Interfaces and Enumerated types in Java

Interface

- An abstraction that allows programmers to specify a contract that must be met by classes
- The contract just specify the functionalities that should be supported
- It does not state how they should be implemented
- Could use classes and inheritance but Java does not support multiple inheritance

Interface

- Concept similar to a *pure* abstract class
- It is a reference type similar to a class
- Can only specify
  - Constants (public final static fields)
  - Method signatures (public abstract methods)
  - Default methods (after java 8)
  - Static methods (after Java 1.8)
  - Method bodies **only exist for default** and **static** methods

- Interfaces are declared using the ***interface*** keyword

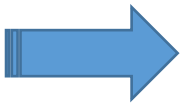
Defining an Interface

- Syntax
  - Specified using the interface keyword
    - The fields are always public static final
    - Methods are always public
    - Non-default/non-static methods are always abstract
    - Default modifiers can be omitted (all or a subset)
- Example:

```
public interface InterfaceName {
    // constant declarations

    // base of natural logarithms
    public static final double E = 2.718282;

    // method signatures
    public abstract void doSomething (int i, double x);
    public int doSomethingElse(String s);
}
```



```
public interface InterfaceName {
    // constant declarations

    // base of natural logarithms
    double E = 2.718282;

    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

Implementing an Interface

- You extend a class, but you implement an interface
- Implementing an interface means that the class promises to implement all the (*abstract*) methods declared in the interface
  - If the class fails to do this, then
    - Compilation error, unless
    - Class is abstract
- To implement an interface use the *implements* clause in the class declaration
  - Ex: *public class Example implements Comparable*
- It is possible to implement more than one interface
  - *public class ExampleTwo implements Comparable, AnotherInterface*

Motivation - Example

- How to implement a generic sort algorithm?
- Need to have a generic functionality for comparing objects

```
public abstract class Comparable {
  public abstract int compareTo(Comparable obj);
}
```

```
public class Sort {
  nublic static void sort(Comparable[] vec);  {
    Comparable obj1, obj2;
    …
    int res = obj1.compareTo(obj2);
    if (res == 0) …  // equal
    else if (res > 0) … // greater
    else …
    …
}
```

- Problem?
- Vector to sort must be of a subtype of Comparable
- May not be possible using single-inheritance
- Solution: Use interface

Person Implementing Comparable

- Specify a interface for representing the functionality of comparing two objects

```
public interface Comparable {
     int compareTo(Comparable obj);
}
```

```
public class Student extends Person implements Comparable {
   // original code of Person
   private int _studentNumber
   // …

   public Student(int number, String name) {
      super(name);
     _studentNumber = number;
    }
   // …
   // implementing the interface
   public int compareTo (Comparable comp) {
     Student st = (Student)comp;
     return _studentNumber – st._studentNumber;
    }
}
```

- Not the ideal solution due to the downcast

Questions I

- How many errors?
  - None.
  - All are equivalent and consistent with default modifier:
    - public static final

```
public  interface Try {
  int a = 10;
  public int b = 10;
  public static final int c = 10;
  final int d = 10;
  static int e = 0;
}
```

- How many errors?
  - One

```
public interface Try {
  int a;
}
```

  - Interface variables must be initialized at the time of declaration

- Identify the error(s) in the following

```
public interface Try {
  int x = 4;
}
```

  - Interface variable are always **final**!

```
public  class Sample implements Try {
  public static void main(String args[]) {
    System.out.println("Initial value of x: " + x);
    x = 20;
  }
}
```

Interface as Types

- When a class is defined, the compiler views the class as a new type
- The same thing is true of interfaces. The compiler regards an interface as a type
  - It can be used to declare variables or method parameters
- How methods are executed on a interface type?
- Methods are executed as polymorphic methods
- Problem: Define a generic method that given two references returns the greatest one
- Type of o1 and o2 implements *Comparable*
  - But, both objects **must** be of the same class
- How to avoid downcast?

```
public Object findLargest(Object o1, Object o2) {
  Comparable obj1 = (Comparable) o1;
  Comparable obj2 = (Comparable) o2;
  if (obj1.compareTo(obj2) > 0)
    return obj1;
  else
    return obj2;
}
```

Interfaces and Inheritance

- An interface can inherit from another interface
- This means that the derived interface inherits all fields and methods of the super interface

```
public interface MySuperInterface {
   public void sayHello();
}
```

```
public interface MySubInterface extends MySuperInterface {
   public void sayGoodbye();
}
```

- Multiple inheritance is valid with interfaces

```
public interface MySubInterface extends SuperInter1, SuperInter2 {
   // …
}
```

- Class implementing a derived interface
  - Must provide an implementation for methods:
    - Declared in the derived interface
    - And in all superinterfaces

```
public class Ex implements MySubInterface{
   public void sayHello() {
      System.out.println("Hi!");
   }
    public void sayGoodbye() {
      System.out.println("Bye!");
   }
}
```

Questions II

- What happens when a class implements two (or more) interface that declare the same method?
  - Implementation of the method once is enough.

