List of Slides

1 Title
2 **Chapter 16:** Inheritance
3 Chapter aims
5 **Section 2:** The Notional Lottery game
6 Aim
7 The Notional Lottery game
8 The Notional Lottery game
9 The Notional Lottery game
10 **Section 3:** The `Person` class
11 Aim
12 The `Person` class
13 The `Person` class
14 Inheritance
17 The `Person` class
18 The `Person` class
19 The `Person` class

20 The `Person` class
21 The `Person` class
22 The `Person` class
23 The `Person` class
24 The `Person` class
25 The `TestPerson` class
26 Trying it
27 Coursework: Stock control system
28 **Section 4:** The `AudienceMember` class
29 Aim
30 Inheritance: a subclass extends its superclass
33 The `AudienceMember` class
34 Inheritance: invoking the superclass constructor
37 The `AudienceMember` class
38 The `AudienceMember` class
39 Inheritance: overriding a method
42 The `AudienceMember` class
43 The `AudienceMember` class

44 Design: UML
45 Design: UML: class diagram
46 The `AudienceMember` class
47 The `AudienceMember` class
48 The full `AudienceMember` code
49 The `TestAudienceMember` class
50 The `TestAudienceMember` class
51 Trying it
52 Coursework: Your first stock item!
53 **Section 5:** The `Punter` class
54 Aim
55 The `Punter` class
57 The `Punter` class
58 The `Punter` class
59 The `TestPunter` class
60 Trying it
61 Coursework: Your catalogue
62 **Section 6:** The `Person` abstract class

63 Aim
64 The `Person` abstract class
65 Inheritance: abstract class
67 The `Person` class
68 The `Person` class
70 Inheritance: abstract method
75 The `Person` class
76 The `Person` class
77 The `Person` class
78 The `AudienceMember` and `Punter` classes
79 Trying it
80 Coursework: An abstract stock item
81 **Section 7:** The remaining simple subclasses of `Person`
82 Aim
83 The remaining simple subclasses of `Person`
84 The `Director` class
85 The `Director` class
86 The `Psychic` class

88 The `TVHost` class
90 Latest inheritance hierarchy
91 The `TestPersonSubclasses` class
92 The `TestPersonSubclasses` class
93 The `TestPersonSubclasses` class
94 Inheritance: polymorphism
95 The `TestPersonSubclasses` class
96 The `TestPersonSubclasses` class
97 Inheritance: polymorphism: dynamic method binding
99 Inheritance: final methods and classes
100 The `TestPersonSubclasses` class
101 Trying it
102 Coursework: More stock items
103 **Section 8:** The `MoodyPerson` classes
104 Aim
105 The `MoodyPerson` classes
106 The `MoodyPerson` classes
107 The `MoodyPerson` class

108 Inheritance: adding more object state
109 The `MoodyPerson` class
110 The `MoodyPerson` class
111 Method: constructor methods: more than one: using this
114 The `MoodyPerson` class
115 Inheritance: adding more instance methods
116 The `MoodyPerson` class
117 The `Teenager` class
118 The `Teenager` class
119 The `Teenager` class
120 The `Teenager` class
121 The `Teenager` class
122 The `TestPersonSubclasses` class
123 The `TestPersonSubclasses` class
124 Inheritance: testing for an instance of a class
125 Inheritance: casting to a subclass
128 The `TestPersonSubclasses` class
130 The `TestPersonSubclasses` class

131 Trying it
132 Coursework: Lots of different mouse mats!
133 **Section 9:** The `Ball` class
134 Aim
135 The `Ball` class
136 GUI API: `Color`
138 The `Ball` class
142 **Section 10:** The `BallContainer` classes
143 Aim
144 The `BallContainer` classes
145 The `BallContainer` classes
146 The `BallContainer` classes
147 The `BallContainer` class
148 The `BallContainer` class
149 The `BallContainer` class
151 The `BallContainer` class
154 The `Machine` class
155 Array: partially filled array: deleting an element

