Interfaces in Java:

In the last document we looked at interfaces in the context of a pre-defined one: the Comparable interface, which contracts you to provide the compareTo() method (should you decide to *implement* the interface, of course).

Today, we'll start by revisiting the notion of what an interface is. The following excerpt is taken from the wiki on java interfaces.

Interfaces Vs Inheritance

The body of the interface contains abstract methods, but since all methods in an interface are, by definition, abstract, the `abstract` keyword is not required. Since the interface specifies a set of exposed behaviors, all methods are implicitly `public`.

Thus, a simple interface may be

```
public interface Predator {
       boolean chasePrey(Prey p);
       void eatPrey(Prey p);
}
```

The syntax for implementing an interface uses this formula:

```
... implements InterfaceName[, another interface, another, ...] ...
```

Classes may implement an interface. For example,

```
public class Lion implements Predator {

        public boolean chasePrey(Prey p) {
               // programming to chase prey p (specifically for a lion)
        }

        public void eatPrey (Prey p) {
               // programming to eat prey p (specifically for a lion)
        }
}
```

If a class implements an interface and does not implement all its methods, it must be marked as `abstract`. If a class is abstract, one of its subclasses is expected to implement its unimplemented methods. Although if any of the abstract class' subclasses does not implement all interface methods, the subclass itself must be marked again as `abstract`.

Classes can implement multiple interfaces

```
 public class Frog implements Predator, Prey { ... }
```

An important point arises from the explanation above. You might think initially from the example given that you could write the Predator class as an ordinary superclass and then use inheritance for Lion etc. (you might even argue that it should be an abstract superclass).

And, yes, that works for Lion (A Lion “is a” Predator). However, the inheritance design breaks down when we consider the Frog class : A Frog “is a” Predator AND “is” Prey. **The problem is, we do not have the capability to inherit from multiple superclasses (Java allows single inheritance only), so for this type of example interfaces are the way to go.**

A Practical example:

Ok, in the practical we created a PartTimeAble interface with a doJob() method in it.

We then had a Student implement the interface so that when he does a job – via the doJob() method – he simply gets extra spending money.

I suppose a valid question is: Why have the interface at all? Why not just the doJob() method without this extra "stuff".

An answer could be: the interface provides a "commonality" between all classes which implement it EVEN THOUGH the classes will probably be entirely different in all other ways. With our old friend, polymorphism, we can take advantage of this.

I'll expand the practical example to illustrate. First though I'm going to change the interface slightly – the return type is now boolean, which allows the code to report back whether the job was actually done or not.

```
public interface PartTimeAble {
	public boolean doJob(Job j);
}
```

And the (partial) Student class looks like this (note that it always does the job)

```
public class Student extends Person implements PartTimeAble {

	private String course;
	private double spendingMoney;

	public Student(String name, int age, String course) {
		super(name, age);
		this.course = course;
		this.spendingMoney = 0.0;
	}

	@Override
	public boolean doJob(Job j) {
		spendingMoney += j.getPrice();
		return true;//Job done

	}
```

Here's a sample tester:

```
public static void main(String[] args) {
	Student s1 = new Student("Jim Jones", 19, "Computer Applications");
	s1.doJob(new Job("Bar Work",5.90, 4));
	s1.doJob(new Job("Temping",6.50, 2));

	System.out.println("Jim can now afford " + s1.howManyNoodles(1.00) + "
			packets of noodles");
```

Ok, now I'm also have another class that's capable of doing part time jobs – an Automaton (self-operating machine).

