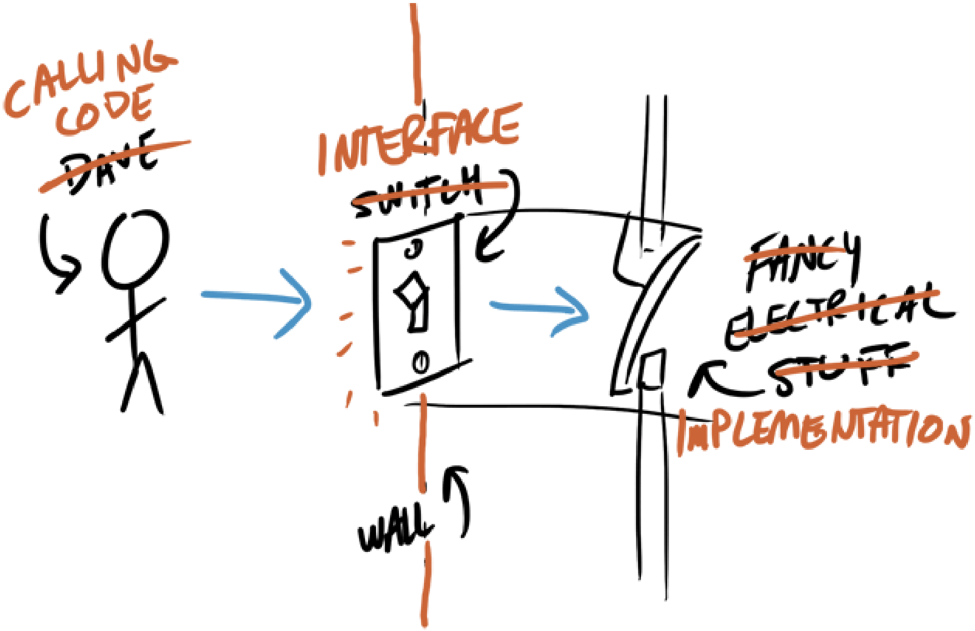


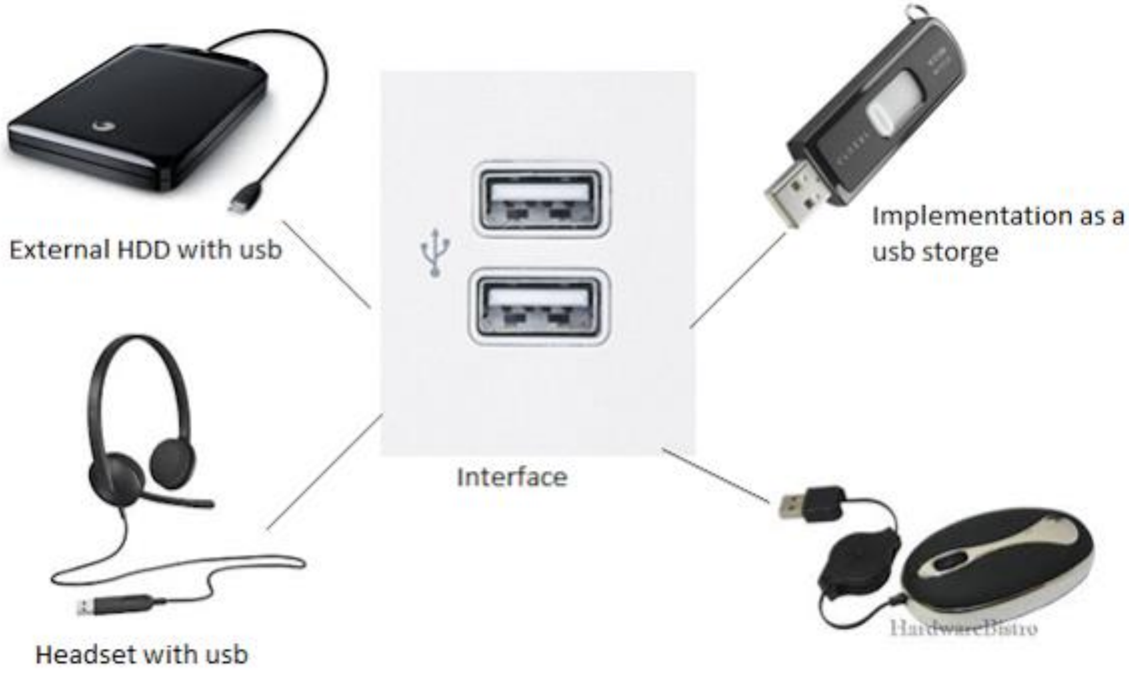
Interfaces

ITCS 209

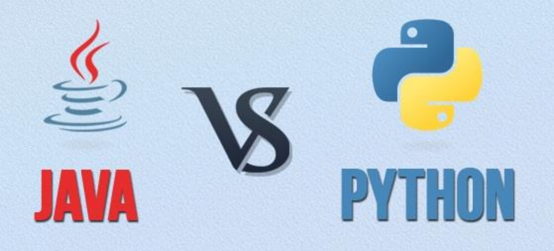
Assistant Prof. Dr. Suppawong Tuarob

Faculty of Information and Communication Technology

Faculty of ICT
Mahidol University


External HDD with usb
Implementation as a
usb storge
Interface
Headset with usb
HardwareBistro

Java vs Python



- Python is very *dynamic*—classes and methods can be added, modified, and deleted as the program runs
  - If you have a call to a function that doesn't exist, Python will give you a **runtime** error *when you try to call it*

- In Java, everything has to be defined before the program begins to execute
  - If you have a call to a function that doesn't exist, the compiler marks it as a **syntax error**
  - Syntax errors are far better than runtime errors
    - Among other things, they won't make it into distributed code

  - What if you know that a Dog must be able to bark(), though the method has not yet implemented?
  - To achieve this, Java requires a special kind of classes that do not have complete implementation.

Abstract methods

- You can *declare* an object without *defining* it:

```
Person p;
```

- Similarly, you can declare a *method* without defining it:

```
public abstract void draw(int size);
```

  - Notice that the body of the method is missing

- A method that has been declared but not defined is an abstract method

Abstract classes I

- Any class containing an abstract method is an abstract class

- You must declare the class with the keyword abstract:

  ```
  abstract class MyClass {...}
  ```

- An abstract class is *incomplete*
  - It has "missing" method bodies

- You cannot instantiate (create a new instance of) an abstract class

Abstract classes II

- You can extend (subclass) an abstract class
  - If the subclass defines all the inherited abstract methods, it is "complete" and can be instantiated
  - If the subclass does *not* define all the inherited abstract methods, it too must be abstract

