Inheritance

A motivating example

Goal: new class `StoppableTimerClass` implementing `StoppableTimer`

```
public interface StoppableTimer extends Timer {
    boolean stopped();
    void stop();
    void restart();
}
```

Intuition:

- `StoppableTimer` has all methods of `Timer` plus `boolean stopped()`, `void stop()`, `void restart()`
- objects of class `StoppableTimerClass` provide
  - all features of `Timer`
  - and the additional features of `StoppableTimer`

Inheritance

A motivating example

Specification of the additional features:
objects of class `StoppableTimerClass` are stoppable timers:

- they can be stopped and restarted
- they count down as conventional timers, when they are not stopped
- initially they are not stopped

Aims:

- reuse code and do not modify preexisting code
  example: field `time` and method `getTime()` same as in `TimerClass`
- allow type compatibility between objects: a `TimerClass` object can be safely replaced with a `StoppableTimerClass` object

Solution: use inheritance

`StoppableTimerClass` extends `TimerClass`

Inheritance

Demo 1

```
public class StoppableTimerClass extends TimerClass implements StoppableTimer {

    private boolean stopped;

    public boolean stopped() { return this.stopped; }
    public void stop() { this.stopped = true; }
    public void restart() { this.stopped = false; }
    @Override public boolean isRunning() {     // redefined
        return super.isRunning() && !this.stopped();
    }
    @Override public int reset(int minutes) {  // redefined
        this.restart();
        return super.reset(minutes);
    }
    public static void main(String[] args) {
        StoppableTimerClass st = new StoppableTimerClass();
        assert st.isRunning() && st.getTime()==60;
        st.stop();
        assert !st.isRunning() && st.getTime()==60;
        st.tick();
        assert !st.isRunning() && st.getTime()==60;
        st.restart();
        st.tick();
        assert st.isRunning() && st.getTime()==59;
    }
}
```

Inheritance

Remark

- objects of `StoppableTimerClass` have field `time`
- but `time` is not visible in `StoppableTimerClass` because it is private

Demo 2

```
public static void main(String[] args) {
    TimerClass timerObj = new TimerClass();
    StoppableTimerClass stoppableTimerObj = new StoppableTimerClass();

    // implicit widening reference conversion for all assignments below
    TimerClass timerObj2 = stoppableTimerObj;
    StoppableTimer stoppableTimer = stoppableTimerObj;
    Timer timer = stoppableTimer;
}
```

Valid subtyping relations

- `StoppableTimerClass` $\leq$ `TimerClass` $\leq$ `Timer`
- `StoppableTimerClass` $\leq$ `StoppableTimer` $\leq$ `Timer`

Access modifier `protected`

Motivation

A new modifier expressly designed to grant access to subclasses

Example

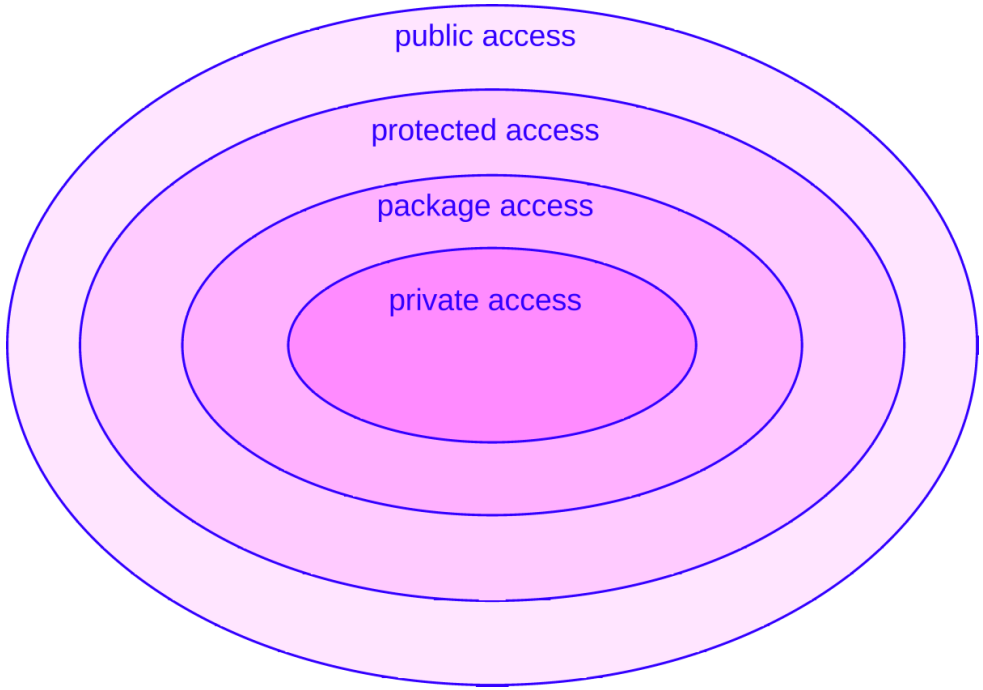
```
public class TimerClass implements Timer {
  protected int time = 60;
  ...
}

public class StoppableTimerClass extends TimerClass implements StoppableTimer {
  ... // now time is visible in StoppableTimerClass
}
```

