

CSCI 132:

Basic Data Structures and Algorithms

Static methods, Abstract Classes, Interfaces

Reese Pearsall
Spring 2023

Static methods are methods in Java that can be called without creating an object of a class

```
public class StaticDemo {  
    public static void fun1(String arg1) {  
        System.out.println(arg1);  
    }  
    public static void main(String[] args) {  
        fun1("Hello");  
    }  
}
```

I do **not** need to create a `StaticDemo` object in order to call the `fun1()` method



Static methods are methods in Java that can be called without creating an object of a class

```
public class StaticDemo {  
    public static void main(String[] args) {  
        AnotherClass.funMethod("Hello");  
    }  
}
```

StaticDemo.java

```
public class AnotherClass {  
    public static void funMethod(String arg)  
    {  
        System.out.println(arg);  
    }  
}
```

AnotherClass.java

If the static method is in another class, we can access it by giving the class name (`AnotherClass`)

Once again, I do not need to create an `AnotherClass` object to call this static method

However, now objects are no longer an implicit argument to this method (cant use `this` anymore)

Static methods are methods in Java that can be called without creating an object of a class

Error: static method cannot be referenced from a non static context

✗ `funMethod("Hello");`

This is a very common error to see in Java.


- You can turn the method static by adding the static keyword in the method definition *(Easy and quick fix)*
- Or you use OOP and call the method on an instance of the class

```
AnotherClass obj = new AnotherClass()  
obj.funMethod("Hello")
```


*(Usually this is the better solution
80% of the time)*

Abstract Classes are restricted classes that cannot be used to create objects. To access it, it must be inherited from another class.

```
public abstract class Employee {  
    ...  
}
```

 `Employee e = new Employee("Sally", 4444, 123456);`

You **cannot** create instances of an abstract class.

`Accountant kevin = new Accountant("Kevin Malone", 4444, 42000, 'C');` 

Instead, we use objects from another class *that inherits from the abstract class*

Abstract Classes are restricted classes that cannot be used to create objects. To access it, it must be inherited from another class.

Why are abstract classes helpful?

This is helpful if we want to use inheritance, but we *don't* want users to be able to create instances of the parent class

(ie create shared functionality, but not allow to create instances of just the shared functionality)

The shared functionality is nothing on its own, and is used solely for subclassing purposes

Interfaces are abstract classes that only contain methods with no body

```
public interface Vehicle {  
    void accelerate(int a);  
    void slowdown(int a);  
    void refuel(int a);  
}
```

Accelerate, Slow down, and refuel are all common behavior that all vehicles will have

However, the specifics of *how* they accelerate, slow down, refuel will be different between vehicles (ie the body of the methods will be slightly different)

Interfaces can be used to specify what a class *must do*, but not *how*

Interfaces are abstract classes that only contain methods with no body

```
public interface Vehicle {  
    void accelerate(int a);  
    void slowdown(int a);  
    void refuel(int a);  
}
```

```
public class Ferrari implements Vehicle {  
}
```

For a Java class to use an interface, it must use the `implements` keyword

We can implement multiple interfaces (unlike inheritance)

Interfaces are abstract classes that only contain methods with no body

```
public interface Vehicle {  
    void accelerate(int a);  
    void slowdown(int a);  
    void refuel(int a);  
}
```

Now, any Class that also has the behavior of `accelerating`, `slowdown`, and `refuel` can implement our interface, and those classes are **forced** to write the body of the methods

```
public class Ferrari implements Vehicle {  
  
    @Override  
    public void accelerate(int a) {  
        ...  
    }  
  
    @Override  
    public void slowdown(int a) {  
        ...  
    }  
  
    @Override  
    public void refuel(int a) {  
        ...  
    }  
}
```

The code of the method body is omitted, but that is where the programmer can put the specific behavior of:

- how a Ferrari will accelerate
- how a Ferrari will slow down
- how a Ferrari will refuel

Interfaces are abstract classes that only contain methods with no body

```
public interface Vehicle {  
    void accelerate(int a);  
    void slowdown(int a);  
    void refuel(int a);  
}
```

You can not create an instance of an interface

In the interface, the method bodies must be empty

- (Remember, the classes that *use* our interface will have the method bodies)

```
public class Ferrari implements Vehicle {  
  
    @Override  
    public void accelerate(int a) {  
        ...  
    }  
  
    @Override  
    public void slowdown(int a) {  
        ...  
    }  
  
    @Override  
    public void refuel(int a) {  
        ...  
    }  
}
```

The code of the method body is omitted, but that is where the programmer can put the specific behavior of:

- how a Ferrari will accelerate
- how a Ferrari will slow down
- how a Ferrari will refuel

Interfaces are abstract classes that only contain methods with no body

Why use interfaces?

Interfaces are great when you need **multiple implementations** of the **same behavior**

It forces classes to implement X methods that might not logically belong to them (*more control*)

It provides **abstraction**
(ie the details of how things are implemented are not revealed in an interface)