Chapter 10

static and final

- Consider the Math class in Java. Do you find anything in particular?

```
public static double toRadians(double angdeg) {
    return angdeg / 180.0 * PI;
}
```

```
public static double toDegrees(double angrad) {
    return angrad * 180.0 / PI;
}
```

```
public static int max(int a, int b) {
    return (a >= b) ? a : b;
}
```

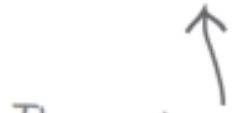
```
public static int min(int a, int b) {
    return (a <= b) ? a : b;
}
```

```
public static int abs(int a) {
    return (a < 0) ? -a : a;
}
```

These methods only considers their input arguments!

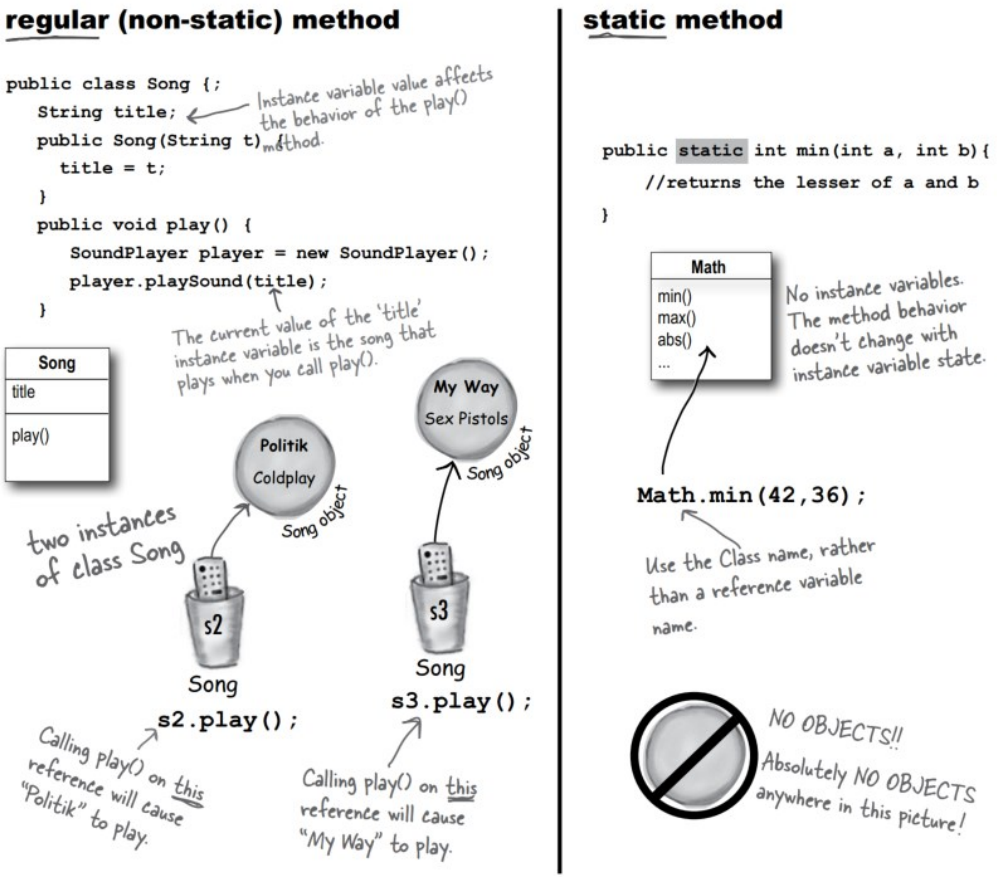
- Methods in the Math class never use instance variable values. So there's nothing to be gained by making an instance of class Math.
- When you want to use these methods, you don't have to create an instance of Math class. As a matter of fact, you can't do that.
- These methods are all **static** methods, you don't need to have an instance of Math. All you need is the Math class

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-343);
```



These methods never use instance variables, so their behavior doesn't need to know about a specific object.

- When there is no need to have an instance of the class, the keyword **static** lets a method run without any instance of the class.
- A static method means "**behavior not dependent on an instance variable**, so no instance/object is required. Just the class."



- Static methods run without knowing about any particular instance of the static method's class.
- Static methods can't use non-static (instance) variables or methods.
- The following code will not work because the static method **main()** does not know about the instance variable 'size'.

```
public class Duck {

   private int size;

   public static void main (String[] args) {
      System.out.println("Size of duck is " + size);
   }

   public void setSize(int s) {
      size = s;
   }
   public int getSize() {
       return size;
   }
}
```

Which Duck? Whose size?

If there's a Duck on the heap somewhere, we don't know about it.

- Can I call a static method using a reference variable instead of the class name?

