Object Orientation in Java

Dr. Eliahu Khalastchi
2017

המסלול האקדמי
המכללה למינהל

Recap…

- Java is an OOP language (almost pure)
- OOP languages uses
  - Objects as data structures (fields and methods)
  - Data abstraction, encapsulation, modularity, polymorphism and inheritance
- An object is an instance of a class
- A class is loaded only once while its instances can be as many as we whish

Recap…

- An instance is created with “new” command
- “new” Calls the class’s constructor
- Members are called from objects
- Static members are called from the class

```java
public class HelloWorld {
 public void print() {
   System.out.println("Hello World!");
 }
}
```

```java
public class Run {
 public static void main(String[] args) {
  HelloWorld h=new HelloWorld();
  h.print();
 }
}
```

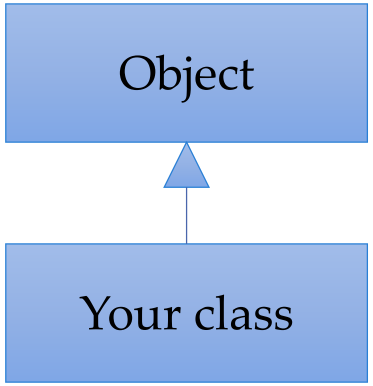
Recap…

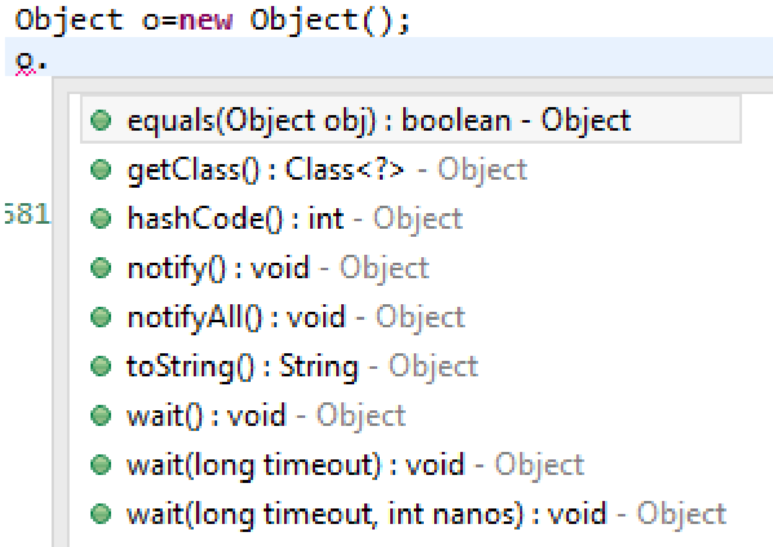
- Given a class A with static method s() and a method m()
- We define A a;

- what of the following will work?
  - a.s();
  - a.m();
  - A.s();
  - A.m();

The Object class

- Every class in Java inherited the class *Object*
- *Object* is the most general class
- Used for general purpose, e.g.,
    - method(Object arg0) – arg0 can be any object
    - Object array[] – can store any objects
- Object’s methods:



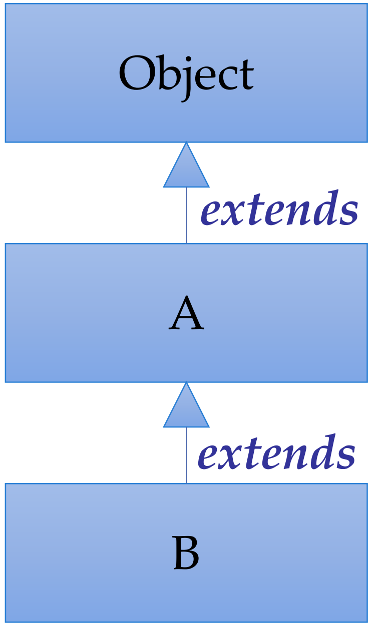


```
Object o=new Object();
o.
```

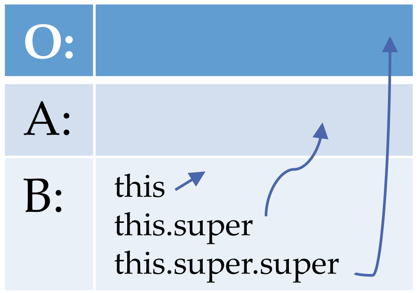
- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

