Classes, interfaces and inheritance

Summary

- **`extends`** is a relation defined over classes and over interfaces
- **`implements`** is a relation between classes and interfaces
- an interface extends zero or more interfaces
- an interface contains by default all public methods of `Object`
- except for `Object`, a class always extends a single class
- a class implements zero or more interfaces

Dynamic dispatch of object methods

Example

```
public class TimerClass implements Timer {
    ...
    public boolean isRunning() {
        return this.getTime() > 0;
    }
    // which method will be called with ’this.isRunning()’?
    public void tick() {
        if (this.isRunning()) // more general than ’this.getTime() > 0’
            this.time--;
    }
}
```

Remarks

- **this**.`isRunning()` is more general than **this**.`getTime()> 0`
- indeed, object method `isRunning()` can be redefined in subclasses
- consequence: no need to redefine `tick()` in `StoppableTimerClass`

Dynamic dispatch of object methods

Example

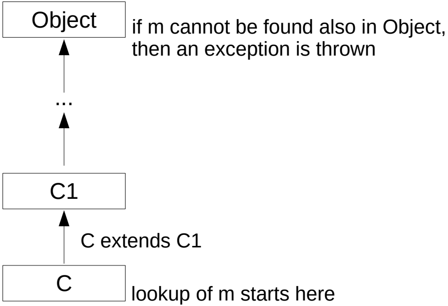
```
public class TimerClass implements Timer {
    ...
    public boolean isRunning() {
        return this.getTime() > 0;
    }
    // which method will be called with ’this.isRunning()’?
    public void tick() {
        if (this.isRunning()) // more general than ’this.getTime() > 0’
            this.time--;
    }
}
```

Remarks

Object method `tick()` can be inherited in subclasses, therefore:

- the static type of **`this`** is `TimerClass`
- the dynamic type of **`this`** can be a subtype of `TimerClass`

Dynamic dispatch of object methods


Object
if m cannot be found also in Object, then an exception is thrown
...
C1
C extends C1
C
lookup of m starts here

Rule for object method call

Method call o.m()

1. let C be the class of the target object o (= its dynamic type)
2. lookup of m starts from C
3. superclasses of C are traversed up to `Object` until m is found
4. if found, then m is run, else `NoSuchMethodError` is thrown

Dynamic dispatch of object methods

Demo 1

```
Timer t = new StoppableTimerClass();
t.isRunning(); // method found in class 'StoppableTimerClass'
t.getTime();   // method found in class 'TimerClass'
t.equals(t);   // method found in class 'Object'
```

Recall

- except for `Object`, every class must extend a single class
- if no direct superclass is specified, then `Object` is implicitly extended

Example

```
public class TimerClass implements Timer {...}

// equivalent declaration
public class TimerClass extends Object implements Timer {...}
```

Dynamic dispatch of object methods

Limitations

- Dynamic dispatch is not supported for
  - object methods called with **`super`**
  - class methods
  - object and class fields
- class methods, object and class fields can be inherited but not redefined

Dynamic dispatch of object methods

Example with **super**

```
public class StoppableTimerClass extends TimerClass {
    ...
    @Override
    public boolean isRunning() {
       // calls  ’isRunning’ of ’TimerClass’ on target object ’this’
       return super.isRunning() && !this.stopped();
    }
 }
```

Remarks

- with **super** dispatch of object methods is always static
- **super**.isRunning() always calls isRunning() of TimerClass
- with **super** the called method does not depend on the dynamic type of **this**

Constructors and inheritance

Important rule

- constructors are not inherited by subclasses
- consequence:
  - new constructors need to be added in each subclass
  - superclass costructors have to be reused
  - to be reused, constructors cannot be private, but, at least protected

Example

```
public class StoppableTimerClass extends TimerClass implements StoppableTimer {

    private boolean stopped;

    public StoppableTimerClass() { // calls 'TimerClass()' implicitly
    }
    public StoppableTimerClass(int minutes) {
        super(minutes);             // calls 'TimerClass(int)'
    }
    public StoppableTimerClass(StoppableTimer other) {
        super(other);               // calls 'TimerClass(Timer)'
        this.stopped = other.stopped();
    }
    ...
}
```

Constructors and inheritance

General rules

- every constructor is responsible for the initialization of the object fields declared in its class
- initialization of the object fields in a class requires first initialization of object fields in its direct superclass