```
Duck d = new Duck();
String[] s = {};
d.main(s);
```

- This code is legal, but that just because the compiler resolves it back to the real class.
- In other words, using variable **d** to invoke main() doesn't imply that main() will have any special knowledge of the object that **d** is referencing.

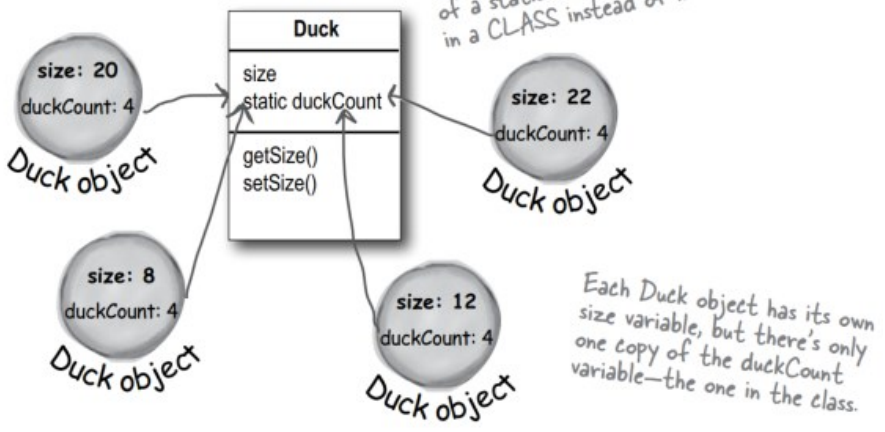
- instance variables: 1 per **instance**
- static variables: 1 per **class**
- Static variables in a class are initialized before any object of that class can be created.
- Static variables in a class are initialized before any static method of the class runs
- Default values for static variables is the same as that for instance variables.
- Quiz 1: a static method can't access a non-static variable. But can a non-static method access a static variable?
- Quiz 2: can an abstract class have static variables?

- **Static variable**: value is the same for ALL instances of the class
- Imagine you wanted to count how many Duck instances are being created while your program is running. How would you do it?
- If you use a `normal' instance variable, its value resets every time you create a new object.
- You need a class that's got only a single copy of the variable, and all instances share that one copy.

```
public class Duck {

   private int size;
   private static int duckCount = 0;

   public Duck() {
       duckCount++;
   }

   public void setSize(int s) {
      size = s;
   }
   public int getSize() {
      return size;
   }
}
```

The static duckCount variable is initialized ONLY when the class is first loaded, NOT each time a new instance is made.

Now it will keep incrementing each time the Duck constructor runs, because duckCount is static and won't be reset to 0.

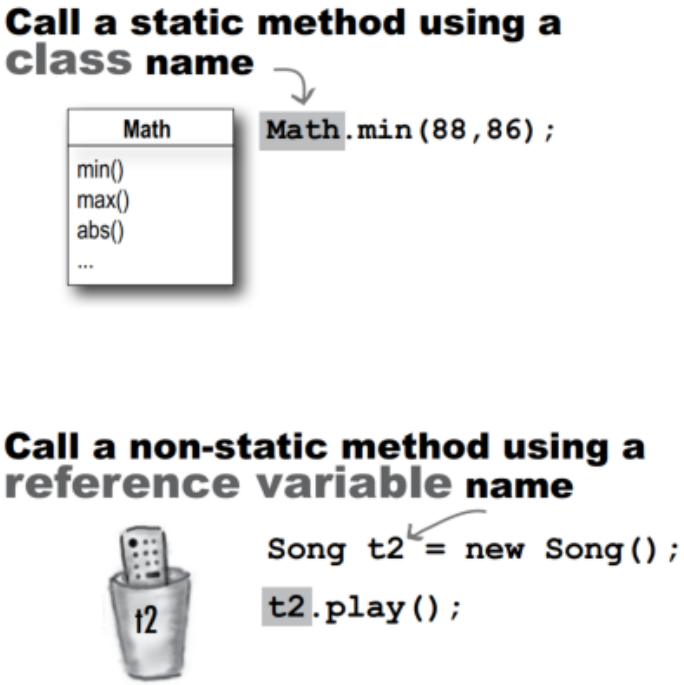


- So that means I can still create an object then call its static methods. Can I create a Math object and call the random() method?