Inheritance

- In order to avoid the *diamond of death* Java allows a class to inherit only 1 class
- Inheritance keyword is ***extends***
- A *sub-class* extends a *super-class*
- ***This*** refers to the current instance of a class
- ***super*** refers to the super-class's instance



new B();



O:

A:

B: this
this.super
this.super.super

Inheritance

- Class A is a super-class of B:

Not a default constructor

Why do we need this?

```
public class A {
 private int x,y;
 public A(int x,int y){
  this.x=x;
  this.y=y;
 }

 @Override
 public String toString(){
 return "implementad on class A\n"+
        "called from "+getClass()+
        ", x="+x+" y="+y+"\n";
 }
}
```

```
public class B extends A{
 public B(int x, int y) {
  super(x, y);
 }
}
```

```
A b=new B(10,10);
B b1=new B(11,11);
System.out.println(b);
System.out.println(b1);
```

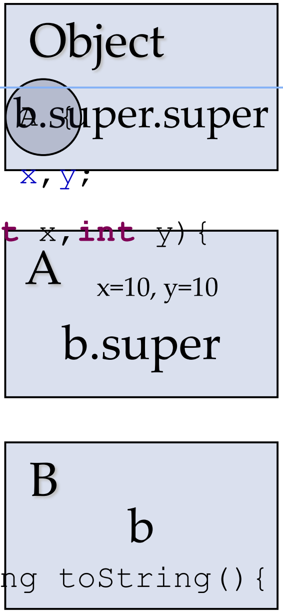
```
implementad on class A
called from class B, x=10 y=10

implementad on class A
called from class B, x=11 y=11
```

Every Ctor must initialize the A part.
super(x,y) must come first; it is like a C++ initialization line…

What will be the output?

Inheritance

getClass() returns the Runtime class of the object.



getClass ?

toString ?

```java
public class A{
 private int x,y;
 public A(int x,int y){
  this.x=x;
  this.y=y;
 }

 @Override
 public String toString(){
 return "implementad on class A\n"+
        "called from "+getClass()+
        ", x="+x+" y="+y+"\n";
 }
}
```

```java
public class B extends A{
 public B(int x, int y) {
  super(x, y);
 }
}
```

```java
A b=new B(10,10);
B b1=new B(11,11);
System.out.println(b);
System.out.println(b1);
```

```
implementad on class A
called from class B, x=10 y=10

implementad on class A
called from class B, x=11 y=11
```

Inheritance

- A *final* class can’t be extended

```
public final class A {
 private int x,y;
 public A(int x,int y){
  this.x=x;
  this.y=y;
 }
 @Override
 public String toString(){
 return "implementad on class A\n"+
        "called from "+getClass()+
        ", x="+x+" y="+y+"\n";
 }
}
```

```
public class B extends A{
 public B(int x, int y) {
  super(x, y);
 }
}
```

Inheritance

- A sub-class can override anything except:
  - ***private***, ***final*** or ***static*** methods
- What methods can be overridden?

```
public class C {
 private void print1() {
  System.out.println("1");
 }
 public final void print2() {
  System.out.println("2");
 }
 public static void print3() {
  System.out.println("3");
 }
}
```

```
public class D extends C{
 private void print1() {
  System.out.println("1d");
 }
 public void print2() {
  System.out.println("2d");
 }
 public void print3() {
  System.out.println("3d");
 }
}
```

