Interfaces and Sub-typing

So far, we have talked about interfaces informally in the ordinary English sense of the word.

- “interface to a class tells the client how to obtain the functionality implemented in that class”

Java has a construct called interface which can be used formally for this purpose

- and for doing some other really cool things…

Java interface

```java
interface IPuzzle{
    void Scramble();
    int Tile(int r, int c);
    boolean Move(char d);
}
```

```java
class IntPuzzle implements IPuzzle{
    public void Scramble(){
    ...
    }
    public int Tile(int r, int c){
    ...
    }
    public boolean Move(char d){
    ...
    }
}
```

Notes

- Interface itself cannot be instantiated.
  - incomplete specification
- It is not enough for a class to just have implementations of interface methods; class header must also assert “implements I” for Java to recognize that the class implements interface I.
- A class may implement several interfaces.
  - (eg) class X implements IPuzzle, IRAq {
    ...
    }

- Name of interface: IPuzzle
- A class can implement this interface by implementing public instance methods with the names and type signatures specified in the interface.
- The class may implement other methods.
Why interface construct?

- One use of interfaces: software engineering
  - specifying and enforcing boundaries between different parts of a team project, as in Puzzle example.
- But interfaces can do much more.
  - Interfaces let you write more “generic” code that reduces code duplication.

Example of code duplication

- Suppose we have two implementations of puzzles:
  - Class IntPuzzle uses an integer to hold state
  - Class ArrayPuzzle uses an array to hold state
- Assume client wants to use both implementations in code
  - perhaps for benchmarking both implementations to pick the best one?
  - client code has a display method as always to print out puzzles
- What would the display method look like?

```java
Class Client{
    IntPuzzle p1 = new IntPuzzle();
    ArrayPuzzle p2 = new ArrayPuzzle();
    ....Display(p1)...Display(p2)...

    public static void Display(IntPuzzle p){
        for (int r = 0; r < 3; r++)
            for (int c = 0;c<3;c++) {
                System.out.print(p.Tile(r,c));
                System.out.print(' ');
            }
    }

    public static void Display(ArrayPuzzle p){
        for (int r = 0; r < 3; r++)
            for (int c = 0;c<3;c++) {
                System.out.print(p.Tile(r,c));
                System.out.print(' ');
            }
    }
}
```

Observation

- Two Display methods are needed because types IntPuzzle and ArrayPuzzle are different, and parameter p must have one type or the other.
- Ironically, the code inside the two methods is identical.
  - Code relies only on assumption that parameter p is passed an object that has an instance method Tile(int,int).
- Is there a way to avoid this code duplication?
  - Use interfaces and sub-typing
Interfaces as types

- Name of an interface can be used as a variable type.
  - (eg) IPuzzle p1, p2;
- Class that implements the interface is said to be a sub-type of the interface type.
  - IntPuzzle and ArrayPuzzle are sub-types of IPuzzle.
- Interface is said to be a super-type of those classes.
  - Ipuzzle is a super-type of type IntPuzzle and ArrayPuzzle.

Note

- Since a class can implement several interfaces, it may have many super-types.
- An interface can be implemented by several classes, so it may have many sub-types.

Paradox with interfaces as types

- We cannot instantiate an interface I.
  - Interface is a partial specification.
- If we cannot create objects of type I, why bother permitting interface names to be types?
  - (eg) IPuzzle p1,p2;
  - Fine, but what would we ever assign to p1 and p2?!?
- To understand this, let us look at a real-life analogy.

Names, Objects and Types

- In programming languages, like in real life, we attribute type both to names (variables) and to objects.
- Example from real life: gender
  - Two types: Male and Female
  - These types are assigned to people (objects):
    - The President of Cornell is a Male.
    - The Provost of Cornell is a Female.
- These types are also assigned to names:
  - Male George, Osama, Helmut, Bubba;
  - Female Rie, Naomi, Indira, Melanie;
Unisex names

- Some names can refer to people of either gender:
  - (eg): Sandy, Pat, Jackie
- How do we fit unisex names into our classification?
- Here is an idea…..

