Java Interfaces

Concept

- **`interface`** is a way to describe what classes should do, without specifying how they should do it.
- It's not a class but a set of requirements for classes that want to conform to the interface

```
public interface Comparable
{
        int compareTo(Object
  otherObject);
}
```

this requires that any class implementing the `Comparable` interface contains a `compareTo` method, and this method must take an `Object` parameter and return an integer

Interface declarations

- The declaration consists of a keyword `interface`, its name, and the members
- Similar to classes, interfaces can have three types of members
  - constants (fields)
  - methods
  - nested classes and interfaces

Interface member – constants

- An interface can define named constants, which are `public`, `static` and `final` (these modifiers are omitted by convention) automatically. Interfaces never contain instance fields.
- All the named constants MUST be initialized

An example interface

```
interface Verbose {
        int SILENT = 0;
        int TERSE = 1;
        int NORMAL = 2;
        int VERBOSE = 3;

        void setVerbosity (int level);
        int getVerbosity();
}
```

Interface member – methods

- They are implicitly `abstract` (omitted by convention). So every method declaration consists of the method header and a semicolon.
- They are implicitly `public` (omitted by convention). No other types of access modifiers are allowed.
- They can't be `final`, nor `static`

Modifiers of interfaces itself

- An interface can have different modifiers as follows
  - `public/package(default)`
  - `abstract`
    - all interfaces are implicitly `abstract`
    - omitted by convention

To implement interfaces in a class

- Two steps to make a class implement an interface
  1. declare that the class intends to implement the given interface by using the `implements` keyword

```
class Employee implements Comparable { . . . }
```

  2. supply definitions for all methods in the interface

```
public int compareTo(Object otherObject) {
     Employee other = (Employee) otherObject;
     if (salary < other.salary) return -1;
     if (salary > other.salary) return 1;
     return 0; }
```

note: in the `Comparable` interface declaration, the method `compareTo()` is `public` implicitly but this modifier is omitted. But in the `Employee` class design, you cannot omit the `public` modifier, otherwise, it will be assumed to have `package` accessibility

- If a class leaves any method of the interface undefined, the class becomes `abstract` class and must be declared `abstract`
- A single class can implement multiple interfaces. Just separate the interface names by comma

```
class Employee implements Comparable, Cloneable {. . .}
```

Instantiation properties of interfaces

- Interfaces are not classes. You can never use the `new` operator to instantiate an interface.

```
public interface Comparable {
        . . . }
Comparable x = new Comparable( );
```

- You can still declare interface variables

```
Comparable x;
```

  but they must refer to an object of a class that implements the interface

```
class Employee implements Comparable {
                . . .
}
x = new Employee( );
```

Extending interfaces

- Interfaces support multiple inheritance – an interface can extend more than one interface
- Superinterfaces and subinterfaces

Example

```
public interface SerializableRunnable extends
java.io.Serializable, Runnable {
        . . .
}
```

Extending interfaces – about constants (1)

- An extended interface inherits all the constants from its superinterfaces
- Take care when the subinterface inherits more than one constants with the same name, or the subinterface and superinterface contain constants with the same name — always use sufficient enough information to refer to the target constants

- When an interface inherits two or more constants with the same name
  - In the subinterface, explicitly use the superinterface name to refer to the constant of that superinterface

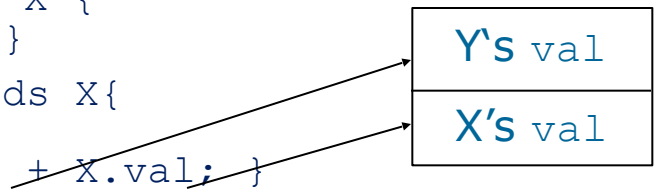
E.g.

```
interface A {
            int val = 1;
        }
interface B {
            int val = 2;
}

interface C extends A, B {
    System.out.println("A.val = "+ A.val);
    System.out.println("B.val = "+ B.val);
}
```

- If a superinterface and a subinterface contain two constants with the same name, then the one belonging to the superinterface is **hidden**
  1. in the subinterface
     - access the subinterface-version constants by directly using its name
     - access the superinterface-version constants by using the superinterface name followed by a dot and then the constant name

E.g

```
interface X {
    int val = 1; }

interface Y extends X{
    int val = 2;
    int sum = val + X.val; }
```



  2. outside the subinterface and the superinterface
     - you can access both of the constants by explicitly giving the interface name.

