CPS209: COMPUTER SCIENCE II

Interfaces and Polymorphism

Goals

- To learn about interfaces
- To be able to convert between class and interface references
- To understand the concept of polymorphism

Basic Idea

- Many general methods or programs are already written (e.g. in a java library), debugged, optimized
  - For example: *sort*, *binarySearch* in Collections/Arrays in the java.util package,

- Instead of writing your own general methods,

<u>use these existing general methods!</u>

- **HOW CAN WE USE THEM??**

Basic Idea

- We need to *connect* the code in <u>your class</u> to the general methods so you can then use them
- This "connection" is done via a Java *Interface*
- *It is useful to think of a Java interface as a connector*

First Step

- Look up documentation of a general method to see the *interface* it expects you to use

Basic Idea

- An *interface* specifies the **names** (not the code!!) of small *connecting methods* that **you need to implement**
  - **i.e. you write the code for these methods inside your class!!**

- *an interface* typically specifies:
  - names of methods
  - documentation of what each method is supposed to do
  - number of parameters and type of each parameter for each method
  - what value and type each method should return

Example

- Say you want to find the "maximum item" in a list of items
  - Example items: Employee, Rect, Coin, BankAccount
- **Question:** what does "maximum item" mean for each class?
- Approach 1:
  - You write a separate method for each of the classes above to search a list of objects of that class and find the "maximum" object
- **Approach 2:**
  - Write one general method that can be used for a list of **any type of object**

Approach2: Using Java Interfaces

- Let's say you are writing a class (example: class Bank) which contains an ArrayList of BankAccount objects
- You want to use an **existing (library) method** you found called:
  - findMaximum(ArrayList a)
- to find the BankAccount object in the ArrayList with the biggest balance
- Note the parameter type of findMaximum!

Approach2: Java Interfaces

- **Here's How (Read carefully!):**
  - A standard java *interface** lists the name and parameters of a simple connecting method called **getMeasure()** **that the findMaximum() method needs to call** for each BankAccount object so it can figure out how to find the maximum BankAccount object in the list.
  - Therefore, before you call findMaximum() and hand it your arraylist of BankAccount objects, **you must implement** this interface inside your class BankAccount:

* typically defined in a library

Approach2: Java Interfaces

- **Here's How (Read carefully!):**
  - What does "implement this interface" mean?
    - **You write code to implement the getMeasure() method inside class BankAccount!**
  - **Weird!** In order for you to use the findMaximum() method, you have to write some "connecting" code!
  - But only a little bit of simple code!!

Example Java Interface: Measurable (NOTE: defined in textbook by author)

```
// This interface can be used to find the size
// of an object. The method getMeasure() returns
// a double number which represents the size
// NOTICE: no code is specified, just the name and
// parameters of the "connecting" method

public interface Measurable
{
    double getMeasure();
}
```

Interface Measurable

- What should getMeasure() return for the following classes?:
  - BankAccount?
  - Coin?
  - Employee?
  - Rect?

- Hint: Ask yourself, how do I measure a Bank Account?
- How do I measure an Employee?

Interfaces vs Classes

- An interface type is similar to a class, but there are several important differences:

  1. All methods listed in an interface type are abstract; i.e. **they don't have any code!!**
  2. All methods listed in an interface type are automatically public
  3. An interface type does not have instance variables!!

Implementing an Interface

- Use the `implements` keyword to indicate that a class implements an interface type
- A class can implement more than one interface
  - e.g. class BankAccount implements Measurable, Comparable, Serializable
- A class must implement **all** the method names that are listed in the interface (write the code for them)

Examples

1.
```
public class BankAccount implements Measurable
{
    public double getMeasure()
    {
        return balance;
    }
    // Additional methods and fields of BankAccount
}
```

2.
```
public class Rect implements Measurable
{
    public double getMeasure()
    {
        return getArea();
    }
    // Additional methods and fields
}
```

Converting reference variables between Class types and Interface types

Converting between class and interface types

- You can convert a reference variable from a class type to an interface type, provided the class implements the interface:

- ```
  BankAccount account = new BankAccount(10000);
  Measurable  m = account; // OK
  ```

- ```
  Coin dime    = new Coin(0.1,"dime");
  Measurable m = dime; // Also OK
  ```

Converting between class and interface types

- Cannot convert between unrelated types

```
Rectangle  r = new Rectangle(5,10,20,30);
Measurable m = r; // COMPILE ERROR
```

- Why? Because the <u>java library class</u> `Rectangle` **does not** implement `Measurable` **interface**

- (Note: we use our own class Rect in examples)

What can you do with interface reference variables?

