Inheritance and Polymorphism

Mikael Svahnberg[*]

2023-09-05

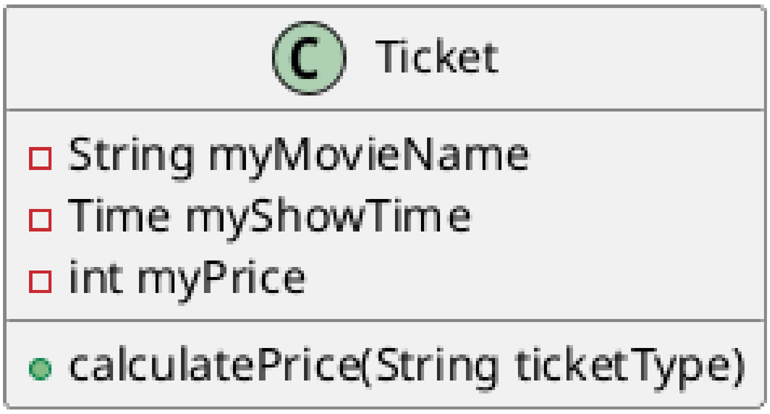
Contents

[*] Mikael.Svahnberg@bth.se

1 Introduction

- Barnes & Kölling Chapter 8, Designing Classes
- Design Principle: *Low Coupling*
- Design Principle: *Encapsulation*
- Inheritance
- Polymorphism
- Multiple Inheritance
  - `extends` vs `implements`
  - abstract classes

2 Objects with Similar but Differing Behaviour



- The `calculatePrice()` sets `myPrice` of each object
- All objects still behave in the same way, and have the same attributes
- No need for different types of tickets. Yet.
- What if we start to have different behaviour? Different attributes?
- We could write massive `if-then-else-if` statements, as below
  - This adds clutter for every behaviour that differs.
  - Breaks the principle of *High Cohesion*
    - Ticket now has the responsibility of being all types of tickets.

```
enum TicketType {Normal, Student, Pensioner, Child};

int Ticket::findBestSeat(TicketType theType) {
  switch(theType) {
```

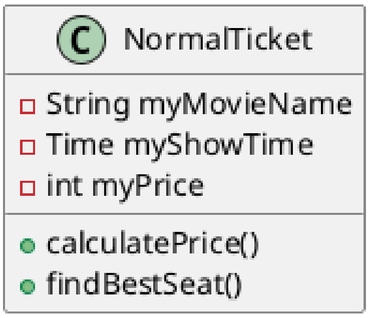
```
  case Student: return this->findBestSeatStudent(); break;
  case Pensioner: return this->findBestSeatPensioner(); break;
  case Child: return this->findBestSeatChild(); break;
  case Normal:
  default:
    return this->findBestSeatNormal();
    break;
  }
}
```

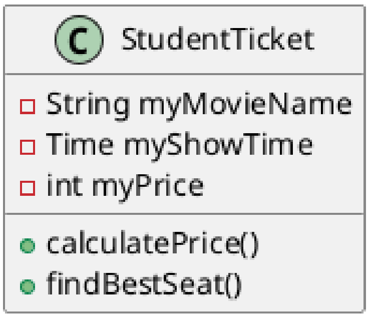
3 A brief excursion: Switch-case

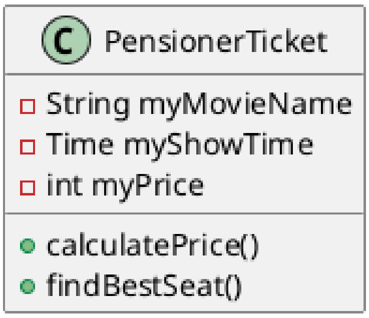
- Introducing another way to write an `if-then-else-if...` -chain
- `switch (some variable) {`
- `case X:` One case statement for each case. Only concrete cases, no ranges or boolean expressions.
  - ... code as normal
  - `break;` MUST have this, otherwise it will just keep executing
- `default:` SHOULD have a default statement, executed if no other case matches.

