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

<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 into 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

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.
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++];
            }
        };
    }
}
```
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());
    }
}
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();
    }
}
```
Classpath\(^1\)

- 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).

\(^1\)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)³

- JAR is a packed format typically used to aggregate many Java class files, associated metadata² and resources (text, images, etc.) into one file to distribute the application software or libraries running on the Java platform.
  - Try an executable JAR!

---

²Metadata refers data of data.
³See https://docs.oracle.com/javase/tutorial/deployment/jar/.