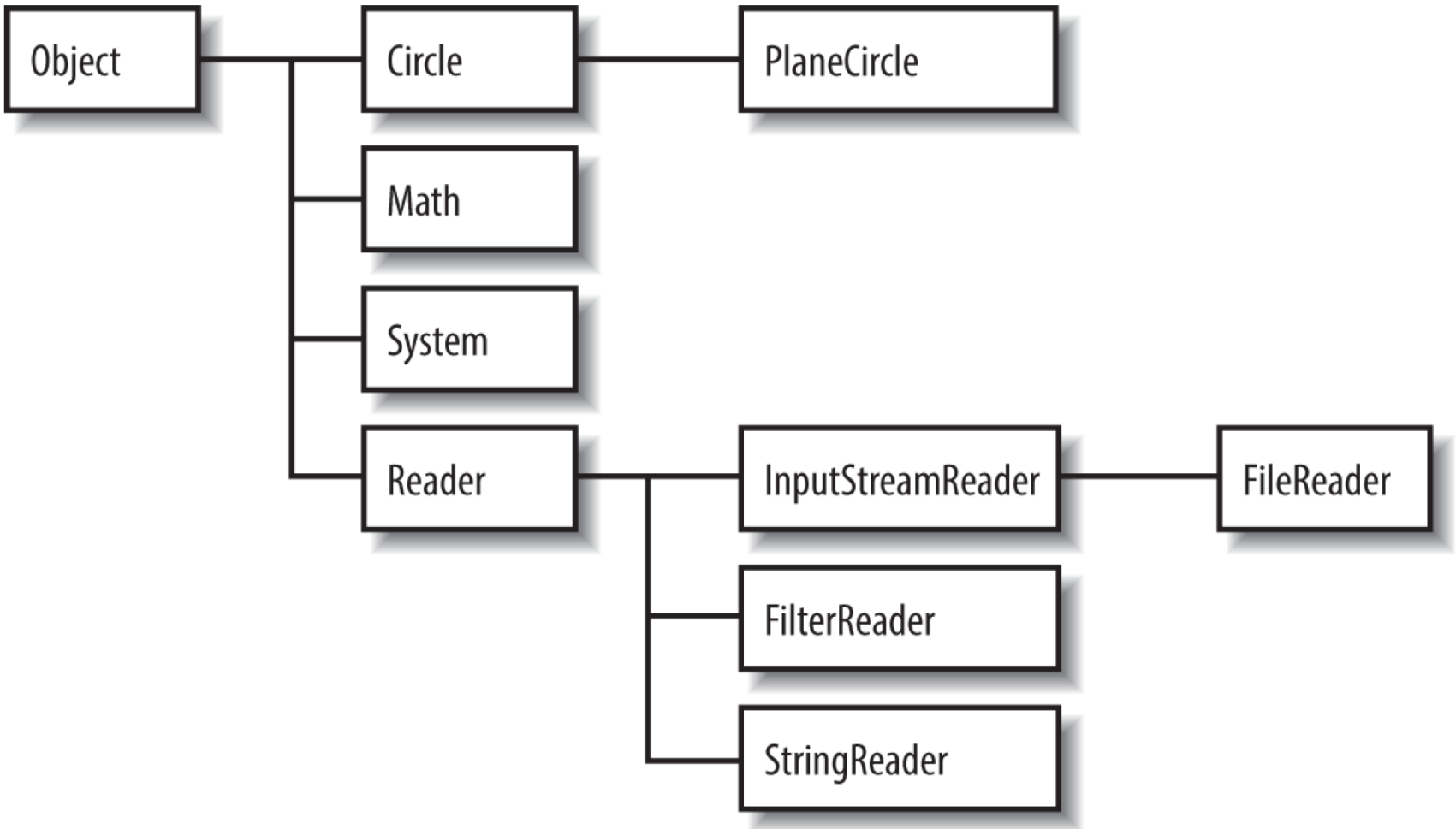
```
1  class Animal {
2      int weight;
3      void eat() { weight++; }
4      void exercise() { weight--; }
5  }
6
7  class Human extends Animal {
8      void writeCode() {}
9  }
10
11 class Dog extends Animal {
12     void watchDoor() {}
13 }
```

- How could **Human** and **Dog** possess those members of **Animal**?
- In this way, it is convenient to define, say **Cat**, by extending **Animal**.