Sub-typing in real life

- Let us add a new type called Human.
- Humans have certain functionalities:
  - They walk upright.
  - They have juxtaposed thumbs.
  - They are intelligent….
- Male and female are sub-types of type Human because they implement this functionality although in different ways.
- Like an interface, type Human cannot be instantiated directly: every human must be either a male or a female.

Sub-types and subsets

- Sub-types and subsets are distinct concepts.
- Sub-types are characterized by some common functionality.
  - Sub-type female in our example is characterized by ability to give birth.
- In this example, a mixture of males and females is a subset of type human, but it is not a sub-type.

Back to names

- Using these types, we can now give types to unisex names as well:
  - Male George, Osama, Helmut, Bubba;
  - Female Rie, Naomi, Indira, Melanie;
  - Human Jo, Sandy, Pat;
Naming people

• Simple picture without sub-typing:
  – Male objects get male names.
  – Female objects get female names.
• Examples:
  //we created a new male object and named it George
  George = new Male(); //type checks
  //give object named Osama the alias Bubba
  Bubba = Osama; //type checks
  //give object named Bubba the alias Melanie
  Melanie = Bubba; //type mismatch
• In last example, we do not need to know anything about who Bubba is to see that there is a type mismatch.

Up-casting

• Situation is a little more complex with unisex names (sub-typing).
• Example: Sandy = new Female();
  – Type of reference returned by RHS is Female.
  – Type of LHS name is Human.
  – Nevertheless, no type error because Female is sub-type of Human.
• Up-casting: type of RHS reference is sub-type of type of LHS name.
• Up-casting is always type-correct.
• Example: Sandy = Laura;
  – You do not need to know the object named Laura to determine that the assignment is type-correct.

Down-casting

• Is this type-correct?
  Bubba = Sandy;
• Answer: depends.
  – Type of RHS name (reference) Sandy is Human which is super-type of LHS name
  – Type of object named Sandy: either Male or Female
  – Whether or not the assignment is legal depends on the type of the RHS reference but on the type of the actual object.
• Down-casting: Type of LHS name is sub-type of RHS reference.
• Down-casting may or may not be legal
  – need to look at object to determine legality

Resolution of paradox with interfaces as types

• Java allows up-casting:
  – IPuzzle p1 = new ArrayPuzzle();
  – IPuzzle p2 = new IntPuzzle();
• Note:
  – Type of reference returned by right-hand side expression of first statement is ArrayPuzzle.
  – Type of variable on left-hand side is IPuzzle.
  – Two types are different, but type of rhs reference is a sub-type of type of the variable.
Why up-casting?

- Sub-typing and up-casting allow you to avoid code duplication in many situations.
- Puzzle example: you and client agree on interface IPuzzle.

interface IPuzzle{
  void Scramble();
  int Tile(int r, int c);
  boolean Move(char d);
}

Your code

Class IntPuzzle implements IPuzzle{
  ....Scramble()...Tile()...Move()...Twist()
}

Class ArrayPuzzle implements IPuzzle{
  ........Scramble()...Tile()...Move()...
}

Class IntPuzzle implements a method called Twist which is not a method of interface IPuzzle.

Client Code

Class Client{
  IntPuzzle p1 = new IntPuzzle();
  ArrayPuzzle p2 = new ArrayPuzzle();
  ....Display(p1)...Display(p2)...

  public static void Display(IPuzzle p){
    for (int r = 0; r < 3; r++)
      for (int c = 0;c<3;c++) {
        System.out.print(p.Tile(r,c));
        System.out.print(' ');
      }
  }
}

Method execution

public static void Display(IPuzzle p){
  for (int r = 0; r < 3; r++)
    for (int c = 0;c<3;c++) {
      System.out.print(p.Tile(r,c));
      System.out.print(' ');
    }
}

