Interfaces

What Are Interfaces?

- a class can only extend one class
- but what if we want to "extend" more (abstract) classes?
- then we use **interface**
  - similar to abstract class
  - but now one class can **implement** any number of interfaces
  - keyword `implements`, separated by comma (`,`)
- like in abstract classes, you cannot create an instance of the interface using keyword `new`

```
public interface Car {
  int distanceWithFullTank (int tankVolume);  abstract method (no body!)
  int MAXIMUM_WEIGHT = 2000;  constant field
}

// there are some implicit keywords here
public abstract interface Car {
  public abstract int distanceWithFullTank (int tankVolume);
  public static final int MAXIMUM_WEIGHT = 2000;
}

// all interfaces are implicitly abstract, so they cannot be marked as final
```

```
public interface Car {
  int distanceWithFullTank (int tankVolume);
  int MAXIMUM_WEIGHT = 2000;
}
public interface Ford {
  String getColor();
}
public class fordModelT implements Car, Ford {
  public int distanceWithFullTank (int tankVolume) {
    return tankVolume * 9;
  }
  public String getColor() {
    return "black";
  }
}
```

abstract methods

implementations

Rules of Implementation

1. Keyword `public` is required
2. Return type must be covariant with the interface method
3. Signature (name & parameters) must match the interface method
4. All inherited methods must be implemented

overriding rules

```
// an interface can extend another interface
public interface canSwim extends Mammal {
  public int swim();
}

public interface Mammal {
  public int eats();
}
```

Elephant must implement both swim() and eats()

```
public class Elephant implements canSwim {
  public int swim() { return 5; }
  public int eats() { return -1; }
}
```

```
// duplicate methods, example #1
public interface Tetrapod {
  public void eats();
}
public interface Mammal {
  public void eats();
}
public class Dog implements Tetrapod, Mammal {
  public void eats() {
    System.out.println("Dog is eating.");
  }
}
```

this is OK, because the return types match (covariant or same)

```
// duplicate methods, example #2

public interface Tetrapod {

  public void eats();

}

public interface Mammal {

  public int eats();

}

public class Dog implements Tetrapod, Mammal {

  public void eats() {

    System.out.println("Dog is eating.");

  }

}
```

this does not compile (non-covariant return types)

another way to think about this is

that `int eats()` is not implemented

Default Interface Methods

- imagine you have an interface which is implemented by 100 classes
- for some reason you need to add a another method in your interface
- if this method were abstract, you would need to implement it in all 100 classes
- this is solved by making a method `default` (non-abstract)
- default method must have a body (default implementation)

```
public interface Mammal {
  public void walks();
  public void eats();
  default void sleeps() {
      System.out.println("A mammal sleeps.");
  }
}
public class Dog implements Mammal {
  public void walks() { System.out.println("Dog walks."); }
  public void eats() { System.out.println("Dog eats."); }
}
public class Cat implements Mammal {
  public void walks() { System.out.println("Cat walks."); }
  public void eats() { System.out.println("Cat eats."); }
}
```

no need to implement `sleeps()` in classes which implement `Mammal`

Rules for Using Default Methods

1. Keyword `default` with a method can only be used in the interface
2. Has to have a body (default implementation)
3. Implicitly `public`
4. Cannot be `abstract`, `final` or `static`
5. May or may not be overriden by a class implementing the interface
6. If the class inherits two or more default methods with the same method signature, then it must override the method

```
public interface Car {
  public default int getMaxSpeed() { return 100; }
}
public interface Truck {
  default int getMaxSpeed() { return 70; }
}
public class Van implements Car, Truck {
  public int getMaxSpeed() {
    return 80;
  }
  public int getMaxSpeedCar() {
    return Car.super.getMaxSpeed();
  }
}
```

we must override `getMaxSpeed()`

because there are two of them with a same signature

this is how we call the default method

from `Car` interface

```
// static interface methods
public interface Car {
  static int getMaxSpeed() { return 100; }
}
```

static methods cannot be overriden!

```
public class Ford implements Car {
  // I want to access getMaxSpeed() from Car
  public int getMaxSpeedCar() {
    return Car.getMaxSpeed();
  }
}
```

```java
// private interface methods
public interface Car {
  private static int calculateSpeed() {
    int speed = 70 * 2;
    return speed;
  }
  public default int getMaxSpeed() {
    return calculateSpeed();
  }
  public default int getRecommendedSpeed() {
    return (int)(calculateSpeed() * 0.8);
  }
```

Rules for Using Private Interface Methods

1. marked with keyword `private`
2. must have a body
3. `private static` methods may be called by **any** method in the interface
4. non-static `private` methods may be called only by non-static methods

```
// default and private non-static methods can call abstract methods!
public interface Car {
  int getMaxSpeed();
  int getHorsePower();
  default void printCarFeatures() {
     System.out.println("Max speed: " + getMaxSpeed() +
                        " | Horse power: " + getHorsePower() );
  }
}
```

these methods are abstract and they have to be implemented

=> when you call `printCarFeatures()` from a class which implements the interface, the implementation given in that class will be used in `printCarFeatures()`