Constructor Chaining

- Once the constructor is invoked, JVM will invoke the constructor of its superclass (recursively).
- 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[8]



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

Example: An Evidence

```
1  class A {
2      A() { System.out.println("A is creating..."); }
3  }
4
5  class B extends A {
6      B() {
7          super(); // you don't need to do this unless necessary.
8          System.out.println("B is creating...");
9      }
10 }
11
12 public class ConstructorChainingDemo {
13     public static void main(String[] args) {
14         B b = new B();
15     }
16 }
```

super

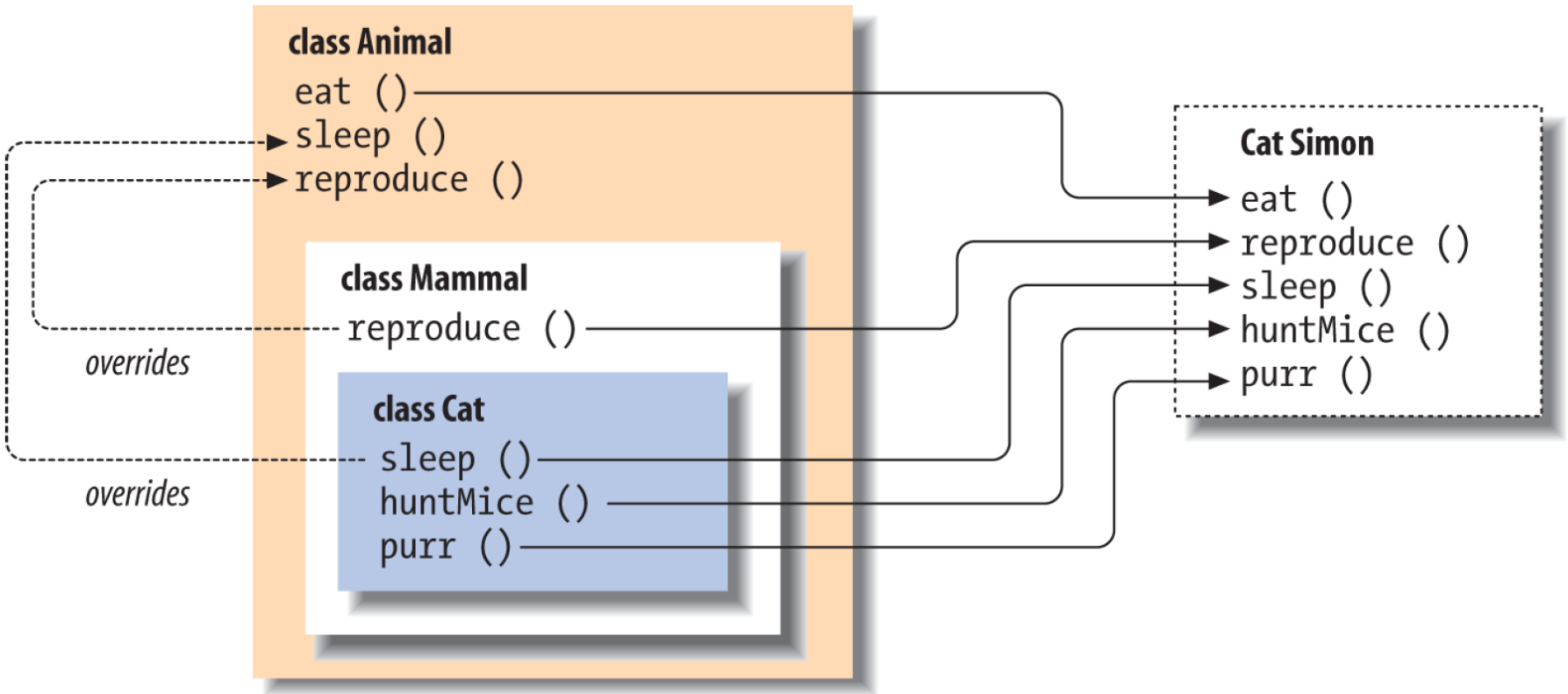
- Recall that this is used to refer to the object itself.
- You can use super to refer to (non-private) members of the superclass.
- Note that super() can be used to invoke the constructor of its superclass, just similar to this().