Inheritance

- All the methods in Java are like C++ virtual methods – **dynamic binding!**

Java

```
public class Test {

 public class A {
   public void print(){
      System.out.println("A");
   }
 }
 public class B extends A{
   public void print(){
      System.out.println("B");
   }
 }

 public void testMe() {
   A a=new B();
   a.print(); // B
 }
}
```

≡

C++

```
class Test{
    public:
    class A{
        public:
        void virtual print(){
            cout<<"A"<<endl;
        }
    };
    class B: public A{
        public:
        void virtual print(){
            cout<<"B"<<endl;
        }
    };
    void testMe(){
        A* a=new B();
        a->print(); // B
    }
};
```

Abstract classes

- Until now we’ve seen what can and cannot be extended
- But what if we want to **force** an implementation of a subclass?
- When ***abstract*** is attached to a method:
  - It is left unimplemented
  - The class becomes abstract, and cannot be instanced
  - Only a subclass that implemented the abstract method can be instanced

Abstract classes

- Abstract classes cannot be instanced

```
public abstract class MyAbstract {
 String name;
 public MyAbstract(String name){
  this.name=name;
 }
 public void welcome(){
  System.out.println("hello "+name);
 }
}
```

```
MyAbstract a=new MyAbstract(); // error!
```

Abstract classes

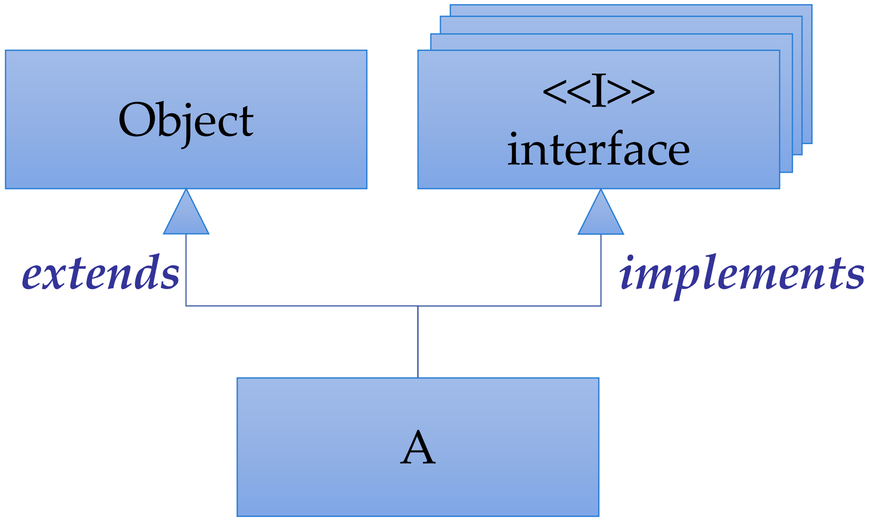
- A concrete subclass can be instanced
  - Inherit all implemented methods as usual
  - But must implement all abstract methods

```
public abstract class MyAbstract {
 String name;
 public MyAbstract(String name){
  this.name=name;
 }
 public abstract void welcome();
}
```