       E.g. in previous example, use `Y.val` and `Y.sum` to access constants `val` and `sum` of interface `Y`, and use `X.val` to access constant `val` of interface `X`.

- When a superinterface and a subinterface contain two constants with the same name, and a class implements the subinterface
  - the class inherits the subinterface-version constants as its static fields. Their access follow the rule of class's static fields access.

```
E.g    class Z implements Y { }

                //inside the class
                System.out.println("Z.val:"+val);   //Z.val
   = 2
                //outside the class
                System.out.println("Z.val:"+Z.val); //Z.val
   = 2
```

  - object reference can be used to access the constants
    - subinterface-version constants are accessed by using the object reference followed by a dot followed by the constant name
    - superinterface-version constants are accessed by explicit casting

```
        E.g.    Z v = new Z( );
                System.out.print( "v.val = " + v.val
                                  +", ((Y)v).val = " +
((Y)v).val
                                  +", ((X)v).val = " +
    ((X)v).val );
```

output: v.val = 2, ((Y)v).val = 2, ((X)v).val = 1

Extending interfaces – about methods

- If a declared method in a subinterface has the same signature as an inherited method and the same return type, then the new declaration *overrides* the inherited method in its superinterface. If the only difference is in the return type, then there will be a compile-time error
- An interface can inherit more than one methods with the same signature and return type. A class can implement different interfaces containing methods with the same signature and return type.
- Overriding in interfaces has **NO** question of ambiguity. The real behavior is ultimately decided by the implementation in the class implementing them. The real issue is whether a single implementation can honor all the contracts implied by that method in different interfaces
- Methods with same name but different parameter lists are `overloaded`

Interface References

Referencing an Interface Variable

intfRef.varName

intfRef.mthName(args)

```
interface A {
  void display(String s);
}

class C1 implements A {
  public void display(String s) {
    System.out.println("C1: " + s);
  }
}

class C2 implements A {
  public void display(String s) {
    System.out.println("C2: " + s);
  }
}

class C3 implements A {
  public void display(String s) {
    System.out.println("C3: " + s);
  }
}
```

```
class InterfaceReferenceVariable
{
  public static void main(String
args[]) {
    A a;
    a = new C1();
    a.display("String 1");
    a = new C2();
    a.display("String 2");
    a = new C3();
    a.display("String 3");
  }
}
```

Result :

C1: String 1

C2: String 2

C3: String 3

Marker interfaces

- A marker (tagging) interface has neither methods nor constants, its only purpose is to allow the use of `instanceof` in a type inquiry.