Method Overriding (1/2)

- A subclass is supposed to re-implement the methods inherited from its superclass.
  - Can you smell it?
- For example, **toString**() is inherited from **Object**.
  - This method will be invoked by **println**().
  - It returns the hashcode[9] of the object by default.
  - It could be overridden so it returns a string of desirable information.
- Another example we have encountered is **finalize**().

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

Example


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

Method Overriding (2/2)

- 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.
- Note that you cannot override the static methods.
- You could invoke the overridden method by using super.
- You should use the annotation[10] @Override to help you.

```
1 class Cat extends Animal {
2     @Override
3     void eat() { weight += 2; }
4
5 }
```

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

Polymorphism[12]

- The word polymorphism literally means "many forms."
- Java allows 4 types of polymorphism:
  - coercion (casting)
  - ad hoc polymorphism (overloading)
  - subtype polymorphism
  - parametric polymorphism (generics)[11]
- Subtype polymorphism allows you to create a single interface to different types (implementations).
- How to make a "single" interface for different types?
  - Use the superclass of those types as the placeholder.
  - Program to abstraction, not to implementation.

[11] We will introduce Java generics in Java Programming 2. Stay tuned.

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

Example: Dependency Reduction (Decoupling)

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

```java
public class PolymorphismDemo {

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

    public static void goStudy(Student s) {
        s.doMyJob();
    }

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

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

Why OOP?

- First, you may know that there are many programming paradigms.[13]
- OOP is the solid foundation of modern software design.
- In particular, encapsulation, inheritance, and polymorphism provide a great reuse mechanism and a great abstraction.
  - 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.[14]
  - Polymorphism exploits the superclass as a placeholder to manipulate the implementations (sub-type objects).

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

[14] This leads to the need of "single interface" as mentioned before.

- This leads to the production of frameworks[15], 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, computers, cellphones, driving.

[15] 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      /* ignore the previous part */
8      @Override
9      void speak() { System.out.println("woof"); }
10 }
11
12 class Cat extends Animal {
13     /* ignore the previous part */
14     @Override
15     void speak() { System.out.println("meow"); }
16 }
17
18 class Bird extends Animal {
19     /* ignore the previous part */
20     @Override
21     void speak() { System.out.println("tweet"); }
22 }
```

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

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.).[16,17]
- In other words, the references are clients asking the objects (right-hand side) for services!

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

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

Casting

- Upcasting (widening conversion) 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 (narrow conversion) 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?
```

Solution: instanceof

- Upcasting is always allowed, but downcasting is not always true even when you use the cast operator.
  - In fact, type checking at compile time is unsound just because the cast operator violets the functionality of type checking.
- Moreover, **T**-type reference can also point to the siblings of **U**-type.
  - Recall that **T**-type is used as the placeholder.
- We can use instanceof to check if the referenced object is of the target type at runtime.

Example