```
Math m = new Math();
double num = m.random(5);
```

- No, you can't.
- Why I cannot instantiate a Math object? (Is it abstract?)
- No, it is not abstract but it has a ***private*** constructor.
- Because the constructor is private, you cannot see it from outside.

- Remember, a method marked private means that only code from within the class can invoke the method.
- If we mark a constructor as **private**, it means essentially the same thing—only code from within the class can invoke the constructor. Nobody can say 'new' from outside the class. That's how it works with the Math class



- What if you want your class to have a fixed value (like a constant), no matter how many times it is subclassed?
- The keyword **final** can give you that.
- A variable marked **final** means that—once initialized—it can never change. In other words, the value of the static final variable will stay the same as long as the class is loaded.
- Look up Math.PI in the API, and you'll find:

```
public static final double PI = 3.141592653589793;
```

- The variable is marked public so that any code can access it.
- The variable is marked static so that you don't need an instance of class Math (which, remember, you're not allowed to create).
- The variable is marked final because PI doesn't change (as far as Java is concerned).
- There is no other way to designate a variable as a constant, but there is a naming convention that helps you to recognize one: ***constant variable names should be in all caps!***

Initialize a *final* static variable:

① **At the time you declare it:**

```
public class Foo {
   public static final int FOO_X = 25;
}
```

notice the naming convention -- static final variables are constants, so the name should be all uppercase, with an underscore separating the words

OR

② **In a static initializer:**

```
public class Bar {
   public static final double BAR_SIGN;

   static {
        BAR_SIGN = (double) Math.random();
   }
}
```

this code runs as soon as the class is loaded, before any static method is called and even before any static variable can be used.

If you don't give a value to a final variable in one of those two places:

```
public class Bar {
   public static final double BAR_SIGN;
}
```

no initialization!

The compiler will catch it:

```
File Edit Window Help Jack-in
% javac Bar.java
Bar.java:1: variable BAR_SIGN
might not have been initialized
1 error
```

- final isn't just for static variables
- You can use the keyword final to modify non-static variables, methods, even a class.
- So final can be interpreted as `cannot change'.
- A final variable means you can't change its value.
- A final method means you can't override the method.
- A final class means you can't extend the class (i.e. you can't make a subclass)
- Quiz 1: can you set value to a final variable via a constructor?
- Quiz 2: can you extend a class who only has private constructors?

- Quiz 1: can you set value to a final variable via a constructor?
- Ans: yes you can. The keyword **final** means that you can only set this variable ***once***. There are two ways to do that:

1. In the constructor
2. When you declare it

- Note: you CANNOT do both

```
public class Test
{
    final int a;

    public Test()
    {
        a = 10;
    }
}
```

```
public class Test
{
    final int a = 10;

    public Test(){}
}
```

Q: If you do not set value to a final variable in a constructor, can you set it afterwards?

- Quiz 2: can you extend a class who only has private constructors?
- Ans: you can't extend the parent class if it has a private default constructor (because you can’t see them!). So usually we also mark it as a **final** class.
- In fact, a better question is "should we extend a class having only a private constructor?“
- A class which has private constructors is usually a **utility** class or a **singleton**

- What is a singleton?
- Singleton means "only one". Each class can create one and only one instance no matter how many times it is used.
- The trick is: you can create an instance by yourself and provide a method to let everyone use this instance
- Make this class with a private constructor so that nobody can instantiate it

```
class Puppy {

    /* create a puppy instance when class is loaded.*/
    private static Puppy aDog = new Puppy();
    private int age = 8;

    /* private constructor. No others can create Puppy. */
    private Puppy(){}

    public static Puppy getInstance() {
        return aDog;
    }

    public int getAge() {
        return age;
    }
}
```

this instance is created when your program is started.

- Why would I want to make a class final?
- For security. For example, no one should make a subclass of the String class or the Math class
- Quiz: Should I mark the methods final at the same time if the class is a final class?
- You don't have to. A final class cannot be subclassed so no methods will be overridden.