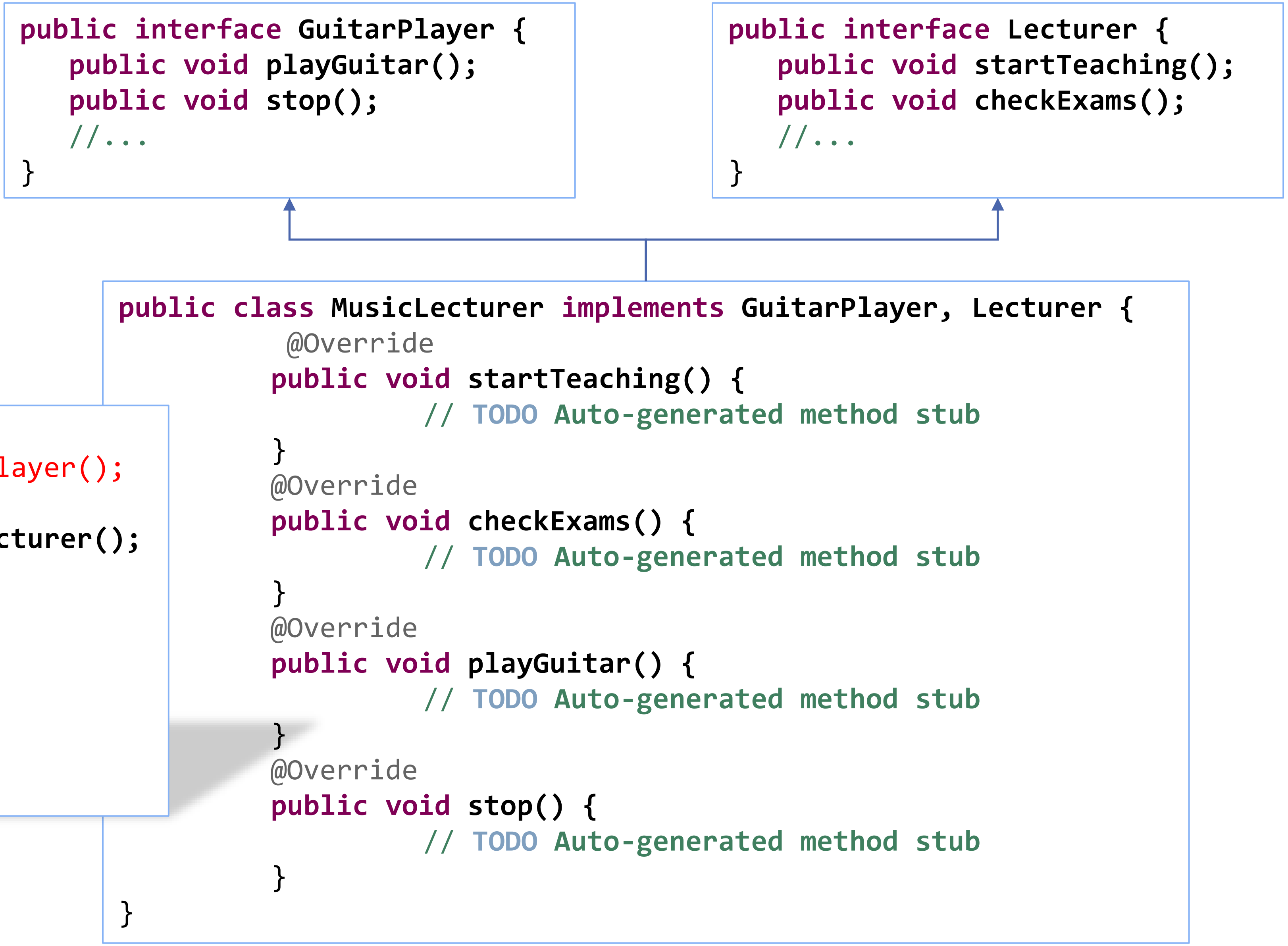
```
public class MyConcrete extends MyAbstract{
 public MyConcrete(String name) {
  super(name);
 }
 @Override
 public void welcome() {
  System.out.println("wlcome "+name);
 }
}
```

Interfaces

- Interfaces are pure abstract classes
- Defined by the keyword ***interface***
- Interfaces are implemented (not extended)
  - by the keyword is ***implements***
- Multiple implementation of interfaces is allowed in Java
- Interfaces are the common language in which objects interact



Interfaces



```
public interface GuitarPlayer {
   public void playGuitar();
   public void stop();
   //...
}
```

```
public interface Lecturer {
   public void startTeaching();
   public void checkExams();
   //...
}
```

```
public class MusicLecturer implements GuitarPlayer, Lecturer {
          @Override
         public void startTeaching() {
                  // TODO Auto-generated method stub
         }
         @Override
         public void checkExams() {
                  // TODO Auto-generated method stub
         }
         @Override
         public void playGuitar() {
                  // TODO Auto-generated method stub
         }
         @Override
         public void stop() {
                  // TODO Auto-generated method stub
         }
}
```

```
// we can't instantiate an interface
//GuitarPlayer guitarPlayer=new GuitarPlayer();
// this is OK:
GuitarPlayer guitarPlayer = new MusicLecturer();
guitarPlayer.playGuitar();
// without casting, we can only expect
// the functionality of a guitar player
//guitarPlayer.startTeaching();
// this is OK:
Lecturer lecturer=new MusicLecturer();
lecturer.startTeaching();
```

How do we use it all?