```
1  class T {}
2  class U extends T {}
3  class W extends T {}
4
5  public class InstanceofDemo {
6
7      public static void main(String[] args) {
8
9          T t = new U();
10
11         System.out.println(t instanceof T); // output true
12         System.out.println(t instanceof U); // output true
13         System.out.println(t instanceof W); // output false
14
15         W w = new W();
16
17         System.out.println(w instanceof T); // output true
18         System.out.println(w instanceof U); // output false
19         System.out.println(w instanceof W); // output true
20
21     }
22 }
```

final

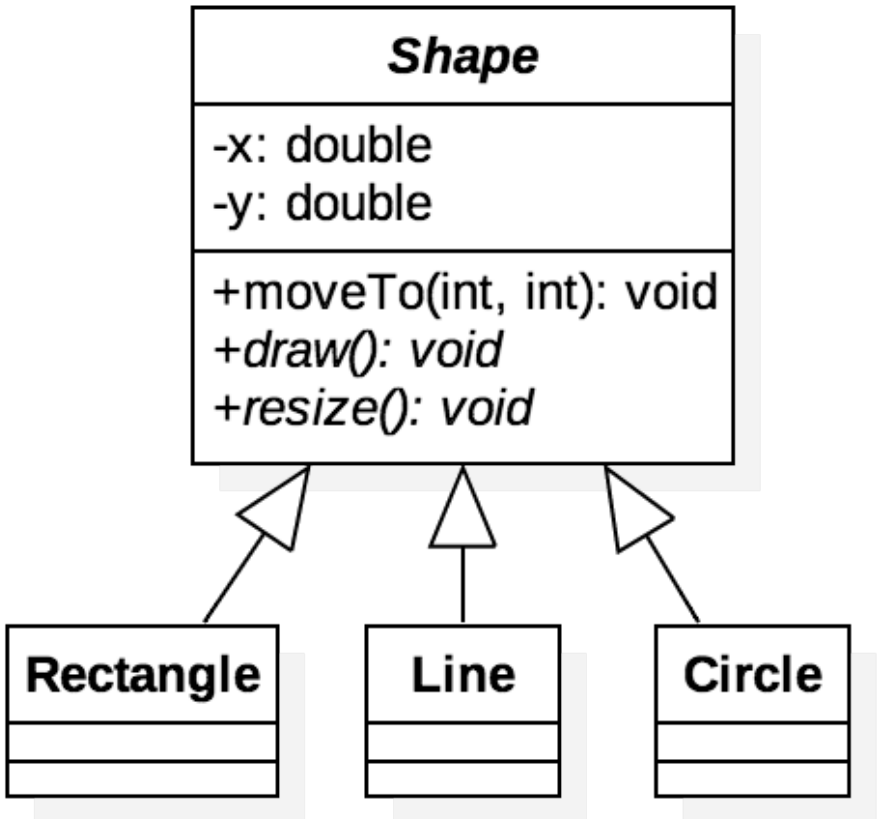
- A final variable is a variable which can be initialized once and cannot be changed later.
  - The compiler makes sure that you can do it only once.
  - A final variable is often declared with static keyword and treated as a constant, for example, Math.PI.
- A final method is a method which cannot be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

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.[18]
- 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.

[18] 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.

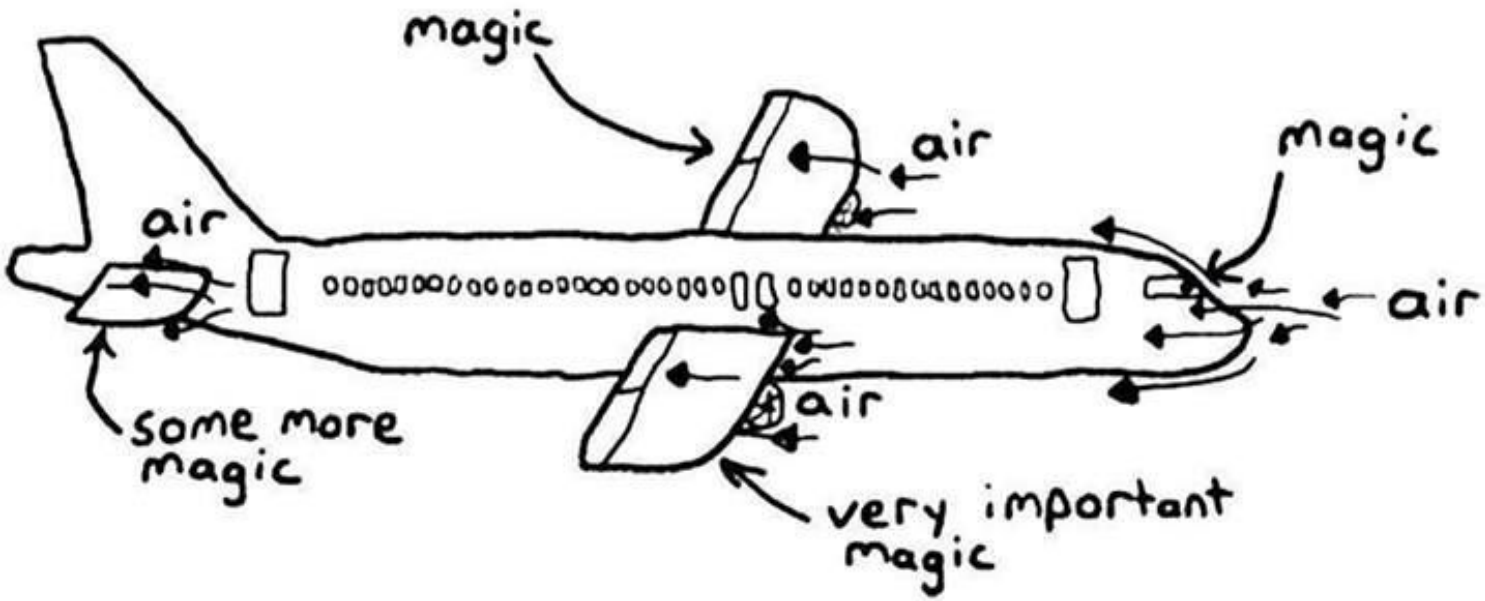
Another IS-A Relationship

- In some situations, objects are supposed to work together without a vertical relationship.
  - Consider the class **Bird** inherited from **Animal** and **Airplane** inherited from **Transportation**.
  - Both **Bird** and **Airplane** are able to fly in the sky.
  - Let's call the method fly(), for example.
- By semantics, the method fly() could not be defined in their superclasses. (Why?)
- Similar to the case study of Student, we wish those flyable objects go flying but in a single interface.
  - Using Object as the placeholder?
- Clearly, we need a horizontal relationship.

Example

