this, static & final

this Keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines the **this** keyword.
- **this** can be used inside any method to refer to the current (invoking) object.
- **this** is always a reference to the object on which the method was invoked.
- You can use **this** anywhere a reference to an object of the current class' type is permitted.
- In the body of the method, the **this** reference can be used like any other object reference to access members of the object.

this Keyword

- As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, you can have **local variables**, including **formal parameters** to methods, which overlap with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, the local variable **hides** the instance variable.

this Keyword

- Example:

```
class A {
        int i=10;
        void show()
        {
                int i=100;
                System.out.println("Local variable i: "+i);
                System.out.println("Instance variable i: "+i);

        }
        public static void main(String args []) {
                A obj = new A();
                obj.show();
        }

}
```

this Keyword

- Example:

```
class A {
        int i=10;
        void show()
        {
                int i=100;
                System.out.println("Local variable i: "+i);
                System.out.println("Instance variable i: "+this.i);

        }
        public static void main(String args []) {
                A obj = new A();
                obj.show();
        }

}
```

this Keyword

- Example:

```
class A {
        int i=10;
        void show()
        {
                int i=100;
                System.out.println("Local variable i: "+i);
                System.out.println("Instance variable i: "+this.i);

        }
        public static void main(String args []) {
                A obj = new A();
                obj.show();
        }

}
```

Output:

Local variable i: 100

Instance variable i: 10

this Keyword

- In particular, the this reference can be used explicitly to invoke other methods in the class.

this Keyword

- Example:

```
class A {
        void show()
        {
                System.out.println("In show() ");
                test();
        }
        void test()
        {
                System.out.println("In test() ");
        }
        public static void main(String args []) {
                A obj = new A();
                obj.show();
        }
}
```

this Keyword

- Example:

```
class A {
        void show()
        {
                System.out.println("In show() ");
                test();
        }
        void test()
        {
                System.out.println("In test() ");
        }
        public static void main(String args []) {
                A obj = new A();
                obj.show();
        }
}
```

Output:

```
In show()
In test()
```

this Keyword

- Example:

```
class A {
        void show()
        {
                System.out.println("In show() ");
                this.test();
        }
        void test()
        {
                System.out.println("In test() ");
        }
        public static void main(String args []) {
                A obj = new A();
                obj.show();
        }
}
```

this Keyword

- Example:

```
class A {
        void show()
        {
                System.out.println("In show() ");
                this.test();
        }
        void test()
        {
                System.out.println("In test() ");
        }
        public static void main(String args []) {
                A obj = new A();
                obj.show();
        }
}
```

Output:

In show()

In test()

this Keyword

- In particular, the this reference can be used explicitly to invoke other methods in the class.
- If, for some reason, a method needs to pass the current object to another method, it can do so using the **this** reference.
- Another use of **this** keyword to call the other constructor of the same class.
- Note that the **this** reference cannot be used in a static context, as static code is not executed in the context of any object.

Understanding static

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- When a member is declared static, it can be accessed *before any objects of its class are created*, and *without reference to any object*.
- You can declare both *methods* and *variables* to be static.
- When objects of its class are declared, **no copy** of a static variable is made. Instead, all instances of the class **share the same static variable**.
- A static variable belongs to the class, and not to any object of the class.
- A static variable is initialized when the class is loaded at runtime.
- Similarly, a class can have static methods that belong to the class, and not to any specific objects of the class.

Understanding static

- Static members can also be accessed via object references, but this is considered bad style.
- Static members in a class can be accessed both by the class name and via object references, but instance members can only be accessed by object references.
- Java supports **static block** that gets executed exactly once, when the class is first loaded.
- Methods declared as static have several restrictions:

✓They can only directly call other static methods.

✓They can only directly access static data.

✓They cannot refer to **this** or **super** in any way.

Understanding static

- Example:

```
class A {
        static int a = 5;
        static int b;
        static void show(int x) {
                System.out.println("x = " + x);
                System.out.println("a = " + a);
                System.out.println("b = " + b);
        }
        static {
                System.out.println("Static block initialized.");
                b = a * 5;
        }
        public static void main(String args []) {
                show(10);
        }
}
```

Understanding static

- Example:

```
class A {
        static int a = 5;
        static int b;
        static void show(int x) {
                System.out.println("x = " + x);
                System.out.println("a = " + a);
                System.out.println("b = " + b);
        }
        static {
                System.out.println("Static block initialized.");
                b = a * 4;
        }
        public static void main(String args []) {
                show(10);
        }
}
```

Output:

```
Static block initialized.
x = 10
a = 5
b = 20
```

Understanding static

- As soon as class A is loaded, all of the static statements are run.
- First, **a** is set to **5**, then the static block executes, which prints a message and then initializes **b** to **a*4** or 20.
- Then main( ) is called, which calls show( ), passing 10 to x.
- The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Understanding static

- Outside of the class in which they are defined, static methods and variables can be used independently of any object.
- To do so, you need only specify the name of their class followed by the dot operator.

classname.method( )

- Here, *classname* is the name of the class in which the static method is declared.
- A static variable can be accessed in the same way—by use of the dot operator on the name of the class.

Understanding static

- Example:

```
class A {
        static int a = 5;
        static int b=10;
        static void show() {
        System.out.println("a = " + a);
        }
}
class B
{
        public static void main(String args []) {
                A.show();
                System.out.println("b = " + A.b);
        }
}
```

Understanding static

- Example:

```
class A {
        static int a = 5;
        static int b=10;
        static void show() {
                System.out.println("a = " + a);
        }
}
class B
{
        public static void main(String args []) {
                A.show();
                System.out.println("b = " + A.b);
        }
}
```

Output:

a = 5

b = 10

Understanding static

Instance Members	These are instance variables and instance methods of an object. They can only be accessed or invoked through an object reference.
Instance Variable	A field that is allocated when the class is instantiated, i.e., when an object of the class is created. Also called *non-static field*.
Instance Method	A method that belongs to an instance of the class. Objects of the same class share its implementation.
Static Members	These are static variables and static methods of a class. They can be accessed or invoked either by using the class name or through an object reference.
Static Variable	A field that is allocated when the class is loaded. It belongs to the class and not to any specific object of the class. Also called *static field* or *class variable*.
Static Method	A method which belongs to the class and not to any object of the class. Also called *class method*.

Understanding final

- Members of the class can be declared as final.
- Instance and static variables can be declared final.
- Note that the keyword final can also be applied to local variables, including method parameters.
- In addition to fields, both method parameters and local variables can be declared final.
- Final static variables are commonly used to define constants.
- Declaring a parameter final prevents it from being changed within the method.
- Declaring a local variable final prevents it from being assigned a value more than once.
- There are two ways to initialize final variables.
  - ✓First, you can give it a value when it is declared.
  - ✓Second, you can assign it a value within a constructor.

Understanding final

- A final variable of a reference type cannot change its reference value once it has been initialized.
- This effectively means that a final reference will always refer to the same object.
- A final method in a class is complete and cannot be **overridden** in any subclass.
- Final variables ensure that values cannot be **changed** and final methods ensure that behavior cannot be changed.
- The compiler may be able to perform code optimizations for final members, because certain assumptions can be made about such members.

Understanding final

- Sometimes you will want to prevent a class from being inherited.
- To do this, precede the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final, too.
- As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.