In This Tutorial

• Explanation of the nested class concept.
• Access modifiers and nested classes.
• The types of nested classes in Java.
• Inheritance and inner classes.
• Nested classes in the JVM.
• Limits and caveats
The Concept

• Like C++, Java supports defining classes inside other classes. The general term for these classes is **nested classes**.
• Classes which are not defined inside other classes are called **top-level classes**.
• In Java, there are 4 types of nested classes.
Reminder: in Java there are 4 access control modifiers. Here’s a more accurate table of their effects:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Accessible from (in addition to defining class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Any class of any package</td>
</tr>
<tr>
<td>protected</td>
<td>Subclasses + any class of the same package</td>
</tr>
<tr>
<td>private</td>
<td>Only classes that share the same top-level class as the defining class</td>
</tr>
<tr>
<td>default</td>
<td>Any class of the same package</td>
</tr>
</tbody>
</table>

What does this mean about nested classes?
Access Control Example

- All classes here have access to private variables of all of the others.
  - Though E must have an instance of the other classes to access them.
Type 1 – Static Nested Classes

- Defined as member classes, with the `static` modifier.
- Most similar to nested classes in C++.
- Helps in encapsulation – for example, if `B` is useful only to `A`.

- A static nested class can have any visibility, just like any other member.
- Cannot access non-static members of enclosing class.

```java
class A {
  ...
  ...
  static class B {
    ...
  }
  ...
}
```
Inner Classes

• Nested classes which are not static are called **inner classes**.
• There are 3 kinds of inner classes:
  – non-static member class, a.k.a. (regular) inner class
  – local class
  – anonymous class
• An instance of an inner classes is associated with an instance of the enclosing class.
• Consequentially, inner classes can access **non-static** members of the enclosing class.
  • Also static members, of course.
More on Inner Classes

- Instances of enclosing / nested classes do not contain each other – they are only associated with each other.
- An instance of an enclosing class may be associated with multiple instances of inner classes.
- An instance of an inner class is associated with exactly one instance of the enclosing class.
Type 2 – Non-Static Member Classes

- Defined as member classes, **without** the static modifier.
- An instance of B is always associated with an instance of A.
- A.B can have any visibility.
- Created with qualified *new*

```java
class A {
    ...
    class B {
        ...
    }
    ...
}
```

A a = new A();
A.B ab = a.*new* B();

// ‘ab’ is associated with ‘a’
Access to enclosing class

• If B is an inner class of A, it may access the associated A instance’s this pointer via the qualified A.this
• This access is only available for non static nested classes (why?).

class A {
    private void foo() { System.out.println("A"); }
    class B {
        void foo() { System.out.println("B"); }
        void bar() {
            A.this.foo(); //prints A
            foo();         //prints B
        }
    }
    ...
Type 3 – Local Classes

• Defined inside a method, and only visible from the point it is declared to the end of the method.
• If defined inside a non-static method, is considered an inner class and has access to non-static members – otherwise, behaves more like static nested class.
• Can access parameters as well as local variables declared in the same method before the class is declared, but only as long as they are marked **final**.
class Outer {
    private int x;
    
    void f(final int z) {
        final int y = 4;
        Inner i = new Inner(); // Illegal - Inner is not declared
    
        class Inner {
            void inc() { x += z + y; } // Legal - y and z are final
        }
        Inner i = new Inner(); // Legal - inner is declared
    }
    
    static void g() {
        Inner i = new Inner(); // Illegal - Inner is not declared
    
        class Inner { // Legal - we can use the same name
            void inc() {
                x++; // Illegal - cannot access non-static field x
            }
        }
    }
}
Type 4 – Anonymous Classes

• Typically, only one instance of a local class is needed per method invocation.