Access modifier `protected`

Simplified rules for `protected`

Protected components can be accessed:

- in any class defined in the same package
- in any subclass, defined in any package



Inheritance

Basic terminology

- `StoppableTimerClass` extends (or inherits from) `TimerClass`
- `StoppableTimerClass` is a direct subclass of `TimerClass`
- `TimerClass` is the direct superclass of `StoppableTimerClass`
- relations subclass and superclass are the transitive closure of direct subclass and direct superclass, respectively `TimerClass`

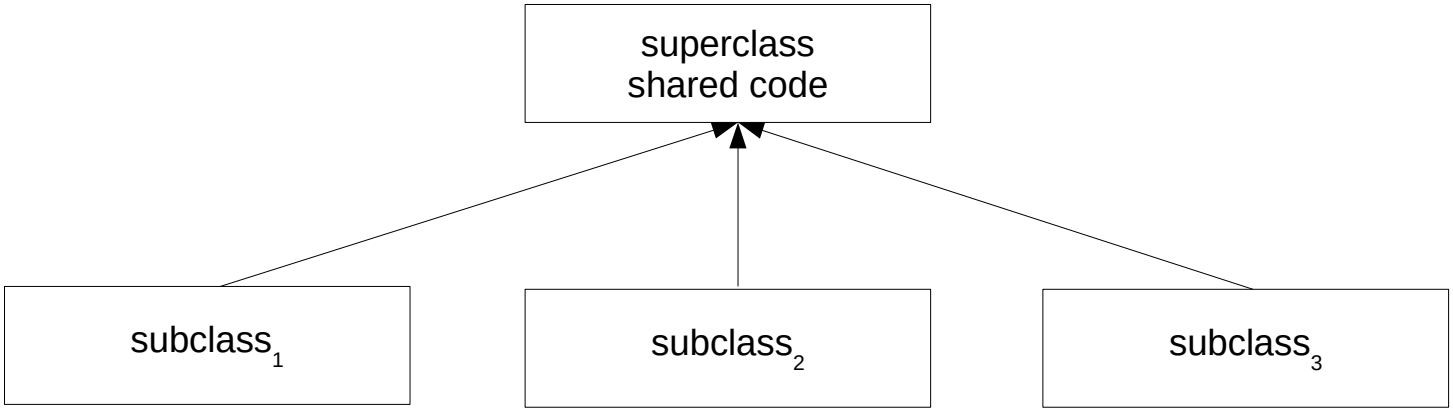
Remark

user-defined classes must extend a single class (default is `Object`)

Inheritance and subtyping

- in Java and many other languages: inheritance implies subtyping
- `StoppableTimerClass` extends `TimerClass` implies `StoppableTimerClass` is a subtype of `TimerClass`
- `StoppableTimer` extends `Timer` implies `StoppableTimer` is a subtype of `Timer`

Inheritance



Main motivations

- reuse: common code can be shared in superclasses
- extensibility: new features can be added in subclasses without changing/re-compiling superclasses
- subtyping:
  - backward compatibility
  - polymorphism

Inheritance

Inheritance and overriding

- overriding: redefinition of inherited object methods in subclasses

Redefined methods in `StoppableTimerClass`

```
public class StoppableTimerClass extends TimerClass {
    private boolean stopped;
    ...
    @Override // @Override optional but strongly recommended
    public boolean isRunning() {
       return super.isRunning() && !this.stopped();
    }
    @Override // @Override optional but strongly recommended
    public int reset(int minutes) {
        this.restart();
        return super.reset(minutes);
    }
}
```