His big thing is that he runs on a battery and may not be able to do the job if he doesn't have enough power.

```
public class Automaton implements PartTimeAble{

	private double batteryLevel;
	private final int BATTERY_UNITS_PER_HOUR = 3;

	private final int INITIAL_CHARGE = 24;

	public Automaton()
	{
		batteryLevel = INITIAL_CHARGE;
	}

	public void recharge()
	{
		batteryLevel = INITIAL_CHARGE;
	}

	public double getBatteryLevel()
	{
		return batteryLevel;
	}

	@Override
	public boolean doJob(Job j) {
		if ((batteryLevel - (j.getTime()*BATTERY_UNITS_PER_HOUR)) < 0)
		{
			return false;//Can't do job
		}
		else
		{
			batteryLevel -= (j.getTime()*BATTERY_UNITS_PER_HOUR);
			return true;
		}


	}
```

Now, I'm going to have a Manager class to manage all these workers. Remember that we have two (could easily have more) totally disparate types of objects but which can both do part-time-work… however, doing the work affects both of them differently i.e. ***the implementation of the interface***.

Before I present the Manager class (called PartTimeManager), I'll show you it in action, so that you hopefully appreciate how polymorphism via the interface is making our code very simple.

```
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		PartTimeManager manager = new PartTimeManager();

		manager.addWorker(new Student("Jim Jones", 19, "CA"));
		manager.addWorker(new Automaton());
		manager.addWorker(new Student("Mike Smith", 24, "CA"));

		//Display details before they do some work
		manager.displayAllWorkers();

		System.out.println("Get all workers to do their default job");
		manager.activateAllWorkers();

		//Display details after they do some work
		manager.displayAllWorkers();
	}
```

Okay, well there must be some arraylists in PartTimeManager (this looks similar to our BookStore class). What will the type (or types) of the ArrayList(s) be?

Let's see…

```
public class PartTimeManager {

//Note: this list can hold any objects that implement
//the PartTimeAble interface
ArrayList<PartTimeAble> partTimeEmployees;

private final Job defaultJob = new Job("Manual Clean", 5.60, 2);

public PartTimeManager()
{
	partTimeEmployees = new ArrayList<PartTimeAble>();
}

public void addWorker(PartTimeAble newWorker)
{
	partTimeEmployees.add(newWorker);

}

//Get all the workers to do a specific job
public void activateAllWorkers(Job j)
{
	for(PartTimeAble currentWorker: partTimeEmployees)
	{
		if(!currentWorker.doJob(j))
		{
	System.out.println("Problem with Worker: " + currentWorker.toString());
		}

	}
}
```

```
//Get all the workers to do the default job
public void activateAllWorkers()
{
	for(PartTimeAble currentWorker: partTimeEmployees)
	{
		if(!currentWorker.doJob(defaultJob))
		{
	System.out.println("Problem with Worker: " + currentWorker.toString());
		}

	}
}


public void displayAllWorkers()
{
	System.out.println("Worker Details");
	System.out.println("----------------------");
	for(PartTimeAble currentWorker: partTimeEmployees)
	{
			System.out.println(currentWorker.toString());
	}

}
```

Below is the tester from earlier AND the output it provides. Look at how the students and automatons updated themselves:

```
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		PartTimeManager manager = new PartTimeManager();

		manager.addWorker(new Student("Jim Jones", 19, "CA"));
		manager.addWorker(new Automaton());
		manager.addWorker(new Student("Mike Smith", 24, "CA"));

		//Display details before they do some work
		manager.displayAllWorkers();

		System.out.println("Get all workers to do their default job");
		manager.activateAllWorkers();

		//Display details after they do some work
		manager.displayAllWorkers();
	}
```

```
Worker Details
----------------------
Student [course=CA, spendingMoney=0.0 Person [name=Jim Jones, age=19]]
Automaton [batteryLevel=24.0]
Student [course=CA, spendingMoney=0.0 Person [name=Mike Smith, age=24]]

Get all workers to do their default job

Worker Details
----------------------
Student [course=CA, spendingMoney=11.2 Person [name=Jim Jones, age=19]]
Automaton [batteryLevel=18.0]
Student [course=CA, spendingMoney=11.2 Person [name=Mike Smith, age=24]]
```