```
1  interface Flyable {
2      void fly(); // implicitly public and abstract
3  }
4
5  class Animal {}
6
7  class Bird extends Animal implements Flyable {
8      void flyByFlappingWings() {
9          System.out.println("flapping wings");
10     }
11     @Override
12     public void fly() { flyByFlappingWings(); }
13 }
14
15 class Transportation {}
16
17 class Airplane extends Transportation implements Flyable {
18     void flyByMagic() {
19         System.out.println("flying with magicssssss");
20     }
21     @Override
22     public void fly() { flyByMagic(); }
23 }
```

how planes fly

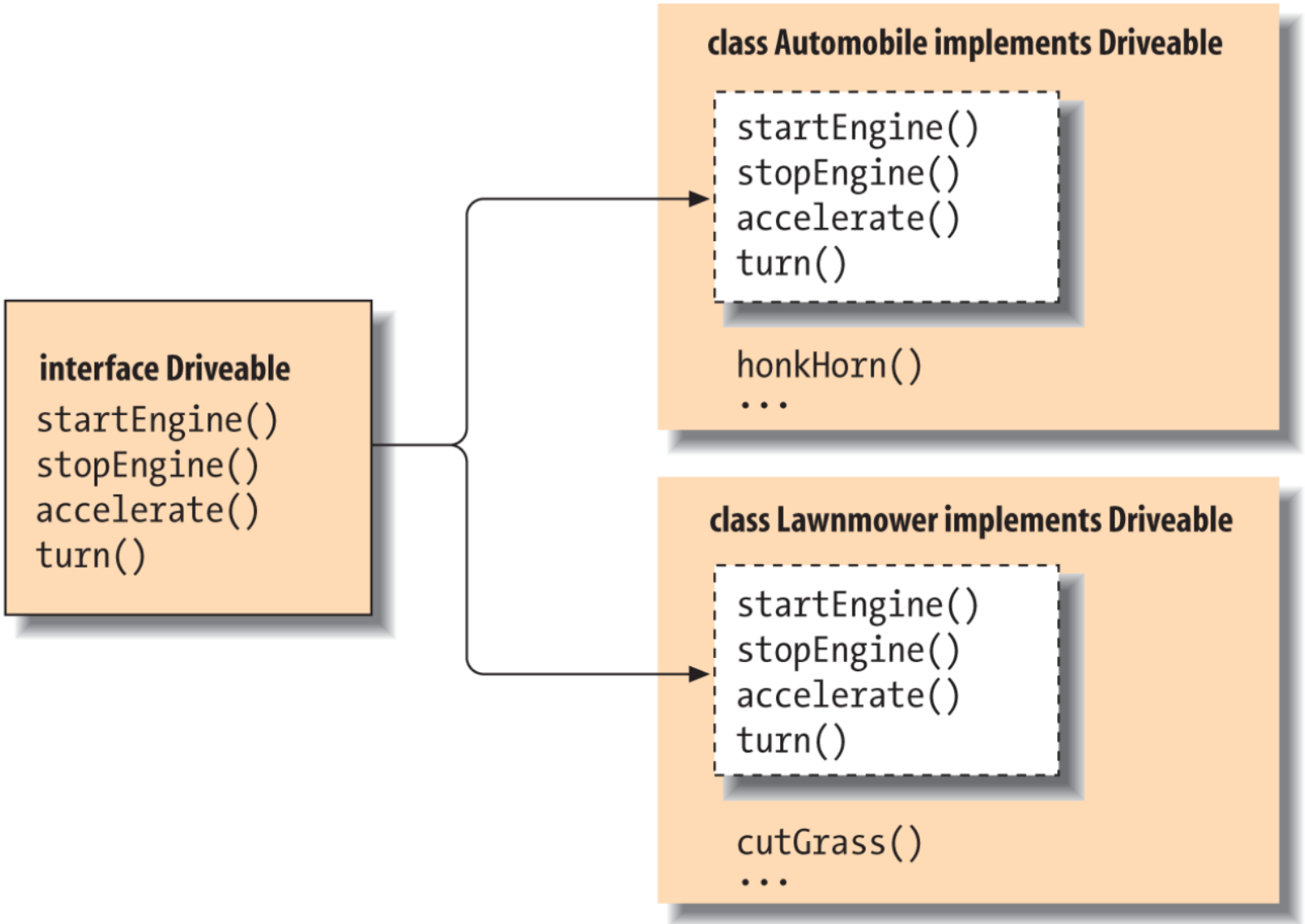