- You can declare a class to be abstract even if it does not contain any abstract methods
  - This prevents the class from being instantiated

Why have abstract classes?

- Suppose you wanted to create a class Shape, with subclasses Oval, Rectangle, Triangle, Hexagon, etc.

- You don't want to allow creation of a "Shape"
  - Only *particular* shapes make sense, not *generic* ones
  - If Shape is abstract, you can't create a new Shape
  - You *can* create a new Oval, a new Rectangle, etc.

- Abstract classes are good for defining a general category containing specific, "concrete" classes

An example abstract class

-
  ```
  public abstract class Animal {
      abstract int eat();
      abstract void breathe();
  }
  ```

- This class cannot be instantiated
- Any non-abstract subclass of Animal must provide the eat() and breathe() methods

Why have abstract methods?

- Suppose you have a class Shape, but it *isn't* abstract
  - Shape should *not* have a draw() method
  - Each subclass of Shape *should* have a draw() method

- Now suppose you have a variable Shape figure; where figure contains some subclass object (such as a Star)
  - It is *a syntax error* to say figure.draw(), because the Java compiler can't tell in advance what kind of value will be in the figure variable
  - A class "knows" its superclass, but doesn't know its subclasses
  - An object knows its class, but a class doesn't know its objects

- **Solution:** Give Shape an *abstract* method draw()
  - Now the class Shape is abstract, so it can't be instantiated
  - The figure variable cannot contain a (generic) Shape, because it is impossible to create one
  - Any object (such as a Star object) that *is* a (kind of) Shape *will* have the draw() method
  - The Java compiler can depend on figure.draw() being a legal call and does not give a syntax error

A problem

- class Shape { … }
- class Star extends Shape {
  ```
      void draw() { … }
      …
  }
  ```
- class Crescent extends Shape {
  ```
      void draw() { … }
      …
  }
  ```
- Shape someShape = new Star();
  - This is legal, because a Star ***is*** a Shape