• **Anonymous classes** are also created inside the body of a method, however
  – Defining the class also defines an instance of it (if any)
  – How can we define an anonymous class that has no running instances at all?
  – No more instances can be defined from that class

• Probably the most complicated of the nested class types, and probably the most useful.
Anonymous Classes - Syntax

```
new <parent class / interface>(<ctor arguments>) { 
    <anonymous class body>
}
```

• Simple example:

```java
interface A {
    void g();
}

class B {
    void f() {
        void a = new A() {
            @Override public void g() {} 
        };
        a.g();
    }
}
```
Anonymous Classes & Constructors

- Anonymous classes cannot have constructors* (they don’t even have a name).
- Constructors of a parent class can be invoked using the argument list.

```java
class A {
    public A(int x) {...}
    void g() {...}
}

class B {
    A f() {
        return (new A(3){...});
    }
}
```

* They can have instance initializers, though. Outside of our scope.
Anonymous Classes – Java 8

• Since Java 8, an interface with exactly one abstract method can be replaced with a lambda expression (can contain several default and or static methods).

• Very useful with creating an anonymous class's instance:

```java
public interface Predicate<T> { boolean test(T t);}

candidates.stream().filter(c -> c.getOOPGrade() > 90);

candidates.stream().filter(new Predicate<Person>(){
    @Override
    public boolean test(Person p) {
        return p.getOOPGrade() > 90;
    }
});
```

Old and obsolete!
Anonymous Classes – Java 8

- Since Java 8, an interface with exactly one abstract method can be replaced with a lambda expression (can contain several default and or static methods).
- Very useful with creating an anonymous class's instance:

```java
public interface Comparator<T> { int compare(T t1, T t2); }

candidates.stream().sort(
    (c1, c2) -> c2.getOOPGrade() - c1.getOOPGrade());

candidates.stream().filter(
    new Comparator<Person>(){
        @Override
        public int compare(Person p1, Person p2) {
            return p2.getOOPGrade() - p1.getOOPGrade();
        }
    });
```

Old and obsolete!
Anonymous Classes – Java 8

• It is recommended to add the `@FunctionalInterface` annotation* to the interface, to indicate it’s should to be replaced with a $\lambda$ expression:

```java
@FunctionalInterface
interface I {
    void foo(Person p1, Person p2);
    default int bar() {...};
    static void cool() {...};
}
```

* Annotations will be taught later on in this course.
# Nested Class Types Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Inner (associated with outer instance)</th>
<th>Definition point</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static nested class</td>
<td>No</td>
<td>As a member of another class.</td>
<td>Depends on access modifier.</td>
</tr>
<tr>
<td>Non-static member class (regular inner class)</td>
<td>Yes</td>
<td>As a member of another class.</td>
<td>Depends on access modifier.</td>
</tr>
<tr>
<td>Local class</td>
<td>Yes (if defined in non-static method)</td>
<td>Inside a method.</td>
<td>From the point it is defined to the end of the method.</td>
</tr>
<tr>
<td>Anonymous class</td>
<td>Yes (if defined in non-static method)</td>
<td>As an expression (since defining it also returns the instance).</td>
<td>None.</td>
</tr>
</tbody>
</table>
• It’s possible to inherit from an inner class, but only as long as the inheriting class can be associated with a subtype of the original outer class. Example of the legal scenarios:

class BaseOuter {
    class BaseInner {}
    // Legal because BaseOuter is a subtype of BaseOuter:
    class BaseInner2 extends BaseInner {}
}

class DerivedOut extends BaseOuter {
    // Legal because DerivedOut is a subtype of BaseOuter:
    class DerivedInner extends BaseInner {}
}
what would happen without this limitation?

```java
class Outer {
    int x;
    class Inner {
        void f() { x++; }
    }
}

class Derived extends Outer.Inner {}

void someMethod() {
    new Derived().f(); // What will happen?
}
```
Nested Classes and the JVM

• There is no concept of nested classes in bytecode or in the JVM. The compiler generates a separate .class file for each class.

• Since access modifiers are enforced in the JVM, the compiler sometimes has to work hard to split nested classes into independent classes.

```java
public class Outer {
    class Inner {
    }
}
```

> javac Outer.java
> ls
Outer.java
Outer.class
Outer$Inner.class
Inner Classes and the JVM

Likewise, any outer local variables used are passed in the constructor, which is why only final locals are permitted (what would happen otherwise?)

```java
public class Outer {
    class Inner {}
}
```

Inner actually compiles to something like:

```java
class Outer$Inner {
    Outer $this;
    Inner(Outer $this) {
        this.$this = $this;
    }
}
```
• Sun’s official tutorial at http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html