Method overriding

Keyword **super**

- useful to reuse an inherited method in its redefinition

Examples

```
super.isRunning() // calls 'isRunning' of 'TimerClass' on target object 'this'

super.reset(minutes) // calls 'reset' of 'TimerClass' on target object 'this'
```

Method overriding

Simplified rules for correct method overriding

```
class P {
    public T m(T1 x1, ..., Tn xn){...}
}
class H extends P {
    @Override
    public T' m(T1 x1, ..., Tn xn){...}
}
```

Requirements on the new method in the subclass H:

- same name of the redefined method
- same numbers and types of parameters of the redefined method
- the return type can be changed, with the following rules:
  - either T, T' reference types, and $T' \leq T$
  - or T, T' primitive types or **`void`** and $T' = T$
- in short: the redefined method can differ only in the return reference type

Method overriding

Rule for annotation `@Override`

- declares that an object method redefines an inherited method
- the compiler checks that overriding is correct
- `@Override` is optional, but strongly recommended

Example

```
class Object { public boolean equals(Object o){...} ... }
class C1 extends Object {
    @Override
    public boolean equals(C1 o){...} // static error: invalid overriding!
}
class C2 extends Object {
    public boolean equals(C2 o){...} // no error, unintentional overloading?
}
```

Method overloading

Definition

- as constructors, methods can be overloaded
- a method with name *m* is overloaded in *C*, if *C* has more definitions of *m*

Example

```
class Object {
  public boolean equals(Object o){...}
  ...
}

class C extends Object {
  public boolean equals(C o){...}
  ...
}
```

Class `C` has two definitions of method `equals`:

- `equals(Object)` (inherited from `Object`)
- `equals(C)` (defined in `C`)

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

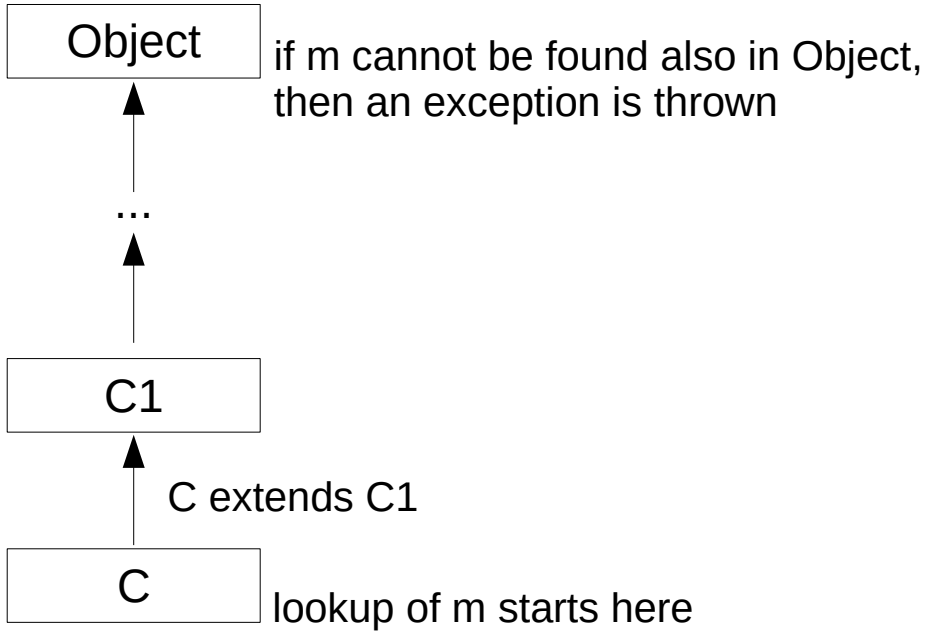
```
public class TimerClass implements Timer {
    ...
    public boolean isRunning() {
        return this.getTime() > 0;
    }
    // which method will be called with 'this.isRunning()'?
    public void tick() {
        if (this.isRunning()) // more general than 'this.getTime() > 0'
            this.time--;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

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



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 3

```
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

Example

```
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`**

Classes, interfaces and inheritance

Rules in Java

- **`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

Example

```
public interface StoppableTimer extends Timer {
    boolean stopped();
    void stop();
    void restart();
}
```

Remark

`StoppableTimer` is a subtype of `Timer`: $\texttt{StoppableTimer} \leq \texttt{Timer}$