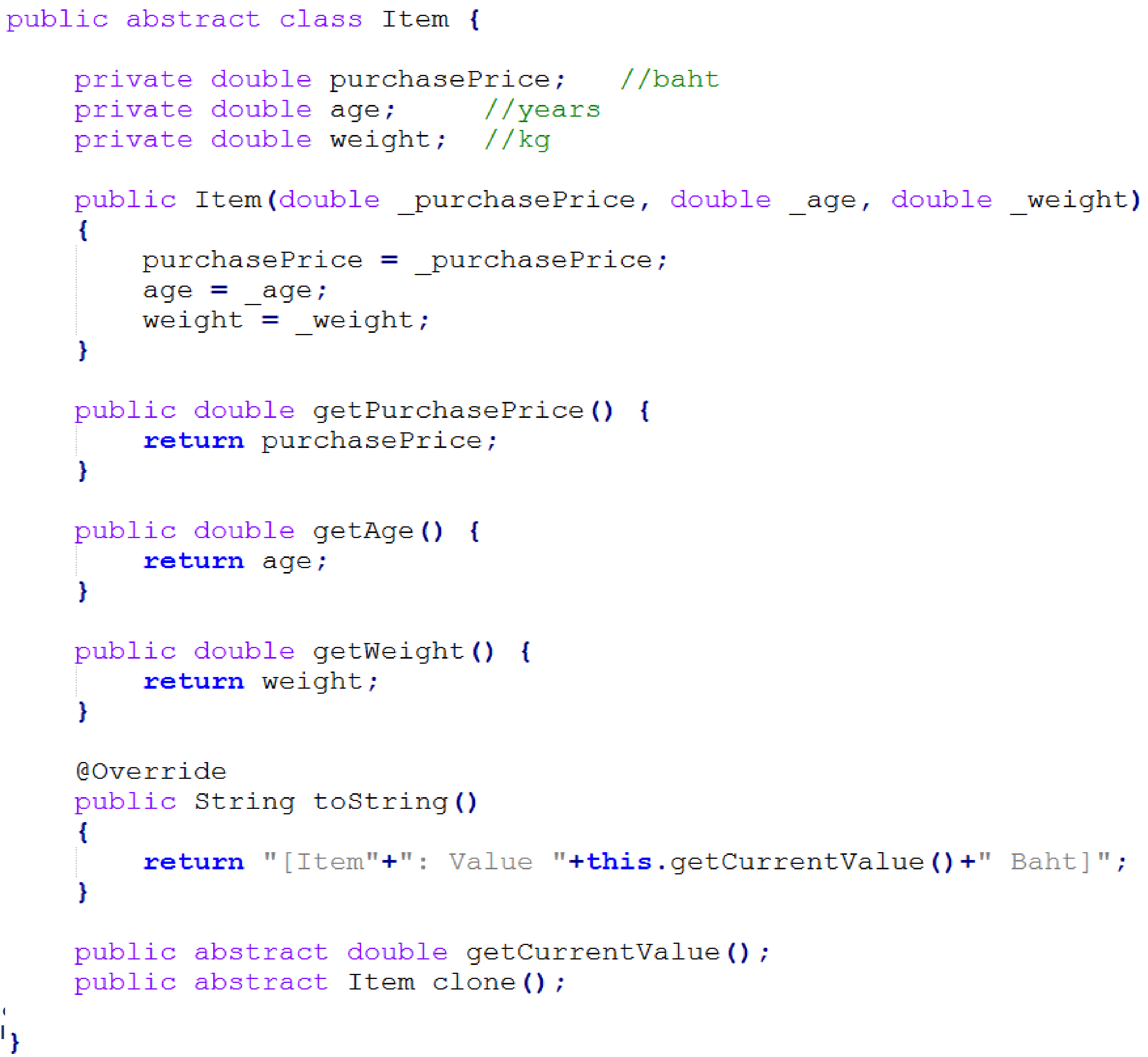
- someShape.draw();
  - This is a syntax error, because *some* Shape might not have a draw() method
  - Remember: ***A class knows its superclass, but not its subclasses***

