Interfaces

Syntax, terminology, examples, polymorphism, casting rationale, new enhancements

Interface basics

- An **interface** describes a set of behaviors by declaring methods.

```
public interface MyInterface {
    void myMethod(int arg);
}
```

- Classes can **implement** interfaces. When a class implements an interface, it must define all of the methods declared in the interface.

```
public class MyClass implements MyInterface {
   // Other class stuff here

   public void myMethod(int arg) {
     // body of method goes here
   }

   // More class stuff here
}
```

Interfaces are…

Interfaces are:

- **Design tools.** Interfaces spell out what an implementing class *can do*.
  - Abstract out behavior that you need for a particular purpose.
  - Some developers write their interfaces first, then program against those interfaces.
- **Development contracts.** They ensure that a particular object provides a given set of methods.
- **Guarantees.** Every class that implements the interface must have public behaviors that match the declarations in the interface.

Some interfaces from the Java API

- `Comparable<T>` – comparing two objects of type T, where T is a class (used in ordering)
- `Runnable` – execution threads
- `Serializable` – objects saved to files
- `Collection<T>` – collections of objects of class type T
- `ActionListener` – for graphical user interfaces

… and many, many more.

File: ForSale.java

```
public interface ForSale {
   /**
    * Is this an expensive item/service?
    * The implementing class decides what "expensive” means.
    * @return true if the object is expensive, false if not.
    */
   boolean isExpensive();

   /**
    * What is the cost of this item/service?
    * @return the cost of this item/service.
    */
   double getCost();

   /**
    * Sets the cost of this item/service.
    * The client should check the integrity of the parameter.
    * @param cost the new cost of the item/service
    */
   void setCost(double cost);
}
```

Interface syntax and restrictions

- File name follows class naming standards: NameOfInterface.java.
- Interface declaration uses *interface* (rather than *class*).
- **Signatures** of non-static methods with their return type are *declared*.
  - Non-static methods are not *defined* in the interface .
  - Non-static methods are declared without access modifiers.
  - All non-static methods are public.
- Interfaces can have public constants (variables that are public, final, and static).
- Interfaces can be (should be!) documented via Javadoc.

File: AirplaneTicket.java

```
public class AirplaneTicket implements ForSale {
   private String home;
   private String destination;
   private double cost;
   private final static int EXPENSIVE_LIMIT = 799;
   public AirplaneTicket(String destination, String home,
                             double cost) {
      this.destination = destination;
      this.home = home; this.cost = cost;
     }
   public String printMe() {
         return "Flight from " + home + " to " + destination;
   }
   public boolean isExpensive() {
      return cost > EXPENSIVE_LIMIT;
   }
   public double getCost() { return cost; }
   public void setCost(double cost) { this.cost = cost;}
}
```

File: PottedPlant.java

```
public class PottedPlant implements ForSale {
   private String kind;
   private double cost;
   private final static double CHEAP = 3.0;

   public PottedPlant(String kind, double cost) {
      this.kind = kind;
      this.cost = cost;
   }
   public double getCost() { return cost; }
   public void setCost(double cost) { this.cost = cost; }
   public boolean isExpensive() {
      return cost > CHEAP;
   }

   public void goOnSale() { cost = cost/2; }
}
```

Declaring interface types

- You can declare a variable as an interface type.
  ```
  ForSale x; // declaration
  ```

- But you cannot instantiate it with the interface.
  ```
  x = new ForSale();  // NO, NO. Does not work
  ```

- You CAN instantiate it with a “concrete” implementing class.
  ```
  x = new AirplaneTicket(“LHR”, “RDU”, 989);
  ```

- And though it was declared as type `ForSale`, you can call `AirplaneTicket` methods with it.
  ```
  if (x.isExpensive()) …
  ```

- Cast to its “runtime” class type before using methods not declared in the interface.
  ```
  ((AirplaneTicket)x).printMe();
  ```

Polymorphism and interfaces

```
ForSale x;

x = new AirplaneTicket(“LHR”, “RDU”, 456);
if (x.isExpensive()) … // false here…

x = new PottedPlant (“Petunia”, 25);
if (x.isExpensive()) … // true here…
```

- The *declared* type for `x` is `ForSale`. In the code, it gets two different *actual* types at runtime.
- If you call an interface method, the code that is executed is the code corresponding to the actual type. This is an example of **polymorphism**.

Casting

```
public void callingCode();
   ForSale x;
   x = new AirplaneTicket(“LHR”, “RDU”, 456);

   System.out.println(x.getCost()); // ok
   System.out.println(x.printMe()); // NOT ok
```

- Even though the actual type for `x` is AirplaneTicket, the compiler sticks to its declared type to see what methods are legit.
- Can get around the problem by **casting**. This code is legal:

```
System.out.println(((AirplaneTicket)x).printMe());
```

- Cast syntax: `(Actual-Type) object`
- The operator `.` has precedence over the cast, making parentheses necessary: `((AirplaneTicket)x)`

Method parameters

```
public void callingCode();
   ForSale x;
   x = new AirplaneTicket(“LHR”, “RDU”, 456);
   exampleMethod(x); // Legal
   … }

public void exampleMethod(ForSale arg) {
   System.out.println(arg.getCost()); // ok
   System.out.println(arg.printMe()); // NOT ok
```

- You can declare a method parameter type as an interface type.
- When you do, try to stick with the methods declared in the interface rather than extra ones defined in the class that is the type for the actual parameter.
- Cast if needed, but beware….

Casting exceptions

```
public void callingCode(){
   ForSale x;
   // initialize x
   // …
}

public void exampleMethod(ForSale arg) {
   System.out.println(arg.getCost());
   ((PottedPlant)arg).goOnSale();
```

- This code will compile.
- This code will blow up at runtime if the actual parameter is not of type `PottedPlant`.
- Exception = `ClassCastException`

Why are interfaces useful? – from software professionals

- They encourage **smart, uniform design**. An interface can give one standard to which all programmers on a project can write. For example, if your project accesses a database for different types of information, you can create an interface that spells out the database operations. All code that accesses the database will look the same.
- They enhance **maintainability**. Any class can implement an interface. As long as you don't change the interface itself, you can change any of the implementing classes or create new ones without having to change the other classes that depend on the interface.
- They enhance **readability**. Readers of the code know exactly what to expect of a class that implements an interface.
- They enhance **flexibility in class definitions**. A class that implements an interface can implement another interface or extend a class without impacting the interface.
- By coding to interfaces, developers can **quickly publish the methods by which their objects are to be used**. This offers developers great flexibility in their implementations. It also makes it easier to connect together classes or subsystems.

Static and default interface methods

Java 8 introduced two additional interface concepts:

1. **static methods** that are defined in the interface itself. These methods are called without an instance. They are public.
2. **default methods**, that are also defined in the interface. They are also public and they provide a default implementation that the implementing classes can optionally override.

```
public interface ForSale {
   public boolean isExpensive();
   public double getCost();
   public void setCost(double cost);

   static public String priceCurrency() {
      return "USD";
   }
   default public boolean isTaxable() {
      return true;
   }
}
```

Using static and default methods

Redefining a default requires an override.

```
public class CropSeed implements ForSale {
   // Data and methods go here

   @Override
   public boolean isTaxable() {
      return false;
   }
 }
```

Call static methods with the interface name. (You cannot use an implementing class name.)

```
 String currency = ForSale.priceCurrency();
```

default is a Java keyword that can be used in interfaces only. *You cannot use it in a class or anywhere else.*