```java
public class InterfaceDemo {
    public static void main(String[] args) {
        Bird b = new Bird();
        goFly(b);

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

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

Interfaces (1/2)

- An interface forms a contract between the object and the outside world.
  - For example, the buttons on remote controls for some machine.
- As you can see, an interface is a reference type, just like classes
- Unlike classes, interfaces are used to define methods w/o implementation so that they cannot be instantiated (directly).
- A class could 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()
...

Interfaces (2/2)

- An interface can extend another interfaces.
  - Like a collection of contracts, in some sense.
- For example, **Runnable**[19] and **Serializable**[20] are two of Java interfaces.
- In JDK8, we have new features as follows:
  - we can declare static fields[21] and methods in the interfaces;
  - we can also define default methods in the interfaces;
  - Java provides so-called functional interfaces for lambdas which are widely used in the stream framework. (Stay tuned in Java 2!)

[19] See Java Multithread.

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

[21] But they should be final.

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

Special Issue: 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.

```
1 ...
2         Double pi = new Double("3.14");
3 ...
```

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.

```
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.
- Good practice when it comes to concurrent programming.[22]

[22] 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          str1 = str1.toLowerCase();
8          System.out.println("str1 = " + str1);
9          System.out.println(str1 == str2); // output false?!
10         System.out.println(str1.intern() == str2); // output
               true
11 ...
```

Special Issue: enum Types[23]

- 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.