- We use inheritance when
  - a subclass “is a” type of the super-class
  - We want to change or extend the super-class’s implementation

- Higher up on the hierarchy classes become more abstract
  - They provide data and functionality common to all the derived classes
  - They define how the concrete class should interact
  - They are not suppose to “know” concrete classes

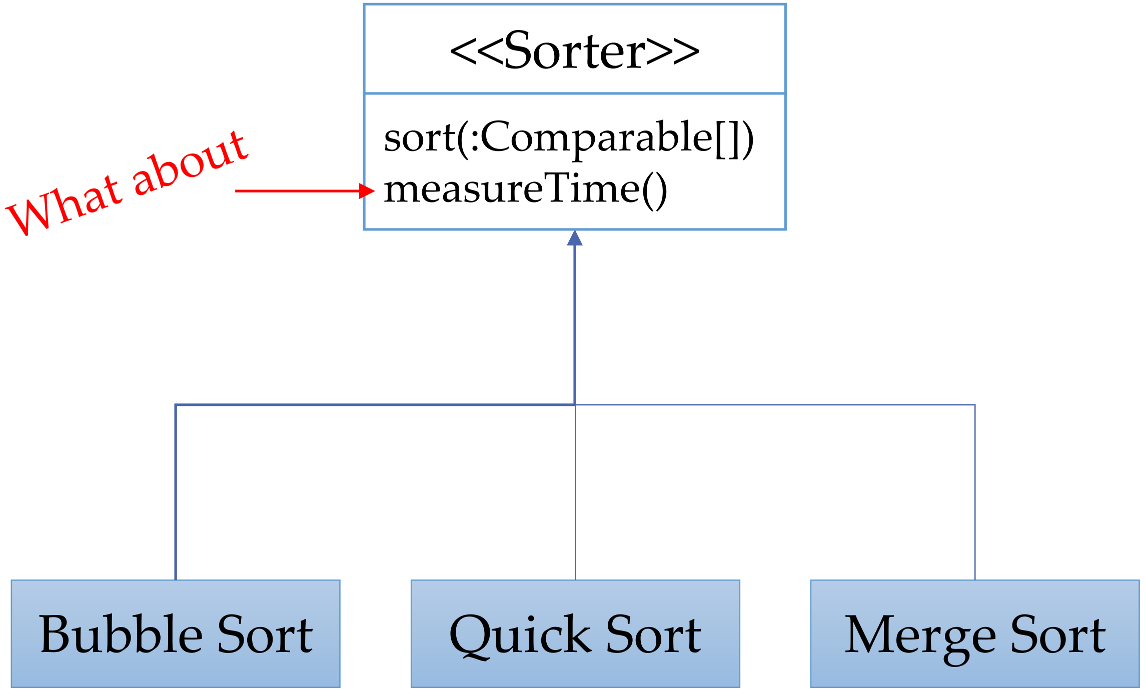
How do we use it all?