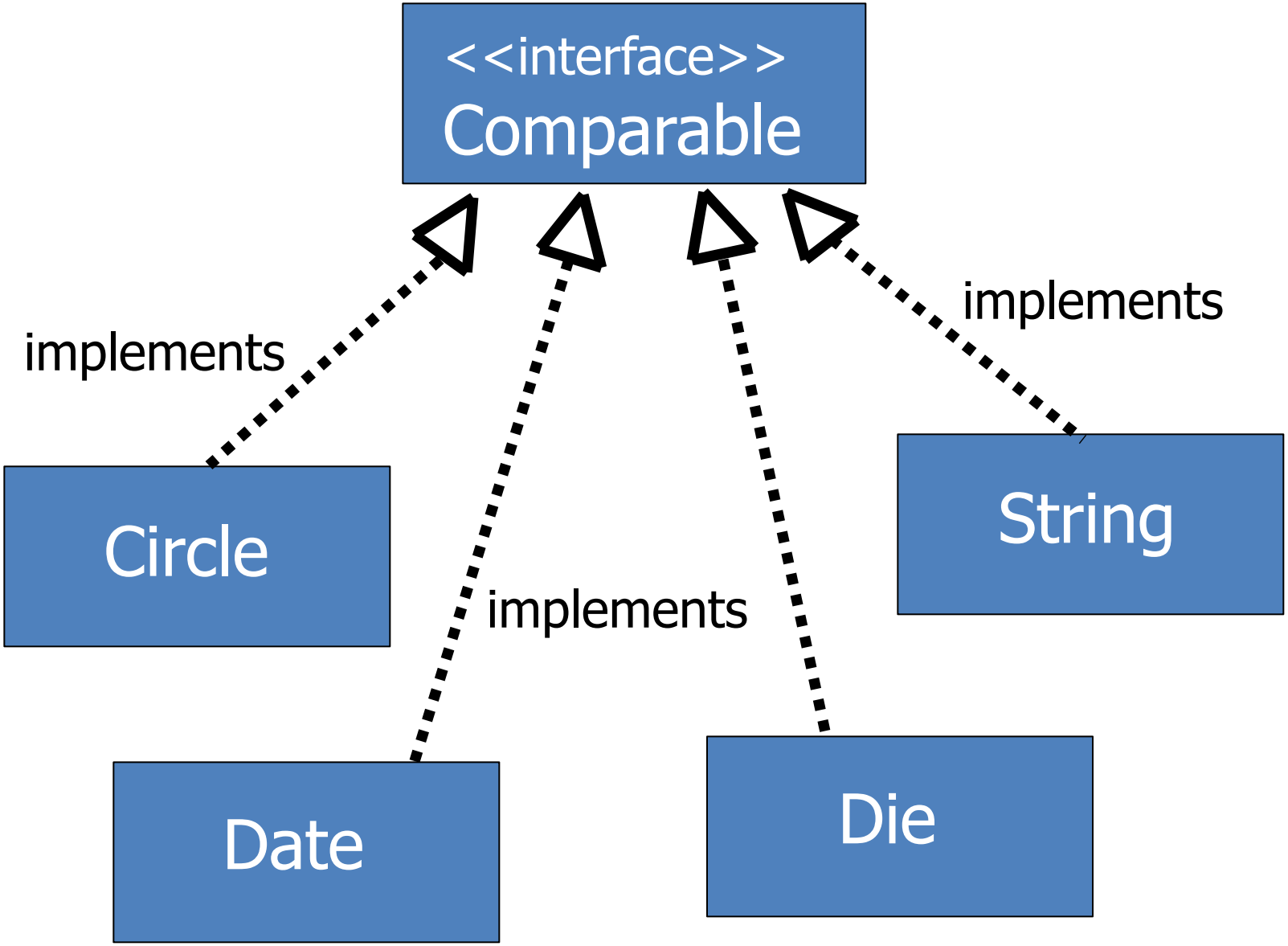
Interfaces and abstract classes

- Why bother introducing two concepts: abstract class and interface?

```
abstract class Comparable  {
    public abstract int compareTo (Object otherObject);
}
class Employee extends Comparable  {
    pulibc int compareTo(Object otherObject) { . . . }
}
```

```
public interface Comparable {
    int compareTo (Object otherObject);
}
class Employee implements Comparable  {
    public int compareTo (Object otherObject) { . . . }
}
```

- A class can only extend a single abstract class, but it can implement as many interfaces as it wants
- An abstract class can have a partial implementation, protected parts, static methods and so on, while interfaces are limited to public constants and public methods with no implementation


<<interface>>
Comparable
implements
implements
implements
implements
Circle
Date
Die
String