156 The `Machine` class
158 The `Rack` class
159 The `Rack` class
160 The `Rack` class
161 **Section 11:** The `Game` class
162 Aim
163 The `Game` class
164 Inheritance: is a versus has a
165 The `Game` class
166 The `Game` class
167 The `Game` class
168 The `Game` class
169 The `Game` class
170 The `Game` class
171 Coursework: Shopping baskets
172 **Section 12:** The `Worker` classes
173 Aim
174 The `Worker` classes

175 The `Worker` classes
176 The `Worker` classes
177 The `Worker` class
178 The `Worker` class
179 The `Worker` class
180 The `Worker` class
182 The `TraineeWorker` class
183 The `TraineeWorker` class
184 The `TraineeWorker` class
185 Inheritance: using an overridden method
187 The `TraineeWorker` class
188 The `TraineeWorker` class
189 The `TraineeWorker` class
190 The `TestWorkers` class
192 Trying it
193 Coursework: Loads of disc space
194 **Section 13:** The `CleverPunter` class
195 Aim

196 The `CleverPunter` class
197 The `CleverPunter` class
198 The `CleverPunter` class
199 The `CleverPunter` class
200 The `CleverPunter` class
203 The `CleverPunter` class
204 The `CleverPunter` class
205 The `TestCleverPunter` class
207 Trying it
208 Trying it
209 Trying it
210 Coursework: Making it more realistic
211 **Section 14:** The GUI classes
212 Aim
213 The GUI classes
214 The GUI classes
215 MoodyPerson.java
216 The GUI classes

217 **Section 15:** The `Object` class and constructor chaining
218 Aim
219 Standard API: `Object`
221 Inheritance: invoking the superclass constructor: implicitly
222 The `Object` class and constructor chaining
223 Person.java
224 The `Object` class and constructor chaining
225 Person.java-WITH-SUPER
226 Inheritance: constructor chaining
227 The `Object` class and constructor chaining
228 TraineeWorker.java
229 Worker.java
230 MoodyPerson.java
231 Person.java
232 The `Object` class and constructor chaining
233 Method: constructor methods: default
236 Trying it
237 Coursework: Exploring constructor chaining

238 **Section 16:** Overloaded methods versus override
239 Aim
240 Overloaded methods versus override
241 Does an **int** match a **double**?
243 Does an **int** match a **double**?
244 Does an **int** match a **double**?
245 Standard API: System: out.println(): with any argument
246 Standard API: Object: toString()
247 System.out.println() and inheritance
248 System.out.println() and inheritance
249 System.out.println() and inheritance
250 Accidental overload
251 Accidental overload
253 Accidental overload
254 Overloaded methods versus override
255 Overloaded methods versus override
256 Overloaded methods versus override
257 Overloaded methods versus override

258 Inheritance: overriding a method: @Override annotation
259 Overloaded methods versus override
260 SafeChiefInspector.java-WITH-HIDE
261 Overloaded methods versus override
262 Coursework: Using the `@Override` annotation
263 Concepts covered in this chapter

Title

Java Just in Time

John Latham

January 28, 2019

Chapter 16

Inheritance

Chapter aims

- A core principle of **object oriented programming**: a **class** might **inherit** properties from another.
- Already met in context of **graphical user interface**s
  - e.g. `HelloWorld` inherited properties from `JFrame`
  - `HelloWorld` was a particular kind of `JFrame`.
- Also, more implicitly with **exception**s
  - different kinds of, say, `RuntimeException`.
- This chapter properly introduces **inheritance**.

Chapter aims

- Unlike previous chapters this has single program example.
  - Finished program has more than 3000 lines of code
  - divided into nearly 40 **class**es.
- Secondary aim: show how can develop and test larger programs incrementally.
  - Development divided into phases
    - ∗ subdivided into sections – implementation and testing of one or more classes.
- We do not explore whole program
  - just parts acting as vehicle for covering inheritance.

