Summary: This topic focuses on interfaces in Java and the role they play as compared to classes, including abstract classes. Ways for emulating multiple inheritance are also considered, since the lack of this feature is one of the original motivations for interfaces in Java.
Interface fundamentals

An interface in Java is just a specification of the external interface for an object.

It is essentially just a list of declarations (or signatures) for the methods of the object, including their types.

- The methods are implicitly public, and cannot be static or final.

Constant declarations are also allowed because sometimes there are constants that all classes implementing an interface will need.

- The constants are implicitly public, static and final.

```java
public interface BoardPiece {
    // Implicitly public static final
    int boardSize = Board.boardSize;

    // Implicitly public, never static or final.
    void goNorth();
    void goSouth();
    void goEast();
    void goWest();
}
```
Interfaces and classes

In Java it is not necessary to define an explicit interface for every class that you write.

- Classes already contain signatures for methods.
- You can use `public`, `private`, etc., to specify which methods are visible in the external interface of objects.

In a sense, every class automatically defines an interface as well.

- This is very convenient for the common case where the class is the only one implementing the interface.
- A class then has both signatures and implementations of methods.

What about abstract classes?

- Abstract classes have signatures for all methods.
- They also may have implementations for some methods.
- An interface is thus just like an abstract class with no methods implemented.
- Abstract classes thus can range from being only an interface, to including all implementations.
- Why are interfaces useful if you can use abstract classes with no implementations instead? (C++ only has abstract classes.)
Why does Java include Interfaces?

There are a number of reasons why you might want to have a way to define interfaces separately from abstract classes.

- Having a special way to define only interfaces makes it very clear that no implementations are provided.

- With an abstract class, you have to check the class itself to see if it contains implementations of methods.

- Having separate interfaces encourages the programmer to think about interfaces separately from implementations.

- It allows Java to emulate common uses of multiple inheritance for interfaces while avoiding some tricky issues when implementations are involved also.

The last of these is in fact the main original motivation for including interfaces in Java.
An example for multiple inheritance

Suppose we want to implement are implementing a simple banking system using objects.

- We define classes for savings accounts, and for checking accounts that inherit from a generic account class.
- Savings accounts have a method to add interest.
- Cheque accounts can process cheques, and have tax on deposits.

```java
public class Account {
    private int balance; // in cents
    public int getBalance() { return getBalance(); }  
    public void deposit(int amount)
        { balance += amount; }
    public void withdraw(int amount)
        { balance -= amount; }
}

public class SavingsAccount extends Account {
    private int interestRate=6; // Percentage
    public void addInterest() {
        deposit( getBalance()*interestRate/100 );
    }
}

public class ChequeAccount extends Account {
    private static final int depositTax = 1; // Percent
    public void deposit(int amount)
        { super.deposit( amount*(100-depositTax/100) ); }
    public void processCheque(int amount)
        { withdraw(amount); }
}
```
An example of multiple inheritance issues

Now, suppose we want to add a new kind of account “Check and Savings” that both pays interest and allows cheques to be written.

- Clearly the interface will just be the combination of the cheque and savings classes.

- But what method implementations should we get if we were allowed to extend multiple classes like in the following?

```java
// Java does NOT let you extend multiple classes!
public class ChequeSavingsAccount
    extends ChequeAccount,SavingsAccount
{
}
```

- The cheque and savings classes have different deposit methods, so it's not clear which one the new class should inherit.

- Worse, if the new class uses the cheque account deposit method, due to late binding the addInterest method will call this method and will deduct the depositTax when interest is added!

- This kind of interaction is subtle, and can lead to subtle bugs.

- Thus the designers of Java chose not to allow multiple inheritance.

- This forces programmers to inherit from only one of the classes, and reimplement the rest, but this has the benefit of forcing programmers to consider the interactions in detail.
Multiple extension for interfaces

- Not having multiple inheritance leads to a major limitation: we couldn't pass a combined account to part of the program that expected only a cheque account or a savings account.

- What we'd like is to have the interface of the new class extend the interfaces the savings and cheque classes, but implement the new class separately.

- This interfaces allow us to do exactly this – a new interface can extend multiple existing interfaces.

```java
public interface AccountInter {
    int getBalance();
    void deposit (int amount);
    void withdraw (int amount);
}
public interface SavingsInter extends AccountInter {
    void addInterest();
}
public interface ChequeInter extends AccountInter {
    void deposit(int amount);
    void processCheque(int amount);
}
public interface ChequeSavingsInter extends ChequeInter,SavingsInter {
}
```
Drawbacks of the interface approach

- The main drawback of Java's approach is that in some cases it makes code reuse more difficult.

- Method bodies may end up being copied from one class to another.

- This only happens rarely, and in practice there are usually alternatives that allow code to be reused.

- The main alternative is to create a new class that has contains from other classes, and delegates method call to each. (We'll see this in the labs in detail).