```
interface A {
  public void aaa();
}
```

```
interface B {
  public void aaa();
}
```

```
public class  MyClass implements A, B {
  public void aaa() { … }
}
```

- How many errors?
  - One
  - A class cannot implement two interfaces that have methods with same name but different return type

```
interface A {
  int aa();
}
```

```
interface B {
  void aa();
}
```

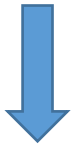
```
public  class Sample implements A, B {
    public void aa() {
    }

    public int aa() {
      return -1;
    }
}
```

Evolving Interfaces

- Suppose you have the following interface

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```



- And, then at a later time, you decide to add a new method, *doIt*

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
   void doIt();
}
```

- What is the impact of doing this?

- All classes that implemented *DoIt* interface will break

Default Methods in Interfaces

- Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces
- Default methods have a body
- If a class does not implement a default method of an interface then the implementation specified in the interface is used
- Previous example

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
   default void doIt() {
     // default implementation
   }
}
```

- If a class implementing **DoIt** does not provide an implementation for **doIt**, then the implementation provided in the interface is used

Extending Interfaces with default Methods

- When you extend an interface that contains a default method, you can do the following:
  1. Not mention the default method at all, which lets your extended interface inherit the default method
  2. Re-declare the default method, which makes it abstract
  3. Redefine the default method, which overrides it

Questions III

- What is wrong with the following interface?
  - The default keyword is missing

```
public interface SomethingIsWrong {
    void aMethod(int aValue){
       System.out.println("Hi Mom");
    }
}
```

- Is the following interface valid?
  - Yes

```
public interface Marker {
}
```

- Identify the error(s) in the following

```
public interface Something {
   default  void aMethod(int aValue){
     System.out.println("Hi Mom");
   }
}
```

```
public class Example implements Something {
   default  void aMethod(int aValue){
       System.out.println("Hi Mom!!!!!!!!!!");
        return 1;
   }
}
```

What are interfaces for?

- Reason 1
  - A class can only extend one other class, but it can implement multiple interfaces
  - This lets the class fill multiple "roles"
- Reason 2
  - You can write methods that work for more than one kind of class

```
public Object findLargest(Comparable o1, Comparable o2) {
  if (obj1.compareTo(obj2) > 0)
    return o1;
  else
    return o2;
}
```

Enumerated types in Java

Anti-pattern: int constants

- How Represent the type day of week?
- What's wrong with using `int` constants to represent each day of week?
  - variation (also bad): using `String`s for the same purpose.
  - No automatic checking!

```
public class Birthday {
  public static final int MONDAY = 0;
  public static final int TUESDAY = 1;
…
  private int _dayOfWeek;

  public Birthday(int d) {
    _dayOfWeek = d;
  }

  public String toString() {
    String str == “”;

    switch( dayOfWeek) {
      case MONDAY;
        str = “Monday”;
        break;
       …
    }
  }
  …
}
```

Enumerated types

- **enum**: A type of objects with a fixed set of constant values

```
public enum Name {
    VALUE1, VALUE2, ..., VALUEn;
}
```

- Usually placed into its own .java file
- Each ***VALUEi*** is an instance of the enumerated type

```
public enum DayOfWeek {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
  SUNDAY;
}
```

- **Effective Java Tip #30:** Use `enum`s instead of `int` constants.

  "The advantages of `enum` types over `int` constants are compelling. Enums are far more readable, safer, and more powerful."

What is an Enum?

- Java Enums are classes that export *one* instance for each enumeration constant via a public static final field
- Enum types are effectively final
  - no accessible constructor
  - clients can neither create instances or extend it
  - so, enum types are ***instance controlled***
- The `enum` WeekOfDayis roughly equal to the following short class:

```
public final class WeekOfDay extends Enum<WeekOfDay> {
    public static final WeekOfDay MONDAY    = new WeekOfDay();
    public static final WeekOfDay TUESDAY   = new WeekOfDay();
    public static final WeekOfDay WEDNESDAY = new WeekOfDay();
    public static final WeekOfDay THURSDAY  = new WeekOfDay();
    …
    private WeekOfDay() {}
}
```

- Enum<T> is a generic type already defined in Java

Enum functionalities

- Enums provide compile-time type safety
  - use it as the type of a variable, field, parameter, or return
- Enums with identically named constants can co-exist
  - as each type has its own namespace
- Compare them with `==` (why don't we need to use `equals`?)
- Any enum type has
  - a static *values()* method that returns an array of its values in the order it was declared
  - a *toString()* that returns the declared name of each enum value
    - can be overridden

Enum methods

method	description
`int compareTo(E)`	all `enum` types are Comparable by order of declaration
`boolean equals(o)`	not needed; can just use ==
`String name()`	equivalent to `toString`
`int ordinal()`	returns an enum's 0-based number by order of declaration (first is 0, then 1, then 2, ...)

method	description
`static E valueOf(s)`	converts a string into an `enum` value
`static E[] values()`	an array of all values of your enumeration

More complex enums

- Can augment Enums types with methods, constructors and attributes
  - just like any Java class
  - Constructor must be private

```
public enum DayOfWeek {
   MONDAY("Segunda-feira"), TUESDAY("Terça-feira"),
   WEDNESDAY("Quarta-feira"), …;
   private String _day;

   private DayOfWeek(String day) {
      _day = day;
   }

   public String toString() {
      return _day;
   }
}
```