Section 2

The Notional Lottery game

Aim

AIM: To introduce the example program used throughout this chapter.

The Notional Lottery game

- A game for children - the Notional Lottery
  - teach young players how unlikely they are to win!
- Traditionally start development of a program by identifying detailed requirements
  - but to keep coverage interesting present just overview here
  - give more detail as we proceed.

The Notional Lottery game

- Game has models of
  - people
  - lottery games.
    - ∗ comprising balls in a machine
    - ∗ ejected into a landing rack.
- End user (child) chooses people and sizes of lottery games
  - some of the people play the lottery.

The Notional Lottery game

- Two phases of development
  - first underlying model of program, i.e. people, games
  - then **graphical user interface**.
- Chapter covers first phase
  - says just a little about second.

Section 3

The Person class

Aim

AIM: To introduce the ideas of **superclass**, **subclass**, **inheritance**, and **is a** relationships.

The Person class

- Several kinds of person
  - e.g. audience members, TV hosts, psychics, etc..
- Child can make people speak
  - different kinds say different things
    - ∗ e.g. audience members always say “Oooooh!”
    - ∗ e.g. TV hosts always say “Welcome suckers!”
    - ∗ e.g. psychics always say “I can see someone very happy!”.
- Some kinds always smile, some always frown, some can change mood.

The `Person` class

- Each kind of person modelled by separate **class**
  - **instance**s made at **run time**.
- But also have some properties in common
  - so have general class called `Person` for common properties
  - more specific kinds inherit these via **inheritance**.

Inheritance

- A **class** is used to model category (classification) of **objects**
  - sometimes want to have sub-categories.
- E.g. program for simulating traffic movement:
  - `Vehicle` – containing properties common to all vehicles
  - sub-categories: bicycle, private car, taxi, bus, lorry etc..
  - each with specific properties:
    - bicycles can be chained to railings
    - lorries need access to unloading points at shops etc.
    - some vehicles carry passengers
    - some carry loads.

Inheritance

- Want to model sub-categories as separate classes
  - with specific properties as required.
- But also model idea they are all vehicles.
- In **object oriented programming**:
  - **superclass** models general category
  - **subclass** models a sub-category.
- E.g. `Vehicle` might be superclass of all vehicles
  - `Bicycle` could be sub-category for bicycles
  - `PrivateCar`, `Taxi`, `Bus`, `Lorry`, etc..

Inheritance

- The **is a** relationship: subclass / superclass
  - e.g. a bicycle **is a** vehicle
  - i.e. an **instance** of `Bicycle` is also an instance of `Vehicle`.
- Relationship known as **inheritance**
  - subclasses **inherit** general properties from superclass
  - add specific properties.

The Person class

- So, AudienceMember will be **subclass** of Person
  - an audience member **is a** person.
- Have other subclasses for other kinds of person.
- Here we develop Person class
  - also TestPerson program.

The Person class

```java
// Representation of a person involved somehow in the lottery.
public class Person
{
  // The name of the person.
  private final String personName;

  // The Person’s latest saying.
  private String latestSaying;


  // Constructor is given the person’s name.
  public Person(String requiredPersonName)
  {
    personName = requiredPersonName;
    latestSaying = "I am " + personName;
  } // Person
```

The Person class

- GUI will display name of person along with picture representing them.

```java
// Returns the Person's name.
public String getPersonName()
{
  return personName;
} // getPersonName
```

- GUI will display latest saying in speech bubble.

```java
// Returns the Person's latest saying.
public String getLatestSaying()
{
  return latestSaying;
} // getLatestSaying
```

The Person class

- GUI will show kind of person
  - each subclass will have different description.
    - e.g. AudienceMember will return "Audience Member".
- Have instance method in **superclass**
  - but redefine in each subclass.

```
// Returns the name of the type of Person.
public String getPersonType()
{
  return "Person";
} // getPersonType
```

The Person class

- GUI will draw person's face with smile or frown.
- Most kinds of person are always happy
  - so define `isHappy()` to **return** `true`
  - subclasses for kinds that are unhappy will redefine it.

```
// Returns whether or not the Person is happy.
public boolean isHappy()
{
  return true;
} // isHappy
```

The `Person` class

- `speak()` causes result from `getCurrentSaying()` to become latest saying
  - so GUI displays it via `getLatestSaying()`.
- Current saying depends on kind of person
  - define `getCurrentSaying()` here
  - redefine in each subclass.