- We use interfaces
  - To define what functionality we expect from a given object
  - When we want to allow another extension

- We use static members when
  - We don't want an instance to use them
  - We want them to load only once
  - We want to define something that is the same for every object

- Finally, ***design patterns*** provides common and very good solutions for common problems… (next lessons)

Example

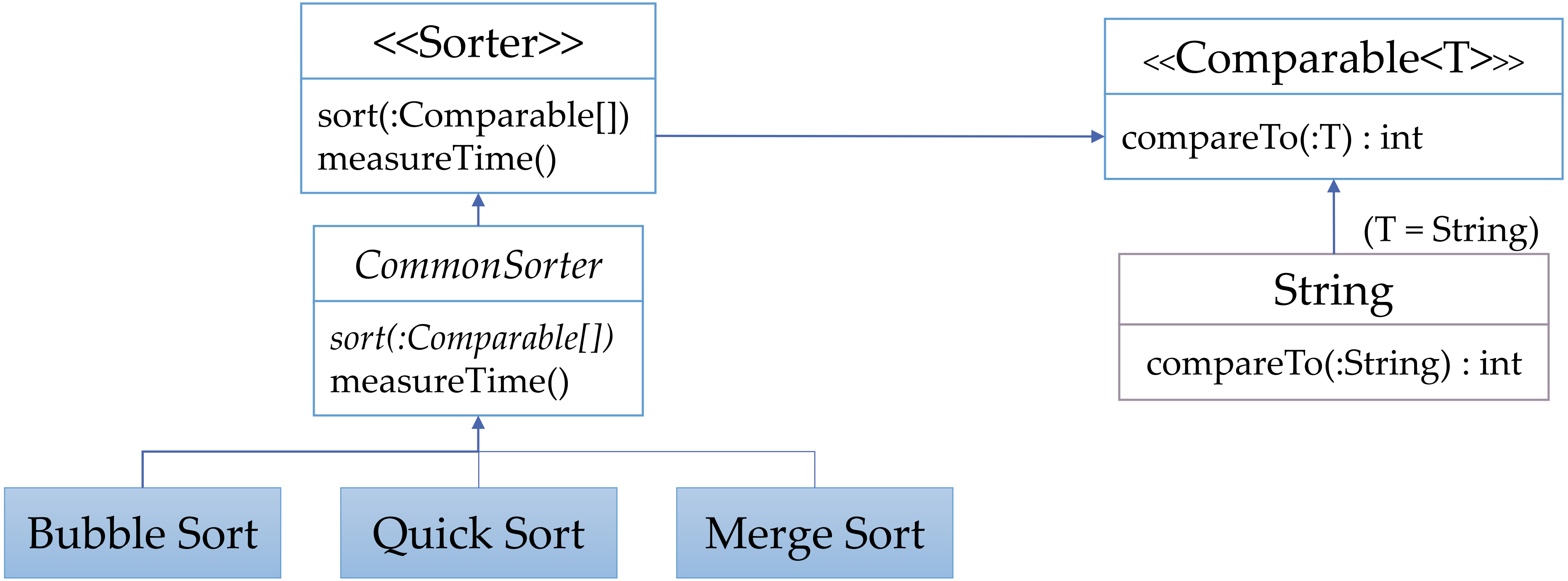


```
BubbleSort sorter=new BubbleSort();
sorter.sort(…);
// and other methods specific for bubble sort
```

```
Sorter sorter=new BubbleSort();
sorter.sort(…);
// and other methods which apply to any sorter
```

Any Sorter!!

Example



<<Sorter>>

sort(:Comparable[])
measureTime()

CommonSorter

sort(:Comparable[])
measureTime()