- `BankAccount account = new BankAccount(10000);`
  `Measurable  m = account; // OK`
- What can you do with reference variable `m`?
  Variable `m` is not of type `BankAccount`!
- `int size = m.getMeasure();  // OK`
- `m.deposit(55.0); // ERROR!!`

Casts

- You need a **cast** to convert a reference variable from an interface type to a class type
- You know it's referring to a BankAccount object, but the compiler doesn't! Apply a cast:

```
BankAccount b = (BankAccount) m;
b.deposit(55.0);
```

- If you are wrong and `m` **is not** referring to a BankAccount object, an exception is thrown (more on exceptions later)

Casts

- Difference with casting numbers and object references:
  - When casting number types you agree to the information loss
    - `int  x = (int) 5.5;`
  - When casting object types you agree to the risk of causing an exception

Polymorphism

- Interface reference variable holds reference to object of a class that implements the interface

- `Measurable m;`
- `m = new BankAccount(10000);`
- `m = new Coin(0.1,"dime");`

- **Note** that the **object** to which `m` refers (points to) is not of type `Measurable`; the variable `m` itself is of type Measurable. The type of the object that `m` is referring to is some class that implements the `Measurable` interface:
  - e.g. BankAccount, Coin

Polymorphism

- You can call any of the methods defined in the interface:
  - In this simple example, there is only one interface method

- `double x = m.getMeasure();`

- **Whose getMeasure() method is called?**

Polymorphism

- **Depends on the actual object!!**
  - If `m` refers to a bank account, calls `BankAccount's getMeasure()`
  - If `m` refers to a coin, calls `Coin's getMeasure()`
- *Polymorphism* (many shapes): Behavior can vary depending on the actual type of an object

Polymorphism

- Called *late binding*: resolved (i.e. JVM figures out) at runtime which method to call
- Different from overloading; overloading is resolved by the compiler (*early binding*)

Self Check

1. Why is it impossible to construct a `Measurable` object?

2. Why can you nevertheless declare a variable whose type is `Measurable`?

3. What do overloading and polymorphism have in common? Where do they differ?

Answers

1. `Measurable` is an interface. Interfaces have no fields and no method implementations.

2. That variable never refers to a `Measurable` object. It refers to an object of some class—a class that implements the `Measurable` interface.

Answers

3. Both describe a situation where one method name can denote multiple methods. However, overloading is resolved early by the compiler, by looking at the types of the parameter variables. Polymorphism is resolved late, by looking at the type of the implicit parameter object just before making the call.

Interface Comparable

- Defined in java.lang.Comparable

```
// This method used to compare objects with other objects.
// Parameters: other  The object to be compared
// Returns: A negative integer if this object is less than the other,
   zero if they are equal, or a positive integer otherwise

public Interface Comparable
{
   int compareTo(Object other);
}
```

Interface Comparable

```
public class BankAccount implements Comparable
{
   private double balance;

    public BankAccount(double initialBalance)
    {
       balance = initialBalance;
    }
   // other BankAccount methods
    // .....

    public int compareTo(Object other)
    {
      BankAccount otherBA = (BankAccount) other;

      if (this.balance > otherBA.balance) return 1;
      else if (this.balance < otherBA.balance) return -1;
      else return 0;
    }
}
```

Interface Comparable ***Typed***

```
public class BankAccount implements Comparable<BankAccount>
{
   private double balance;

    public BankAccount(double initialBalance)
    {
       balance = initialBalance;
    }
   // other BankAccount methods
    // .....

    public int compareTo(BankAccount otherBA)
    {
      if (this.balance > otherBA.balance) return 1;
      else if (this.balance < otherBA.balance) return -1;
      else return 0;
    }
}
```

Interface Comparable

- See code example

Interface Comparator

- See Country Example

Other Uses of Interfaces in OO Programming

Interfaces for Implementing *Behaviors*

- **<u>Sometimes</u>** inheritance *is-a* relationship is restrictive as a high-level code structuring mechanism
  - Not everything can be nicely described as an **is-a** relationship hierarchy
- Interfaces add another complimentary structuring mechanism
  - Provides flexibility without destroying simplicity
  - **Use interfaces together with inheritance**
- You can typically think of interfaces as a way to add *behaviors* to your classes
  - e.g. games – class Enemy implements Shooting

Example from Unity Game Engine

- Unity uses a "component" system which is more flexible for games than inheritance
  - game objects are constructed through *composition* rather than inheritance
- In Java, an interface can be used to add components (behaviors) instead of complex inheritance "branches"