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)$^2$

- Java handles deallocation$^1$ 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.

---

$^1$Release the memory occupied by the unused objects.
$^2$http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html
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 ↔ student; doctor ↔ patient.
• If A uses B, then it is an **aggregation**, stating that B exists independently from A.
  • For example, knight ↔ sword; company ↔ 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 ↔ room.
Example: Lines

- **+2**: two **Point** objects used in one **Line** object.
public class Line {
    private Point head, tail;

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

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

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

- Circle, Triangle, and Polygon.
- Book with Authors.
- Lecturer and Students in the classroom.
- Zoo with many creatures, say Dog, Cat, and Bird.
- Channels 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

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); } // why?
    void writeCode() {}
}

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

    void watchDoor() {}
}
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()`.
See Fig. 3-1 in p. 113 of Evans and Flanagan.
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.
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.\(^4\)
  - 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`.

\(^4\)Notice that the static methods do not follow this rule.
Example

class Animal
  eat ()
  sleep ()
  reproduce ()

class Mammal
  reproduce ()

class Cat
  sleep ()
  huntMice ()
  purr ()

Cat Simon
  eat ()
  reproduce ()
  sleep ()
  huntMice ()
  purr ()
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.\(^5\)

If the compiler is not able to resolve the binding, such binding is known as **dynamic binding** or **late binding**.

- For example, method overriding.

\(^5\)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.\(^6\)
  
  • As you can see in Cat Simon.

• This is so-called **subtype polymorphism**.

\(^6\)An overridden method in Java acts like a virtual function in C++.
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.

Example: Uniform Interface

class Student {
    void doYourJob() { /* Do not know the detail yet. */ }
}

class HighSchoolStudent extends Student {
    void doHomework() {}

    @Override
    void doYourJob() { doHomework(); }
}

class CollegeStudent extends Student {
    void writeFinalReports() {}

    @Override
    void doYourJob() { writeFinalReports(); }
}
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;
    // enhance the flexibility by reducing dependency,
    // for future extension (scalability)
    public static void goStudy(Student s) {
        s.doYourJob();
    }

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

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

Zheng-Liang Lu
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.).$^8,^9$

---

$^8$See https://en.wikipedia.org/wiki/Liskov_substitution_principle.
$^9$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.

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

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

```
1 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.
```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!
    }
}
```
• 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.

```java
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"); }
}

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

---

10 The classes that sit near the bottom of the hierarchy are called `concrete` classes.
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 magicssssss");
    }

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

magic

air

some more magic

very important magic

air

magic
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();
    }
}
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**\textsuperscript{11} and **Serializable**\textsuperscript{12}.
• 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**.

\textsuperscript{11}See Java Multithread.
\textsuperscript{12}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 interface, not to implementation.**

---

13 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");
...
```
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>java.lang.Void</td>
</tr>
<tr>
<td>boolean</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>char</td>
<td>java.lang.Character</td>
</tr>
<tr>
<td>byte</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>short</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>int</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>long</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>float</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>double</td>
<td>java.lang.Double</td>
</tr>
</tbody>
</table>
Autoboxing and Unboxing of Primitives

- The Java compiler automatically wraps the primitives in their wrapper types, and unwraps them where appropriate.

```java
...  
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,

```java
... 
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.\(^\text{14}\)

• Another example is String objects.

enum Types

- An enum type is a 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.

\footnote{The keyword \texttt{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!
```java
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);
    }
}
```
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;
    }
}

Zheng-Liang Lu
Java Programming
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

<table>
<thead>
<tr>
<th>Scope</th>
<th>private</th>
<th>(package)</th>
<th>protected</th>
<th>public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within the class</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Within the package</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Inherited classes</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Out of package</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
</tbody>
</table>
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

- Nested classes
  - Inner classes
    - Inner classes
    - Method local Inner classes
    - Anonymous Inner classes
  - Static 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!
    }
}
}

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

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> { // <...> is generic

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

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

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

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

---

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)\textsuperscript{18}

- JAR is a packed format typically used to aggregate many Java class files, associated metadata\textsuperscript{17} and resources (text, images, etc.) into one file to distribute the application software or libraries running on the Java platform.
  - Try an executable JAR!

\textsuperscript{17}Metadata refers data of data.

\textsuperscript{18}See https://docs.oracle.com/javase/tutorial/deployment/jar/.