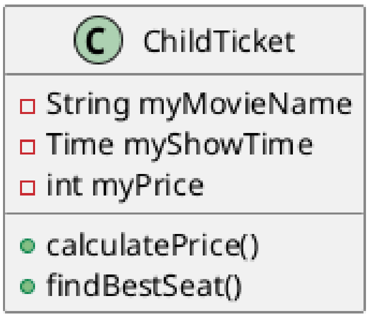
```
int someVar=0;
switch(someVar) {
case 0:
  System.out.println("Zero");
  break;
case 1:
  System.out.println("One");
  break;
default:
  System.out.println("I can’t count that far!");
  break;
}
```

4 Differing Behaviour → Different Classes









- Each class knows how to `calculatePrice()` and `findBestSeat()` for that type of object.
- Design Principle *High Cohesion* and Design Principle *Encapsulation*
  - Each class *encapsulates* the unique behaviour
  - Each class has a single well defined responsibility
  - Each type of Ticket class is not aware of any other types of Tickets.
- How do we manage a collection of tickets?
  - One collection for each type of ticket?
  - One collection with all types of tickets? How would we do that in Java?

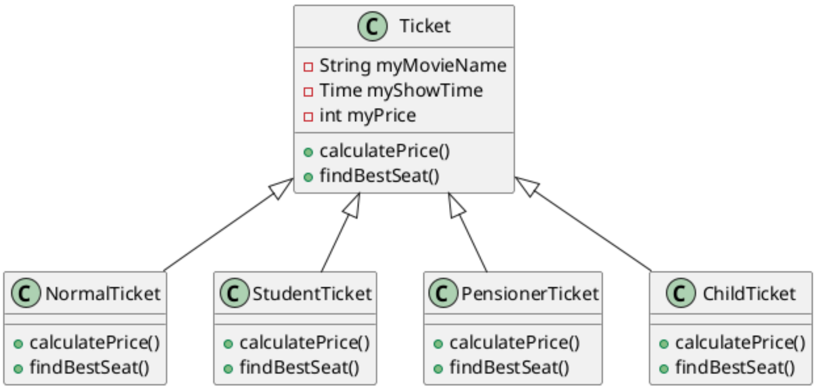
5 Collections of Differing Objects

- Perhaps a collection of collections?

```
import java.util.ArayList;

ArrayList<> myNormalTickets = new ArrayList<NormalTicket>();
ArrayList<> myStudentTickets = new ArrayList<StudentTicket>();
```

```
ArrayList<> myPensionersTickets = new ArrayList<PensionerTicket>();
ArrayList<> myChildrenTickets = new ArrayList<ChildTicket>();

ArrayList<> myTickets = new ArrayList<ArrayList<>>(); // Not entirely sure about the synta
myTickets.add(myNormalTickets);
myTickets.add(myStudentTickets);
myTickets.add(myPensionersTickets);
myTickets.add(myChildrenlTickets);

// Does your head hurt yet? Mine does.
// This is obviously a bad idea.
```

6 Trust the Compiler: Inheritance



```
ArrayList<Ticket> myTickets = new ArrayList<>();
myTickets.add(new NormalTicket());
myTickets.add(new StudentTicket());
myTickets.add(new PensionerTicket());
myTickets.add(new ChildTicket());

for(Ticket t : myTickets) { t.calculatePrice(); }
for(Ticket t : myTickets) { t.findBestSeat(); }
```

- The compiler knows the type of each object.
- The compiler selects the right method implementation for `calculatePrice()` and `findBestSeat()`.
- *We* have told the compiler that objects of the classes `NormalTicket`, `StudentTicket`, `PensionerTicket`, and `ChildTicket` can be treated the same as the class `Ticket`, except when they implement their own unique behaviour for some method or attribute.
- For this to work, the method must *also* be declared in the "super class".
- *Polymorphism* is an *extremely important* feature of object oriented programming

  - Poly Morphos ≈ Many Forms
  - Same method signature, many different implementations.

7 Inheritance in Java: Extends

```
public class StudentTicket extends Ticket {
  public StudentTicket() {
    super("no student movie", "no time"); // Calling the constructor of the super class
  }

  public int findBestSeat() {
    // Perhaps get a starting position from the super class
    // Using the reference "super" to get to the original findBestSeat() method
    // Otherwise I would just recursively call myself...
    int seat = super.findBestSeat();

    // Then do some Student-specific magic to get the
    // best seat for them...
    mySeat = 10; // Do not need to use "super" here. Sub-classes inherit all methods and a
    return this.mySeat; // But I can use the this reference if I need to clarify
  }
}
```

7.1 Usage

```
import java.util.ArrayList;
public class Start {
  public static void main(String [] args) {
    ArrayList<Ticket> tick = new ArrayList<>();
    tick.add(new Ticket());
    tick.add(new StudentTicket());

    for(Ticket t : tick) { System.out.println(t.findBestSeat()); }
    // Note that at this point I no longer know or care
    // about which sub-type each object has.
    // I only need to treat them as Tickets, and the compiler
    // will take care of the rest.
  }
}
```

8 Inheritance in C++

- `class subClass : public superClass`
- Polymorphic methods must be declared `virtual`
- The sub-class method may be specified with `override`
  - This double-checks that the original method was indeed declared `virtual`

  - override is not required but good practice.