Bubble Sort

Quick Sort

Merge Sort

<<Comparable<T>>>

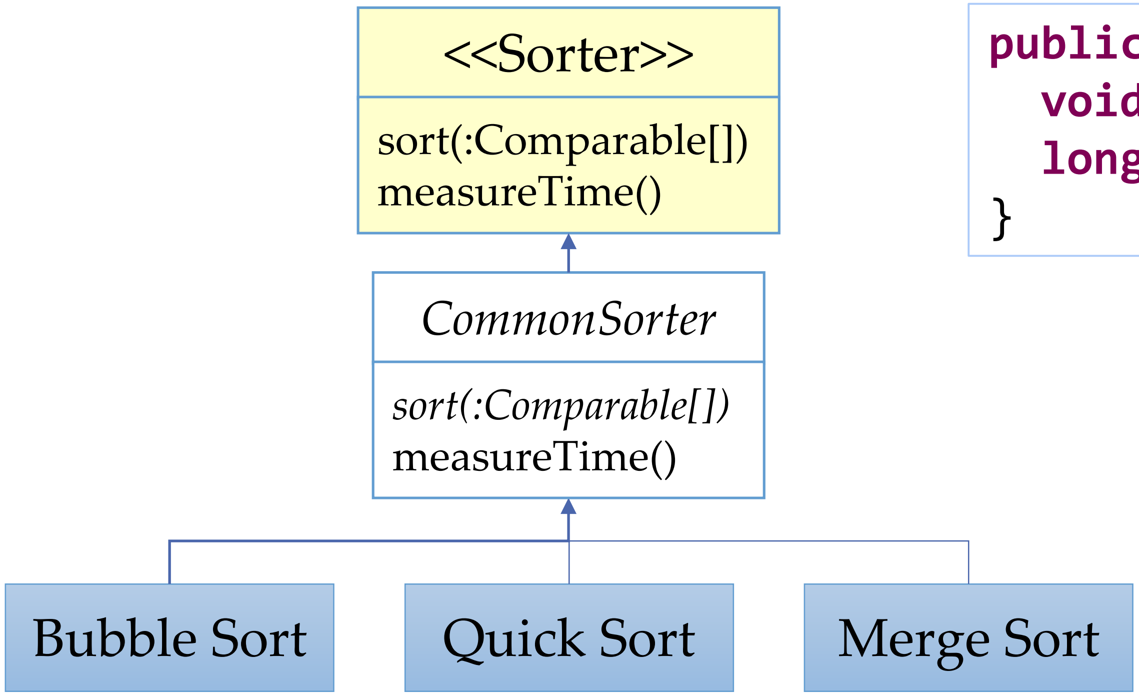
compareTo(:T) : int

(T = String)

String

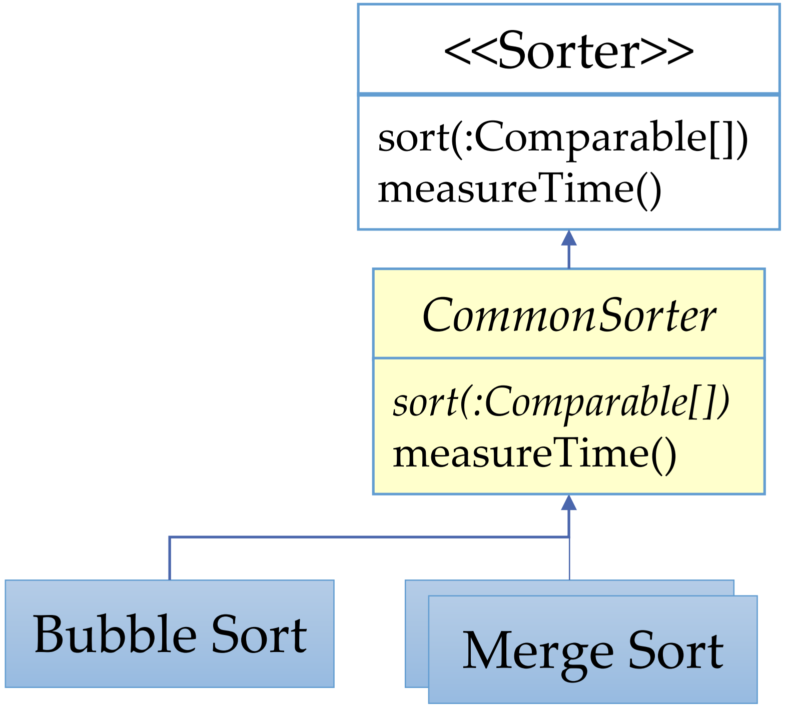
compareTo(:String) : int

Example



```
public interface Sorter<T>{
  void sort(Comparable<T>[] comparables);
  long measureTime(Comparable<T>[] comparables);
}
```

