**Idées cours interfaces**

<http://www.cs.sjsu.edu/~pearce/modules/lectures/oop/basics/interfaces.htm>

**Interfaces and Abstract Classes**

A method consists of two parts: operation plus implementation.

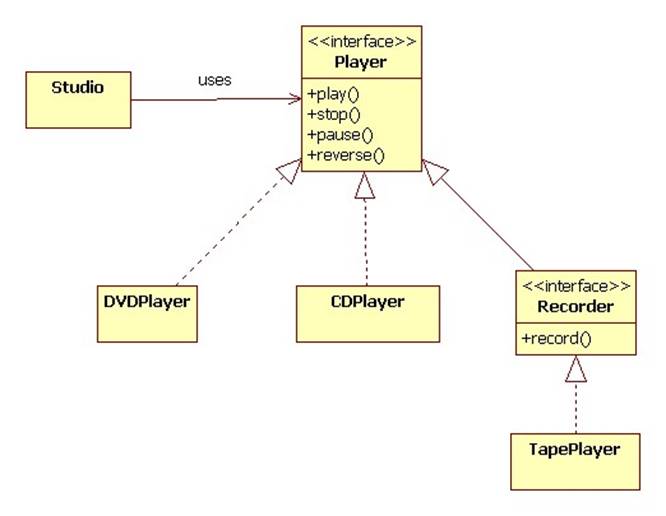
An interface is a collection of operations.

Interfaces can extend other interfaces. In this case the extending interface inherits all of the operations of the extended interface.

A class C implements an interface I if C implements all of the operations in I. We can therefore thing of an interface as the specification of one or more classes.

**UML Notation**

Here's the UML notation for two interfaces and three implementing classes:



In this example the TapePlayer class must implement the operations in the Recorder interface. This includes the declared operation, record, as well as the four inherited operations.

**Java Notation**

Java provides special notation for declaring and implementing interfaces:

interface Player {  
 void play();  
 void stop();  
 void pause();  
 void reverse();  
}

interface Recorder extends Player {  
 void record();  
}

class TapePlayer implements Recorder {  
 public void play() { ... }  
 public void stop() { ... }  
 public void pause() { ... }  
 public void reverse() { ... }  
 public void record() { ... }  
}

**C++ Notation**

In C++ an interface is a class containing public pure virtual functions:

class Player {  
public:  
 virtual void play() = 0;  
 virtual void stop() = 0;  
 virtual void pause() = 0;  
 virtual void reverse() = 0;  
};

class Recorder: public Player {  
public:  
 virtual void record() = 0;  
};

class TapePlayer: public Recorder {  
public:  
 void play();  
 void stop();  
 void pause();  
 void reverse();  
 void record();  
};

If TapePlayer is declared in TapePlayer.h, then the implementations of its operations probably appear in TapePlayer.cpp:

void TapePlayer::play() { ... }  
void TapePlayer::stop() { ... }  
void TapePlayer::pause() { ... }  
void TapePlayer::reverse() { ... }  
void TapePlayer::record() { ... }

**Polymorphism**

Because I is an interface and not a class, we can't say that C is a subclass of I, but we can think of C and I as types and stay that C is a subtype of I.

Formally:

Let A and B be types (classes, interfaces, primitive types, array types, etc.) Then the following statements are true

A <: A  
A <: B and B <: C => A <: C  
A extends or implments B => A <: B

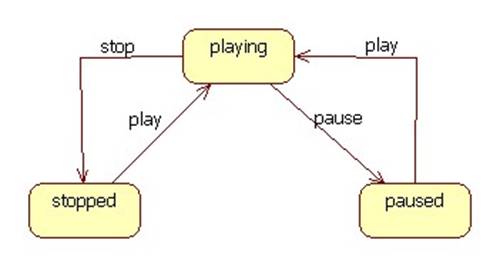
In Java we can add:

char <:short <: int <: long <: float <: double  
A <: Object

**Abstract Classes**

An interface does not contain variables or methods. Any variables and methods must be declared by implementing classes. This can lead to code replication when a variable or method is common to all implementing classes.

For example, suppose all players have a state that some of the methods must set (in addition to other possible tasks):



In each of the three implementing classes we would need to introduce a state variable and code for setting the variable.

Alternatively, we can declare Player to be an abstract class:

abstract class Player {  
 static enum State {STOPPED, PLAYING, PAUSED}  
 private State state = STOPPED;  
 public void play() { state = PLAYING; }  
 public void pause() { state = PAUSED; }  
 public void stop() { state = STOPPED; }  
 public abstract reverse();  
}

Here's the C++ declaration:

class Player {  
private:  
 enum State {STOPPED, PLAYING, PAUSED};  
 State state;  
public:  
 virtual void play() = 0;  
 virtual void stop() = 0;  
 virtual void pause() = 0;  
 virtual void reverse() = 0;  
};

Despite the fact that play, stop, and pause are declared to be pure virtual functions, we can still provide implementations in Player.cpp:

void Player::play() { state = PLAYING; }  
void Player::stop() { state = STOPPED; }  
void Player::pause() { state = PAUSED; }

In UML abstract classes and methods are italicized:



Implementing classes can redefine the inherited methods and call them:

class DVDPlayer extends Player {  
 public void play() {  
 super.play(); // change state  
 // DVD-specific play code goes here  
 }  
 // etc.  
}

Abstract classes can't be instantiated:

Player p = new Player(); // error

A class is abstract if it contains an abstract method. This can happen if a class declares an abstract method or inherits one and doesn't implement it.

Suppose the implementor of DVDPlayer forgets to implement the abstract reverse method. Then DVDPlayer is an abstract class. In Java the compile complains that DVDPlayer should be declared abstract. In C++ the compiler complains when it sees code attempting to instantiate DVDPlayer.