- Subtle point: which Tile method is invoked in code shown above?
  - Tile method in IntPuzzle class?*
  - What if object passed in is of type ArrayPuzzle?
  - Tile method in ArrayPuzzle class?*
  - What if object passed in is of type IntPuzzle?
  - Tile method in IPuzzle interface?*
  - Huh?*

- To understand this, let us look again at execution model.
Resolving the name “p.tile”

• Stack frame for invocation of Display has storage for variables p, r, c.
• Suppose method is passed an IntPuzzle object in parameter p as shown.
• Invocation “p.Tile(r,c)” in body of Display is executed as discussed earlier:
  – Look up method Tile in object O referenced by p.
  – Invoke that method passing it this (object O), r, c.
  – In our example, therefore, we would invoke the Tile method implemented in the IntPuzzle class.

Think

• Type of parameter p: Ipuzzle
  – Ipuzzle itself does not have a Tile method!
• Actual method that gets invoked is implemented sometimes in the ArrayPuzzle class and sometimes in the IntPuzzle class!
• Dynamic method binding:
  – Name “p.Tile” is not resolved to a single method.
  – In different invocations, name may be resolved to different methods.
• Method Display is sometimes said to be a polymorphic method.
  – Parameters are not restricted to be of a single type.

Other languages

• Dynamic method binding is a powerful mechanism that enables generic programming.
• In languages like C, effect of dynamic method binding can be obtained by passing function pointers, which may lead to weird bugs because it is not type-safe.
• Java-style dynamic method binding is more robust and less prone to errors.
  – Implementation of Java uses function pointers.
  – Java programmers cannot use function pointers directly.
  – Compare: GOTO vs. structured programming.
Another use of up-casting

- Sub-types and up-casting are useful for storing heterogeneous objects in data structures.
- Example:
  ```java
  IPuzzle[] AP = new IPuzzle[0..9];
  AP[0] = new IntPuzzle();
  AP[1] = new ArrayPuzzle();
  ```
- Note up-casting:
  - names AP[0] etc. are of type IPuzzle
  - Objects created on right hand sides are of sub-types of IPuzzle.

### instanceof

- Suppose we stick a bunch of ArrayPuzzle and IntPuzzle objects into an IPuzzle array AP.
- Suppose AP is passed to another method which walks over the array and counts how many IntPuzzle objects there are.
- How does this method examine the type of the objects stored in array AP?

  ```java
  boolean b = AP[i] instanceof IntPuzzle;
  # b will be true if AP[i] refers to IntPuzzle object; false otherwise
  // general syntax: reference instanceof className
  ```

Another useful thing

- String className = obj.getClass().getName();
  rhs expression returns a string with the name of class of the object named obj
  (eg). IPuzzle p = new IntPuzzle();
  System.out.println(p.getClass().getName());
  will print IntPuzzle
- Similar methods exist in Java for finding all interfaces implemented by an object etc.
- Other such methods can be found in Java class Class

Down-casting in Java

- Java permits down-casting but casting is specified explicitly.
  ```java
  public static void foo (IPuzzle p){
    if (p instanceof IntPuzzle)
      IntPuzzle ip = (IntPuzzle)p;
      ....}
  ```
- Compile-time: check that type of reference p is super-type of type of LHS name ip.
  - Making you write cast explicitly forces you to document down-casting.
- Run-time: check that type of object referenced by RHS is a sub-type of type of LHS name.
- Caution:
  - expression (IntPuzzle)p does not change the type of reference p
  - evaluating the expression returns a reference to the object referred to by p but with type IntPuzzle
  - Analogy: if x is an int variable, expression (x + 2) does not change value in x, it merely returns a new value
Down-casting in real life

• Canine names
  Canine Spot, Rover;
  • Spot = (Canine)Sandy;
    – Compile-time: is type of reference Sandy (Human) a super-type of type of Spot (Canine)? No. Compiler error.
  • George = (Male)Sandy;
    – Compile-time: is type of reference Sandy (Human) a super-type of type of George (Male)? Yes.
    – Run-time: is object referenced on RHS a sub-type of Male?
      • No: error - throw class cast exception.
      • Yes: everything is cool.