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

Example: Colors

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

- 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!

```
1  Class Pen {
2      Color color;
3      Pen(Color color) { this.color = color; }
4  }
5
6  Class Clothes {
7      Color color;
8      T_Shirt(Color color) { this.color = color; }
9      void setColor(Color new_color) { this.color = new_color; }
10 }
11
12 public class EnumDemo {
13     public static void main(String[] args) {
14         Pen crayon = new Pen(Color.RED);
15         Clothes T_shirt = new Clothes(Color.random());
16         System.out.println(crayon.color == T_shirt.color);
17     }
18 }
```

Exercise: Directions

```
1  enum Direction {UP, DOWN, LEFT, RIGHT}
2
3  /* equivalence
4  class Direction {
5      final static Direction UP = new Direction("UP");
6      final static Direction DOWN = new Direction("DOWN");
7      final static Direction LEFT = new Direction("LEFT");
8      final static Direction RIGHT = new Direction("RIGHT");
9
10     private final String name;
11
12     static Direction[] values() {
13         return new Direction[] {UP, DOWN, LEFT, RIGHT};
14     }
15
16     private Direction(String str) {
17         this.name = str;
18     }
19 }
20 */
```

Special Issue: 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**.

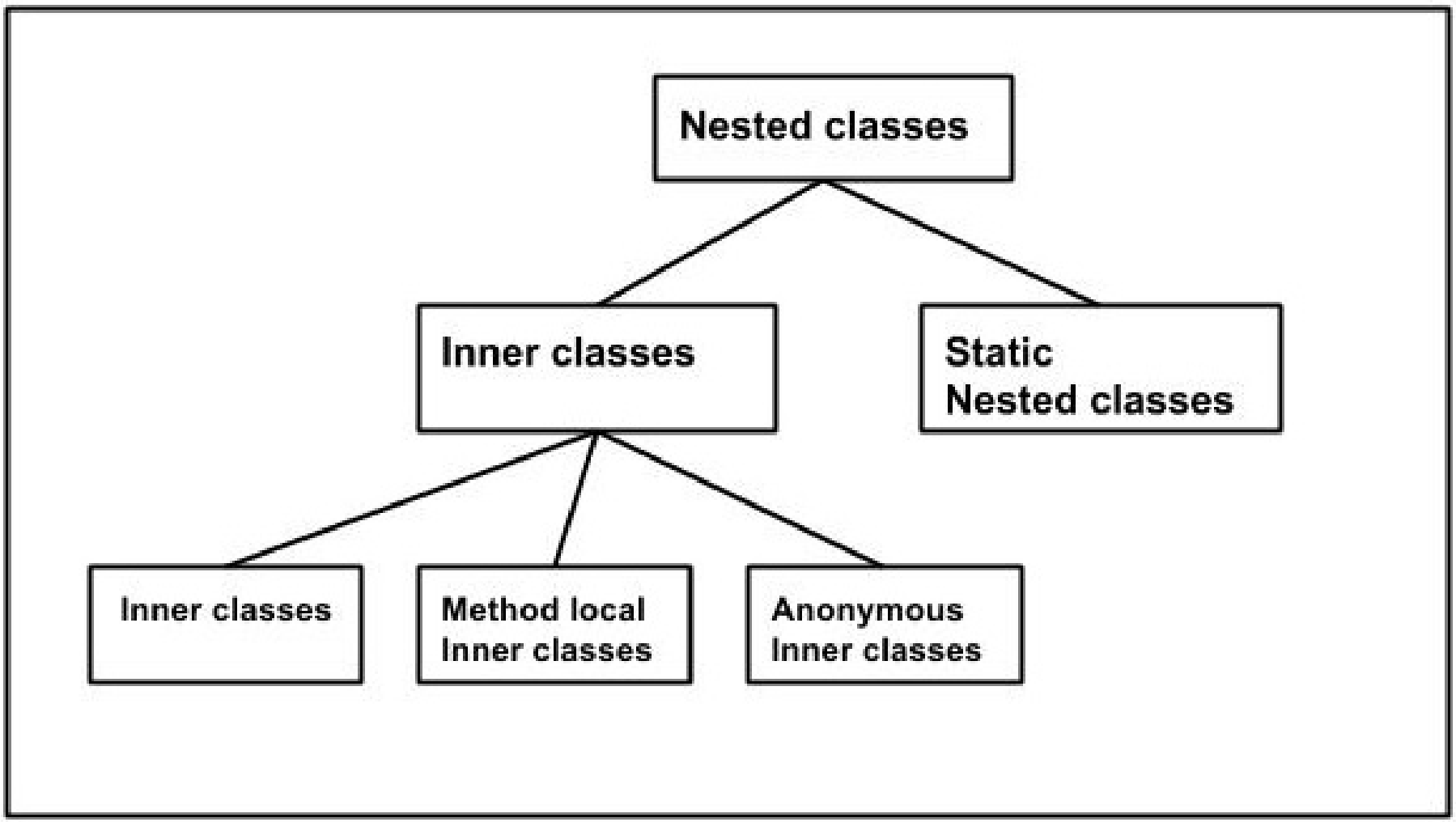
Access Control

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

Special Issue: 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

- 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.
- In the inner classes, you can declare final static variables but no static methods.

Example: Inner Class

```
1  class OuterClass {
2
3      private int x = 1;
4      private InnerClass innerObject = new InnerClass();
5
6      class InnerClass {
7          public void print() {
8              System.out.println(x); // ok!
9          }
10     }
11
12     void doSomeAction() { innerObject.print(); }
13 }
14
15 public class InnerClassDemo {
16     public static void main(String[] args) {
17         new OuterClass().doSomeAction(); // output 1
18
19         new InnerClass(); // you cannot do this
20     }
21 }
```

Example: Method-Local Inner Class

```
1  class OuterClass {
2
3      void doSomething() {
4          class LocalClass { // should be in the beginning