The Person class

```
// Returns the Person’s current saying.
public String getCurrentSaying()
{
  return "I have nothing to say";
} // getCurrentSaying


// Causes the person to speak by updating their latest saying from
// their current saying.
public void speak()
{
  latestSaying = getCurrentSaying();
} // speak
```

The Person class

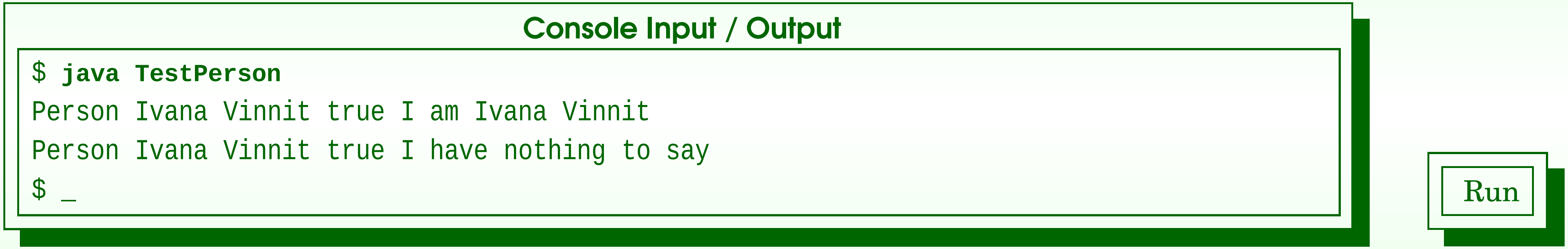
```
  // Mainly for testing.
  public String toString()
  {
    return getPersonType() + " " + getPersonName()
           + " " + isHappy() + " " + getLatestSaying();
  } // toString

} // class Person
```

The TestPerson class

- Test each section of incremental development as we go along.

```java
// Create a Person and make them speak.
public class TestPerson
{
  public static void main(String[] args)
  {
    Person person = new Person("Ivana Vinnit");
    System.out.println(person);
    person.speak();
    System.out.println(person);
  } // main

} // class TestPerson
```

Trying it



```
$ java TestPerson
Person Ivana Vinnit true I am Ivana Vinnit
Person Ivana Vinnit true I have nothing to say
$ _
```

Coursework: Stock control system

(Summary only)

Write a **class** that can be used to keep track of stock items, and test it.

Section 4

The `AudienceMember` class

Aim

AIM: To finish introducing **superclass**, **subclass** and **inheritance**, and briefly meet **UML**. Also, to introduce the principles of invoking the **constructor method** of the superclass, and having **instance method**s that **override** one from the superclass.

Inheritance: a subclass extends its superclass

- A **subclass** is **extension** of its **superclass**
  - may have more properties than superclass
    - as well as **inherit**ing properties of superclass.
- Heading of subclass states it **extend**s superclass.