Why down-casting?

• Sometimes you want to
  – access an array of heterogenous objects
  – invoke a method on objects of some sub-type of array element type
  – method is not one of the interface methods, but is implemented only by that sub-type.
• In this situation, you can use down-casting.

Human

Male Female

Canine

Example

```java
void twister(IPuzzle[] AP) {
    for (int I = 0; I<AP.length; I++) {
        if (AP[I] instanceof IntPuzzle) {
            IntPuzzle p = (IntPuzzle)AP[I];
            p.Twist(); //method implemented only by IntPuzzle
        }
    }
}
```

Poor use of down-casting

```java
void mover(IPuzzle[] AP) {
    for (int I = 0; I<AP.length; I++) {
        if (AP[I] instanceof IntPuzzle) {
            ((IntPuzzle)AP[I]).Move('N');
        } else ((ArrayPuzzle)AP[I]).Move('N');
    }
}
```

• Heterogenous data in data structure AP.
• Do not use down-casting if you are invoking interface method (in this case, Move) on objects in data structure.
• Code on left will have to be modified if you add another class that implements interface.
• Code on right works without modification: code reuse is promoted.
Super-interfaces

• Suppose you want to extend the specification of an interface to include more methods.
  – IPuzzle: Scramble, Move, Tile
  – ImprovedPuzzle: Scramble, Move, Tile, SamLoyd

• Two approaches to writing down extended interface:
  – Start from scratch and write an interface
  – Extend the IPuzzle interface

Extending interfaces

interface IPuzzle{
    void Scramble();
    int Tile(int r, int c);
    boolean Move(char d);
}

interface ImprovedPuzzle extends IPuzzle{
    void SamLoyd();
}

//class that implements a sub-interface B must
//implement all methods in super-interfaces of B.

Super-interfaces

• IPuzzle is a super-interface of ImprovedPuzzle interface.
• ImprovedPuzzle interface is a sub-interface of IPuzzle.
• ImprovedPuzzle can be used as a type for variables like any other interface.
• It will be a sub-type of IPuzzle type.

Type Hierarchy

Interfaces:
- IPuzzle
- ImprovedPuzzle
- IRan
- IRaq

Classes:
- AClass
- BClass

class AClass implements ImprovedPuzzle, IRan {
    ……..
}
//There is no need to specify explicitly that AClass implements
//interface IPuzzle.
• Suppose class C implements a sub-interface IB. There is no need to declare super-
interface of IB in the “implements” clause of class C.
• Rules for up-casting and down-casting references stay the same as before.

• Interfaces have two main uses:
  – Software engineering:
    • Good fences make good neighbors.
  – Sub-typing:
    • Type of interface is super-type of type of class implementing that interface.
    • Use sub-types to write more generic, polymorphic code.
• Sub-typing is a central idea in programming languages.
• Sub-typing is sometimes referred to informally as is-a relationship.
  – (eg) Every Female is-a Human.

• Up-casting: super-type name on lhs of assignment
  – Example: Sandy = Laura;
  – Used in writing generic methods and for declaring data structures that can hold heterogenous data
  – Up-casting is always legal.
• Down-casting: sub-type name on lhs of assignment
  – Explicit cast required in Java.
  – Example: Laura = (Female) Sandy;
  – May or may not be legal.
  – Runtime check inserted: may throw exception
  – Less common than up-casting.

Up-casting and down-casting

• Consider statement  
  z = expression
• Three types:
  – Type of variable z
  – Type of expression
  – Type of object returned by expression at run-time
  – In general, these three types may be different as in second statement of example shown above.
    • Type of x is TB
    • Type of y is TA
    • Type of object is TC