A solution

-
  ```
  abstract class Shape {
      void draw();
  }
  ```

-
  ```
  class Star extends Shape {
      void draw() { ... }
      ...
  }
  ```

-
  ```
  class Crescent extends Shape {
      void draw() { ... }
      ...
  }
  ```
- `Shape someShape = new Star();`
  - This is legal, because a Star ***is*** a Shape
  - However, `Shape someShape = new Shape();` is *no longer* legal
- `someShape.draw();`
  - This is legal, because every actual instance *must* have a `draw()` method



```java
public abstract class Item {

    private double purchasePrice;   //baht
    private double age;      //years
    private double weight;  //kg

    public Item(double _purchasePrice, double _age, double _weight)
    {
        purchasePrice = _purchasePrice;
        age = _age;
        weight = _weight;
    }

    public double getPurchasePrice() {
        return purchasePrice;
    }

    public double getAge() {
        return age;
    }

    public double getWeight() {
        return weight;
    }

    @Override
    public String toString()
    {
        return "[Item"+": Value "+this.getCurrentValue()+" Baht]";
    }

    public abstract double getCurrentValue();
    public abstract Item clone();
}
```

Interfaces

- An interface declares (describes) methods but does not supply bodies for them

```
 interface KeyListener {
      public void keyPressed(KeyEvent e);
      public void keyReleased(KeyEvent e);
      public void keyTyped(KeyEvent e);
}
```

- All the methods are implicitly public and abstract
  - You can add these qualifiers if you like, but why bother?
- You cannot instantiate an interface
  - An interface is like a *very* abstract class—*none* of its methods are defined
- An interface may also contain constants (final variables)

Implementing an interface I

- You extend a class, but you implement an interface
- A class can only extend (subclass) one other class, but it can implement as many interfaces as you like
- Example:

```
class MyListener
        implements KeyListener, ActionListener { … }
```

Implementing an interface II

- When you say a class implements an interface, you are promising to *define* all the methods that were *declared* in the interface
- Example:

```
class MyKeyListener implements KeyListener {
        public void keyPressed(KeyEvent e) {...};
        public void keyReleased(KeyEvent e) {...};
        public void keyTyped(KeyEvent e) {...};
}
```

  - The "..." indicates actual code that you must supply
- Now you can create a new MyKeyListener

Partially implementing an Interface

- It is possible to define some but not all of the methods defined in an interface:

```
abstract class MyKeyListener implements KeyListener {
    public void keyTyped(KeyEvent e) {...};
}
```

- Since this class does not supply all the methods it has promised, it is an abstract class
- You must label it as such with the keyword abstract
- You can even *extend* an interface (to add methods):
  - interface FunkyKeyListener extends KeyListener { ... }

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”
  - In writing Applets, it is common to have one class implement several different listeners
  - Example:

```
class MyApplet extends Applet
        implements ActionListener, KeyListener {
  ...
 }
```

- Reason 2: You can write methods that work for more than one kind of class

How to use interfaces

- You can write methods that work with more than one class
-
  ```
  interface RuleSet { boolean isLegal(Move m, Board b);
                      void makeMove(Move m); }
  ```
  - Every class that implements RuleSet must have these methods
-
  ```
  class CheckersRules implements RuleSet { // one implementation
      public boolean isLegal(Move m, Board b) { ... }
      public void makeMove(Move m) { ... }
  }
  ```
- `class ChessRules implements RuleSet { ... } // another implementation`
- `class LinesOfActionRules implements RuleSet { ... } // and another`
- `RuleSet rulesOfThisGame = new ChessRules();`
  - This assignment is legal because a rulesOfThisGame object *is* a RuleSet object
- `if (rulesOfThisGame.isLegal(m, b)) { makeMove(m); }`
  - This statement is legal because, *whatever* kind of RuleSet object rulesOfThisGame is, it *must* have isLegal and makeMove methods

instanceof

- instanceof is a keyword that tells you whether a variable "is a" member of a class or interface
- For example, if

```
class Dog extends Animal implements Pet {...}
Animal fido = new Dog();
```

  then the following are all true:

```
fido instanceof Dog
fido instanceof Animal
fido instanceof Pet
```

- instanceof is seldom used
  - When you find yourself wanting to use instanceof, think about whether the method you are writing should be moved to the individual subclasses

Interfaces, again

- When you implement an interface, you promise to define *all* the functions it declares

- There can be a *lot* of methods

```
interface KeyListener {
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
}
```

- What if you only care about a couple of these methods?

Adapter classes

- Solution: use an adapter class
- An adapter class implements an interface and provides empty method bodies

```
class  KeyAdapter implements KeyListener {
    public void keyPressed(KeyEvent e) {………….xxxxxx……… };
    public void keyReleased(KeyEvent e) { };
    public void keyTyped(KeyEvent e) { };
}
```

- You can override only the methods you care about
- This isn’t elegant, but it does work
- Java provides a number of adapter classes