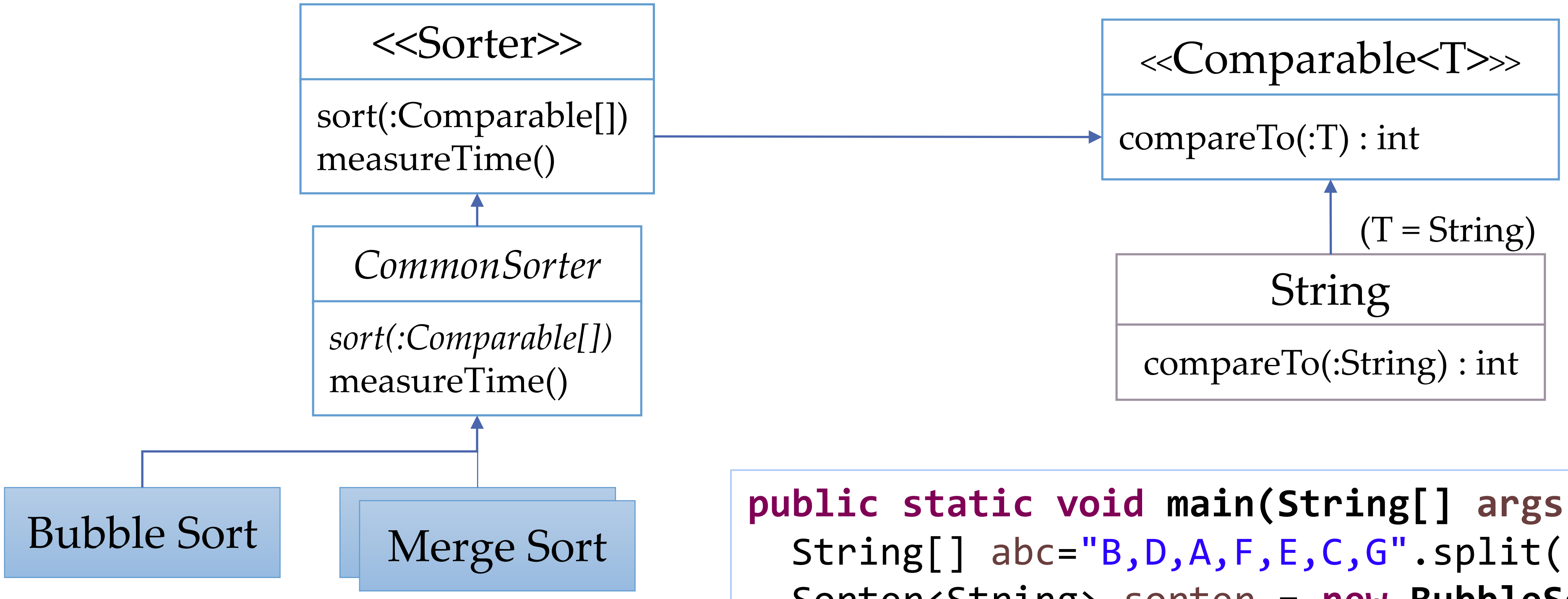
Example



```
public abstract class CommonSorter<T> implements Sorter<T> {
  @Override
  public long measureTime(Comparable<T>[] comparables) {
    long time0=System.currentTimeMillis();
    sort(comparables);
    return System.currentTimeMillis()-time0;
  }
}
```

```
public class BubbleSort<T> extends CommonSorter<T>{
  @Override
  public void sort(Comparable<T>[] comparables) {
    //...
    if(comparables[i].compareTo(comparables[i+1]))>0)
      switchCells(comparables[i],comparables[i+1]);
    //...
  }
}
```

Example



```java
public static void main(String[] args) {
  String[] abc="B,D,A,F,E,C,G".split(",");
  Sorter<String> sorter = new BubbleSort<String>();
  sorter.sort(abc);
}
```