Java specific rules

- the body of a constructor can only begin with
  - either `this(...)` (=call of another constructor of the same class)
  - or `super(...)` (=call of a constructor of the direct superclass)
- if the body does not begin with `this(...)` or `super(...)`, then the implicit call `super()` is inserted
- `this(...)` and `super(...)` can only be placed in the first line of a constructor body

Constructors and inheritance

Demo 2

```
public class TimerClass implements Timer {
   private int time = 60;
        ...
   public TimerClass(Timer other) {                   // copy constructor
        this.time = other.getTime();
    }
        ...
}
public class StoppableTimerClass extends TimerClass implements StoppableTimer {

    private boolean stopped;
        ...
    public StoppableTimerClass(StoppableTimer other) {   // copy constructor
        super(other);
        this.stopped = other.stopped();
    }
        ...
    public static void main(String[] args) {
        StoppableTimer t1 = new StoppableTimerClass(2);
        t1.stop();
        StoppableTimer t2 = new StoppableTimerClass(t1); // see next slide
     }
}
```

Constructors and inheritance

Single steps

1. a new object of `StoppableTimerClass` is created with `time=0`, `stopped=`**`false`**
2. `StoppableTimerClass(StoppableTimer)` is called
3. `TimerClass(Timer)` is called
4. `Object()` is called
5. the object field initializer for `time` is executed: `time=60`
6. the rest of the body of `TimerClass(Timer)` is executed: `time=120`
7. no object field initializers are defined for `StoppableTimer`
8. the rest of the body of `StoppableTimerClass(StoppableTimer)` is executed: `stopped=`**`true`**

Object creation and initialization: the full picture

0. create a new instance of C with all declared and inherited object fields initialized with their defaut values

1. evaluates arguments and pass them to the parameters of the constructor

2. if the constructor starts with *this(...)*, then call the constructor in the same class (starting from point 1)

3. if the constructor starts with super*(...)*, then call the constructor in the superclass (starting from point 1)

4. execute object field initializers in the standard order

5. execute the rest of the body of the constructor

Abstract classes

A motivating example

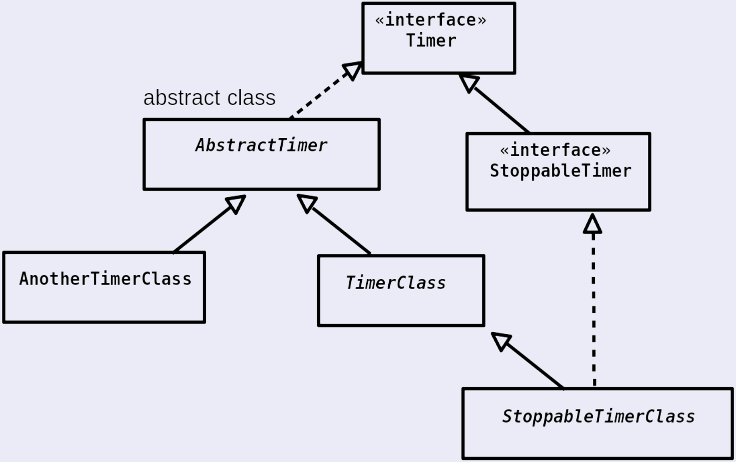
```
public class TimerClass implements Timer {
    ...
    private static void checkMinutes(int minutes) {
        if (minutes < 0 || minutes > 60)
            throw new IllegalArgumentException();
    }
    public boolean isRunning() { return this.getTime() > 0; }
    ...
}

public class AnotherTimerClass implements Timer {
    ...
    private static void checkMinutes(int minutes) {...} // same code as above
    public boolean isRunning() {...}                    // same code as above
    ...
}
```

- Observation: code for `checkMinutes` and `isRunning` is duplicated
- Question: can code be refactored (=organized in a better way) to be shared?

Abstract classes

Code is shared with an abstract class



Abstract classes

Example

```java
public abstract class AbstractTimer implements Timer {

    protected AbstractTimer(){}                        // for subclasses use

    protected static void checkMinutes(int minutes) { // for subclasses use
        if (minutes < 0 || minutes > 60)
            throw new IllegalArgumentException();
    }
    public boolean isRunning() {
        return this.getTime() > 0;
    }

    abstract public int getTime();          // optional declaration

    abstract public void tick();            // optional declaration

    abstract public int reset(int minutes); // optional declaration
}
```