```
public final class Math {

    /**
     * Don't let anyone instantiate this class.
     */
    private Math() {}

    /**
     * The {@code double} value that is closer than any other to
     * <i>e</i>, the base of the natural logarithms.
     */
    public static final double E = 2.7182818284590452354;

    /**
     * The {@code double} value that is closer than any other to
     * <i>pi</i>, the ratio of the circumference of a circle to its
     * diameter.
     */
    public static final double PI = 3.14159265358979323846;
```

- Summary
- A static method should be called using the class name `Math.random()`
- A static method can be invoked without any instances of the method's class on the heap
- A static method cannot access instance variables.
- If you don't want the class to be instantiated, mark the constructor as private.
- A static variable is a variable shared by all members of a given class
- To make a constant in Java, mark a variable as both static and final
- The naming convention for constants is to make the name all uppercase.
- A final method cannot be overridden.
- A final class cannot be extended.

- Exercise: which of these are legal?

①
```
public class Foo {
   static int x;

   public void go() {
      System.out.println(x);
   }
}
```

②
```
public class Foo2 {
  int x;

  public static void go() {
      System.out.println(x);
  }
}
```

③
```
public class Foo3 {
  final int x;

  public void go() {
      System.out.println(x);
  }
}
```

④
```
public class Foo4 {
  static final int x = 12;

  public void go() {
      System.out.println(x);
  }
}
```

⑤
```
public class Foo5 {
  static final int x = 12;

  public void go(final int x) {
      System.out.println(x);
  }
}
```

⑥
```
public class Foo6 {
  int x = 12;

  public static void go(final int x) {
      System.out.println(x);
  }
}
```

- More statics... static imports
- Static imports exist only to save you some typing
- The basic idea is that whenever you're using a static class, a static variable, etc., you can import them, and save yourself some typing.

Some old-fashioned code:

```
import java.lang.Math;
class NoStatic {
  public static void main(String [] args) {
    System.out.println("sqrt " + Math.sqrt(2.0));
    System.out.println("tan " + Math.tan(60));
  }
}
```

Same code, with static imports:

```
import static java.lang.System.out;
import static java.lang.Math.*;
class WithStatic {
  public static void main(String [] args) {
    out.println("sqrt " + sqrt(2.0));
    out.println("tan " + tan(60));
  }
}
```

The syntax to use when declaring static imports.

Static imports in action.

Use Carefully: static imports can make your code confusing to read

- Static block
- Static block is the group of statements that gets executed when the class is loaded
- Mostly static block is used to create static resources when the class is loaded.

```
public class myStatic
{
    private static int x;

    // static block
    static {
        System.out.println("static block starts.");
        // can access only static variables and methods
        x = 20;
    }

    private myStatic(){}

    public static void setX(int y)
    {
        x = y;
    }

    public static void printOut()
    {
        System.out.println("value is"+x);
    }
}
```

- Some notices
- Static methods can NOT be overridden. Because with method overriding, which implementation shall be executed is decided at runtime. That is, the JVM must known which object is calling the method. But static methods can be overloaded.
- However, the compiler will not complain if you override static methods.
- A static method is implicitly regarded as a final method

```
class ClassA{
    public static void foo(){System.out.println("from A");}
}

class ClassB extends ClassA{
    public static void foo(){System.out.println("from B");}
}

public class testFinal
{
    public static void main(String[] args){

        ClassA a = new ClassB();  // polymorphism

        /* output is "from A" because this statement is resovled
         * as  ClassA.foo();
         */
        a.foo();
    }
```

- As mentioned before, an interface contains abstract methods. However, an interface can also contains "static" things.
- You can declare static variable and static method in an interface, but we rarely put variables in an interface.
- Quiz: can you declare static variable in a method?
- Ans: in Java, you cannot declare static variable in a method. The question is: how long do you want your 'static within a method' variable to last?

* C++ allows static variables within a method and it acts like the class variables. People rarely use that because it's quite confusing.

- <Extension>
- In Java, if you want to have some global constant values, typically you can do it in this way:
- Before Java 1.4, we create a public final class which has a private constructor and some final static variables.

```
public class Actions
{
    public static final int UP=0;
    public static final int DOWN=1;

    public static final String MSG="hello!";
}
```

- Putting static variables in an interface is also allowed but it is a BAD idea.

- After Java 5, if you only want to define constants (no methods are involved), you can use the **enum** (enumerate) type
- An enum is a special "class" that represents a group of constants, it inherits the abstract class java.lang.Enum (which inherits the Object class). Hence, it has default behaviors for values() or toString() methods.

```
public enum Actions
{
    UP, DOWN, MSG;
}
```

- One advantage of enum is that you can pass the enum as method parameters. By doing so, developers cannot pass values which is not defined in that enum class.

- Example of enum

```
enum Actions {
    LEFT, RIGHT, FORWARD
}

public class CarTest {

    public void go(Actions act) {
        if (act == Actions.LEFT) {
            System.out.println("go left");
        }
        else if (act == Actions.RIGHT) {
            System.out.println("go right");
        }
        else if (act == Actions.FORWARD) {
            System.out.println("go forward");
        }
        else {
            System.out.println("stop");
        }
    }
```