Interfaces

- Interfaces contain only abstract methods (even though they are not declared as abstract)
- Interfaces are implemented by a class, not extended
- By implementing an interface, a class must implement all the methods in the interface
- It's like signing a contract promising to implement certain methods

```
public interface Animal {
    void eat();
    void sleep();
}

public class Dog implements Animal {
    // has to implement eat() and sleep()
}
```

Interfaces vs Abstract classes

- Interfaces are almost like abstract classes with only abstract methods
- Abstract classes can have implementations of methods, interfaces can only have abstract methods*
- A class can implement many interfaces, but it can only extend one abstract superclass

* with some exceptions in Java

Interfaces as reference types

- Interface is a reference type
- The type of the object reference can be an interface if the object implements that interface

```
public interface Animal {
    void eat();
}

public class Dog implements Animal {
       public void eat() {
         System.out.println("dog is eating");
    }
}

Animal animal = new Dog(); // interface as the reference type
```

Interfaces as method arguments and return types

- Interfaces can also be used as method argument types and return types

```
public void feedAnimal(Animal animal) { // interface as method argument
    animal.eat();
}

public Animal getNewDog() { // interface as return type
    return new Dog();
}
```

Changes to Interfaces in Java 8 and Java 9

- In Java 8 Interfaces can have default methods with implementations
- Default methods was introduced to be able to add methods to standard Java interfaces that was already used in codebases all over the world without breaking existing code
- In Java 8 Interfaces can also have static methods
- In Java 9 Interfaces can have private methods
- But this is an advanced topic and not really the point of using interfaces

Loose coupling

- Coupling refers to the degree of direct knowledge that an object has of other objects
- Tight coupling means that changes to one object can force changes in other objects
- Loose coupling means the objects are mostly independent
- Tight coupling reduces flexibility and reusability
- We like loose coupling!
- Interfaces facilitate loose coupling
- By using interfaces as reference types, the code that uses the reference types becomes loose coupled to the actual object class type

Loose coupling

```
public void feedAnimal(Animal animal) { // interface as method argument
    animal.eat();
}
// The method doesn’t know what the type of the object really is.
// It can handle any object that implements the Animal interface.
// It doesn’t know and it doesn’t care! This is loose coupling!
// The method could be used like this:
feedAnimal(new Cat()); // if Cat class implements Animal
feedAnimal(new Dog()); // if Dog class implements Animal
feedAnimal(new Horse()); // if Horse class implements Animal
```