5              private int x = 2;
6              void print() { System.out.println(x); }
7          }
8
9          new LocalClass().print(); // output 1 and 2
10     }
11 }
```

Anonymous 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.
- Use them when you need to use these types only once.

Example: Button

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

Exercise: Let's Fly Again

```
1  interface Flyable {
2      void fly();
3  }
4
5  public class AnonymousClassDemoTwo {
6
7      public static void main(String[] args) {
8
9          Flyable butterfly = new Flyable() {
10             @Override
11             public void fly() { /* ... */ }
12         };
13
14         butterfly.fly();
15     }
16 }
```

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

Another Example: 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

```
1  import java.util.Iterator;
2
3  class Box implements Iterable<Integer> { // <...>: generics
4
5      int[] items = {10, 20, 30};
6
7      public Iterator<Integer> iterator() {
8          return new Iterator<Integer>() {
9              private int ptr = 0;
10
11             public boolean hasNext() {
12                 return ptr < items.length;
13             }
14
15             public Integer next() {
16                 return items[ptr++];
17             }
18         };
19     }
20 }
```

```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

```
1  class OuterClass {
2      static int x = 1;
3      private int y = 2;
4
5      static class StaticClass {
6          private int z = 3;
7          void doSomething() {
8              System.out.println(x);
9              System.out.println(y); // you cannot do this
10             System.out.println(z);
11         }
12     }
13 }
14
15 public class StaticNestedClassDemo {
16     public static void main(String[] args) {
17         new OuterClass.StaticClass().doSomething();
18     }
19 }
```

Classpath[24]

- 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).

[24] `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)[26]

- Jar is a packed format typically used to aggregate many Java class files, associated metadata[25] and resources (text, images, etc.) into one file to distribute the application software or libraries running on the Java platform.
  - Try an executable jar!

[25] Metadata refers data of data.

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