Comparable interface

```
public interface Comparable {
   public int compareTo( Object o );
}
```

for x, y objects of the same class

x.compareTo(y) < 0 means “x < y”

x.compareTo(y) > 0 means “x > y”

otherwise, “x is neither < nor > y”

- recommended that compareTo() is consistent with equals()
- if o cannot be cast to the same class as x, then generates a ClassCast Exception

```
public class Student implements Comparable
{  private String name;
   public int compareTo( Object o )
   {
        Student other = (Student) o;
        return ((this.name).compareTo(other.name));
   }
}
```

use < & > to compare primitives

invoke compareTo() method to compare objects

remember to have "implements Comparable" on the class header

Mixing inheritance, interfaces

- It is legal for a class to extend a parent and to implement any number of interfaces

```
public class BankAccount {
// ...
}

public class NumberedAccount
    extends BankAccount implements Comparable {
  private int myNumber;

  public int compareTo(Object o) {
    return myNumber - ((BankAccount)o).myNumber;
  }
}
```

A problem with interfaces

```
public interface Shape2D {
   int getX();
   int getY();
   double getArea();
   double getPerimeter();
}
```

- Every shape will implement `getX` and `getY` the same, but each shape probably implements `getArea` and `getPerimeter` differently

A bad solution

```
public class Shape implements Shape2D {
   private int myX, myY;

   public Shape(int x, int y) {
       myX = x;  myY = y;
   }
   public int getX() { return myX; }
   public int getY() { return myY; }

   // subclasses should override these, please
   public double getArea() { return 0; }
   public double getPerimeter() { return 0; }
}
```

- BAD: the `Shape` class can be instantiated
- BAD: a subclass might forget to override the methods

Abstract classes

- **abstract class**: a hybrid between an interface and a class
  - used to define a generic parent type that can contain method declarations (like an interface) and/or method bodies (like a class)
  - like interfaces, abstract classes that cannot be instantiated (cannot use `new` to create any objects of their type)

What goes in an abstract class?

- implement common state and behavior that will be inherited by subclasses (parent class role)
- declare generic behaviors that subclasses must implement (interface role)

Abstract class syntax

- put `abstract` keyword on class header and on any generic (abstract) methods
  - A class can be declared `abstract` even though it has no abstract methods
  - Any class with abstract methods *must* be declared `abstract`, or it will not compile
- Variables of abstract types may be declared, but *objects* of abstract types cannot be constructed

Abstract class example

```
public abstract class Shape implements Shape2D {
   private int myX, myY;

   public Shape(int x, int y) {
       myX = x;  myY = y;
   }
   public int getX() { return myX; }
   public int getY() { return myY; }

   public abstract double getArea();
   public abstract double getPerimeter();
}
```

- `Shape` class cannot be instantiated
- all classes that extend `Shape` **must** implement `getArea` and `getPerimeter` or else must also be declared `abstract`

Extending an abstract class

```
public class Rectangle extends Shape {
  private int myWidth, myHeight;

  public Rectangle(int x, int y, int w, int h) {
    super(x, y);
    myWidth = w;   myHeight = h;
  }

  public double getArea() {
    return myWidth * myHeight;
  }
  public double getPerimeter() {
    return 2*myWidth + 2*myHeight;
  }
}

// ... example usage ...
Shape rect = new Rectangle(1, 2, 10, 5);
```

Interface / abstract class chart

TABLE 3.1
Comparison of Actual Classes, Abstract Classes, and Interfaces

Property	Actual Class	Abstract Class	Interface
Instances (objects) of this can be created	Yes	No	No
This can define instance variables and methods	Yes	Yes	No
This can define constants	Yes	Yes	Yes
The number of these a class can extend	0 or 1	0 or 1	0
The number of these a class can implement	0	0	Any number
This can extend another class	Yes	Yes	No
This can declare abstract methods	No	Yes	Yes
Variables of this type can be declared	Yes	Yes	Yes