- Polymorphism only works with pointers to objects!

```
#ifndef TICKET_HH
#define TICKET_HH
#include <string>

class Ticket {
public:
  Ticket(void) : Ticket("-- not specified -- ", "-- not specified --") {}
  Ticket(std::string theName, std::string theTime) : movieName(theName),showTime(theTime)
  virtual int findBestSeat() { mySeat=0; return mySeat; }
  std::string toString(void) {
    return "Ticket for " + movieName + " at " + showTime + " sitting in seat " + std::to_s
  }
protected:
  int mySeat;
private:
  std::string movieName;
  std::string showTime;
};

#endif
```

```
#ifndef STUDENTTICKET_HH
#define STUDENTTICKET_HH

#include "ticket.hh"

class StudentTicket : public Ticket {
public:
  StudentTicket(void) : Ticket("no student movie", "no time") {}

  int findBestSeat(void) override {
    int seat = Ticket::findBestSeat();
    mySeat = 10;
    return this->mySeat;
  }
};

#endif
```

8.1 Usage

```
#include <vector>
#include <iostream>
using namespace std;
#include "ticket.hh"
#include "studentticket.hh"
```

```
int main(void) {
  vector<Ticket*> tick;
  tick.push_back(new Ticket());
  tick.push_back(new StudentTicket());

  for (auto t : tick) { cout << t->findBestSeat() << endl; }
  for (auto t : tick) { cout << t->toString() << endl; }
}
```

9 Public, Protected, Private in a class

- Methods and Attributes in a class can be declared `public`, `private`, or `protected`
- `public` is accessible everywhere
- `private` and `protected` are only accessible from inside the class.
- `private` is not accessible in sub-classes.
- `protected` can be accessed in sub-classes.

Design Principle: *Encapsulation*

- Keep the internal design of a class hidden
- Only expose what should be used by others

10 Public, Protected, Private inheritance

C++ can inherit in three ways:

- `public` → everything remains the same.
  - `public` → `public`
  - `protected` → accessible by sub-classes
  - `private` → not accessible by sub-classes.
  - This is the most common case of inheritance
  - Java only has public inheritance
- `protected` → changes accessibility of the base class
  - `public` → `protected`
  - `protected` → `protected`
  - `private` → `private`
- `private` → changes accessibility of the base class
  - `public` → `private`
  - `protected` → `private`
  - `private` → `private`

11 Why protected or private inheritance?

- I may want to *use* a base-class, but not make it part of my public interface
  - → `protected` inheritance
  - Inheritance for code re-use
  - Keep public interface clean
- I may want to use a base-class *and* not make it available to any sub-classes
  - → `private` inheritance
  - Inheritance for code re-use
  - Keep internal interface clean, perhaps avoid issues with multiple inheritance.
  - I want to inherit the *type*, but do not need the behaviour.

12 Multiple Inheritance in C++

```
class Warrior : public Wizard, public Swordsman {
public:
  Warrior(void);
  virtual void fight(void) override;
  // ...
protected:
private:
};
```

- Warrior is now both a Wizard and a Swordsman (with all their methods and attributes)
- But what about the `fight()` method?
  - Should we fight like a wizard or a swordsman?
  - the Warrior class *has* to make that decision and reimplement the `fight()` method.

```
void Warrior::fight(void) {
  if (is_holding("wand")) {
    Wizard::fight();
  } else if (is_holding("sword")) {
    Swordsman::fight();
  } else {
    bravely_run_away();
  }
}
```

13 Multiple Inheritance in Java

- Cannot inherit behaviour
- Can only inherit declaration; an *interface*

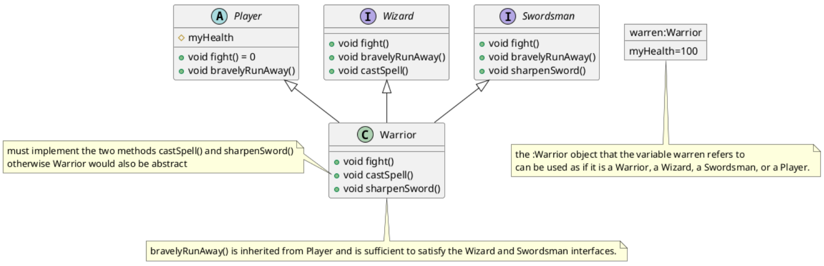
```
public interface Wizard {
  public void fight(); // Only the declaration here.
}

public class Warrior implements Wizard, Swordsman {
  public void fight() {
    // The only implementation of fight() is here.
    // even if it has been specified in many places.
  }
}
```