Abstract classes

Example

```java
// checkMinutes(int) and  isRunning() are no longer defined here

public class TimerClass extends AbstractTimer {
    public TimerClass() {
    }
    public TimerClass(int minutes) {
        this.time = TimerClass.checkMinutes(minutes) * 60;
    }
    public TimerClass(Timer other) {
        this.time = other.getTime();
    }
    public int getTime() {
        return this.time;
    }
    public void tick() {
        if (this.isRunning())
            this.time--;
    }
    public int reset(int minutes) {
        int prevTime = this.getTime();
        this.time = TimerClass.checkMinutes(minutes) * 60;
        return prevTime;
    }
}
```

Abstract classes

In a nutshell

- abstract classes are partial implementations, typically of interfaces
- they can contain:
  - both abstract and non abstract object methods
  - all other elements of a non abstract class
- they cannot be used to create objects

Abstract classes

Some Java details

- declared with the class modifier **abstract**

  ```
  public abstract class AbstractTimer ...
  ```

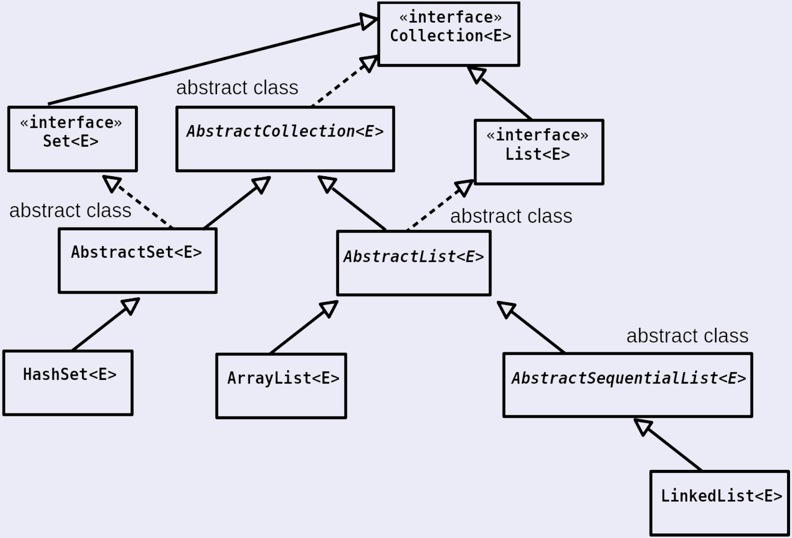
- not allowed to create objects of abstract classes

  ```
  new AbstractTimer(); // compile-time error!
  ```

  anyway constructors, typically protected, may be useful for sublasses
- a class must be abstract if it declares or inherits abstract methods
- a class can be abstract without any abstract method
  this is useful when we do not want to create objects from it

Abstract classes

A typical hierarchy with classes, abstract classes and interfaces



More details on overriding

Rule 1

private object methods cannot be redefined, because they are not visible

Example

```java
public class Parent {
    private String className() {
        return "Parent";
    }
    public String display() {
        return this.className();
    }
}
public class Heir extends Parent {
    public String className() { // does not redefine className() of Parent!
        return "Heir";
    }
    public static void main(String[] args){
        Heir h=new Heir();
        assert h.className().equals("Heir");
        assert h.display().equals("Parent");
    }
}
```

More details on overriding

Rule 2

visibility of the redefined method can only be enlarged

Example

```
public class Parent {
    protected String className() {
        return "Parent";
    }
    public String display() {
        return this.className();
    }
}
public class Heir extends Parent {
    @Override // redefines className() of Parent and extends its visibility
    public String className() {
        return "Heir";
    }
    public static void main(String[] args){
        Heir h=new Heir();
        assert h.className().equals("Heir");
        assert h.display().equals("Heir");
    }
}
```

Multiple versus single class inheritance

Definitions

- single class inheritance: a class can extend only one class
- multiple class inheritance: a class can extend more classes

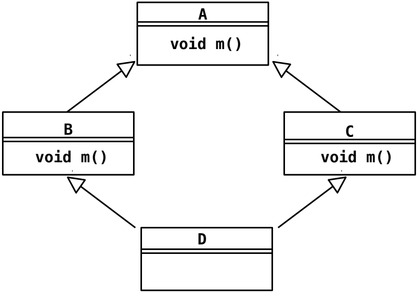
Pros and cons of multiple inheritance

- pros: more flexible
- cons: more complex semantics and implementation

Multiple inheritance and mainstream languages

- C++, Python support multiple class inheritance
- Java, C# and Kotlin support multiple inheritance only for interfaces

The diamond problem



Example

```
D d = new D();
d.m(); // ambiguous situation: which method should be dispatched?
```