14 Abstract Classes

- an *abstract class* is missing at least one method implementation.
- You can not instantiate objects from an abstract class, you must first sub-class it.
- In Java you use an abstract class when
  - You want to encapsulate some common functionality
  - You want to partially define an API, where sub-classes fill in the details
- In C++, this is the closest to an `interface` that you will get.

```
abstract class Player {
  public abstract void fight();

  public void bravely_run_away() {  // can have some concrete methods and others abstract.
    System.out.println("You successfylly attack backwards");
  };
}

// ...

class Wizard extends Player {
  public void fight() {
    // Here is the implementation for fight()
  }
}

class Player {
public:
  virtual void fight(void) = 0; // abstract
  virtual void bravely_run_away(void); // will be implemented in the Player class
protected:
  int myHealth; // Unlike a java interface, we can introduce attributes in an abstract cla
private:
};

class Wizard : public Player, public Writable {
public:
  virtual void fight(void); // Re-declared here to tell the compiler that we intend to imp
};

void Wizard::fight(void) {
  // implementation for fight()
}
```

15 Objects and Classes and Interfaces (oh, my!)



must implement the two methods castSpell() and sharpenSword() otherwise Warrior would also be abstract

the :Warrior object that the variable warren refers to can be used as if it is a Warrior, a Wizard, a Swordsman, or a Player.

bravelyRunAway() is inherited from Player and is sufficient to satisfy the Wizard and Swordsman interfaces.

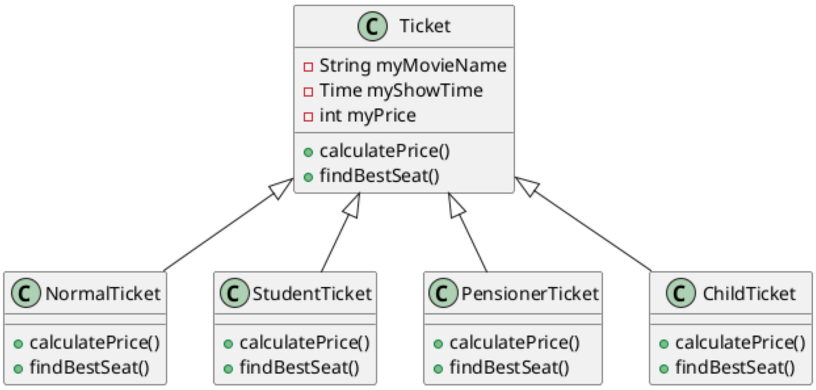
```
Warrior warren = new Warrior();
warren.fight();
warren.castSpell();
warren.sharpenSword();
warren.bravelyRunAway();

Wizard willy = new Warrior();
willy.castSpell();
// willy.sharpenSword(); // NOT ok, the variable willy refers to a Wizard.

// Player pete = new Player(); // NOT ok, Player is abstract
Player aliasToWarren = (Player) warren;
// aliasToWarren.castSpell(); // NOT ok, aliasToWarren refers to a Player, which does not 
aliasToWarren.fight();
System.out.println(aliasToWarren instanceof Wizard); // This is still true, so we can re-c
  // We should be careful with re-casting objects
  // It is better to hold a reference to a sufficiently generic object
  // and "lift" behaviour upwards in the inheritance hierarchy
  // so that the same method name can be used but with different implementations
```

- Design Principle: *Encapsulation*
  - By using a variable of the right type, only the interface we currently need is accessible to us

  - At that point, we are maybe not even aware that the same object may have different roles
  - E.g. `ArrayList<Player>` may be all we want and need.

16 Inheritance Hierarchies

- Multiple inheritance is useful to describe different roles an object may have
- More commonly we want to describe many different types of objects that share the same interface



17 Summary

- Design Principle `High Cohesion`
- Design Principle `Encapsulation`
- Design Principle `Low Coupling`
- `switch-case`
- Inheritance
  - public, protected private inheritance
  - abstract class
  - interface
  - casting an object to a different type
- Polymorphism
  - `virtual`
- Multiple inheritance vs inheritance hierarchies

18 Next Lecture: Design Patterns: Strategy

- (Freeman & Robson, Chapter “intro”)
- Freeman & Robson, Chapter 1: Welcome to Design Patterns
- Design Principle: *Encapsulation*
- Design Principle: *High Cohesion*
- Design Principle: *Low Coupling*
- Design Pattern: *Strategy*