Inheritance: a subclass extends its superclass

- E.g. a `Bicycle` **object** has properties of a `Vehicle`
  - but also can be chained to railings.

```
public class Bicycle extends Vehicle
{
  ...
  public void chainToRailings(Railings railings)
  {
    ...
  } // chainToRailings
  ...
} // class Bicycle
```

Inheritance: a subclass extends its superclass

- Used to represent **is a** relationships between model **class**es of programs.
- Also commonly used in **graphical user interface**s.
- E.g. `HelloWorld` is subclass of `javax.swing.JFrame`
  - `HelloWorld` is an extension of `JFrame`
  - **instance** of `HelloWorld` is a `JFrame` **object** too
    - but with extra properties.

```
import javax.swing.JFrame;
public class HelloWorld extends JFrame
{
  ... Code to add a JLabel with the text "Hello World!" in it.
} // class HelloWorld
```

The AudienceMember class

- Every **instance** of `AudienceMember` **is a** `Person` **object** too.

```java
// Representation of an audience member watching the lottery.
public class AudienceMember extends Person
{
```

Inheritance: invoking the superclass constructor

- Code of **constructor method** in **subclass** typically starts with **superclass constructor call**
  - **reserved word `super`** followed by **method argument**s.
  - Must be first **statement**
  - superclass must have constructor matching arguments.

Inheritance: invoking the superclass constructor

- E.g. vehicle is given position, direction and speed.

```
public class Vehicle
{
  ...
  public Vehicle(Position requiredPosition,
                 Direction requiredDirection, Speed requiredSpeed)
  {
    ... Code that does something with requiredPosition,
    ... requiredDirection and requiredSpeed.
  } // Vehicle
  ...
} // class Vehicle
```

Inheritance: invoking the superclass constructor

- Unlikely to make instances of `Vehicle` directly – want more specific kinds.
- Position, direction and speed passed to `Vehicle` constructor.

```
public class Bicycle extends Vehicle
{
  ...
  public Bicycle(Position position, Direction direction, Speed speed)
  {
    super(position, direction, speed);
    ... Code specific to making a Bicycle, if any, goes here.
  } // Bicycle
  ...
} // class Bicycle
```

The AudienceMember class

- So **super** here means constructor of `Person`.

```java
// Constructor is given the person’s name.
public AudienceMember(String name)
{
  super(name);
} // AudienceMember
```

- Name passed to `AudienceMember` constructor
  - is passed to `Person` constructor
  - which stores in **instance variable** `personName`.
  - Also `latestSaying` is initialized.

The `AudienceMember` class

- `Person` has lots of **instance methods inherit**ed by `AudienceMember`
  - `getPersonName()`, `getLatestSaying()`,
  - `getPersonType()`, etc..
- Definition of `getPersonType()` not suitable here
  - **return**s `"Person"` instead of `"Audience Member"`.

Inheritance: overriding a method

- A **subclass inherit**s **instance method**s of its **superclass**.
- Sometimes subclass needs to change definition of instance method
  - simply redefines it
  - subclass version **override**s inherited definition
  - must have same name and **type**s of **method parameter**s
    - otherwise is different method
  - must still be instance method
  - and have matching **return type**.

Inheritance: overriding a method

- E.g. most vehicles perform emergency stop in same way.

```
public class Vehicle
{
  ...
  public void emergencyStop()
  {
    ... General code for most vehicles.
  } // emergencyStop
  ...
} // class Vehicle
```

Inheritance: overriding a method

- But bicycles are different!

```
public class Bicycle extends Vehicle
{
  ...
  public void emergencyStop()
  {
    ... Specific code for bicycles.
  } // emergencyStop
  ...
} // class Bicycle
```

The AudienceMember class

Coffee time: Why can we not override a **class method**? Hint: instance methods are accessed via (a **reference** to) an object. How are class methods (usually) accessed?

```java
// Returns the name of the type of Person.
public String getPersonType()
{
  return "Audience Member";
} // getPersonType
```

Coffee time: What would happen if we accidentally mistyped the name of this instance method, as say, `getPersontype`? What would happen if instead we got the name right, but declared it here to be a **void method**?

The AudienceMember class

- Also override getCurrentSaying().

```
  // Returns the Person’s current saying.
  public String getCurrentSaying()
  {
    return "Oooooh!";
  } // getCurrentSaying

} // class AudienceMember
```

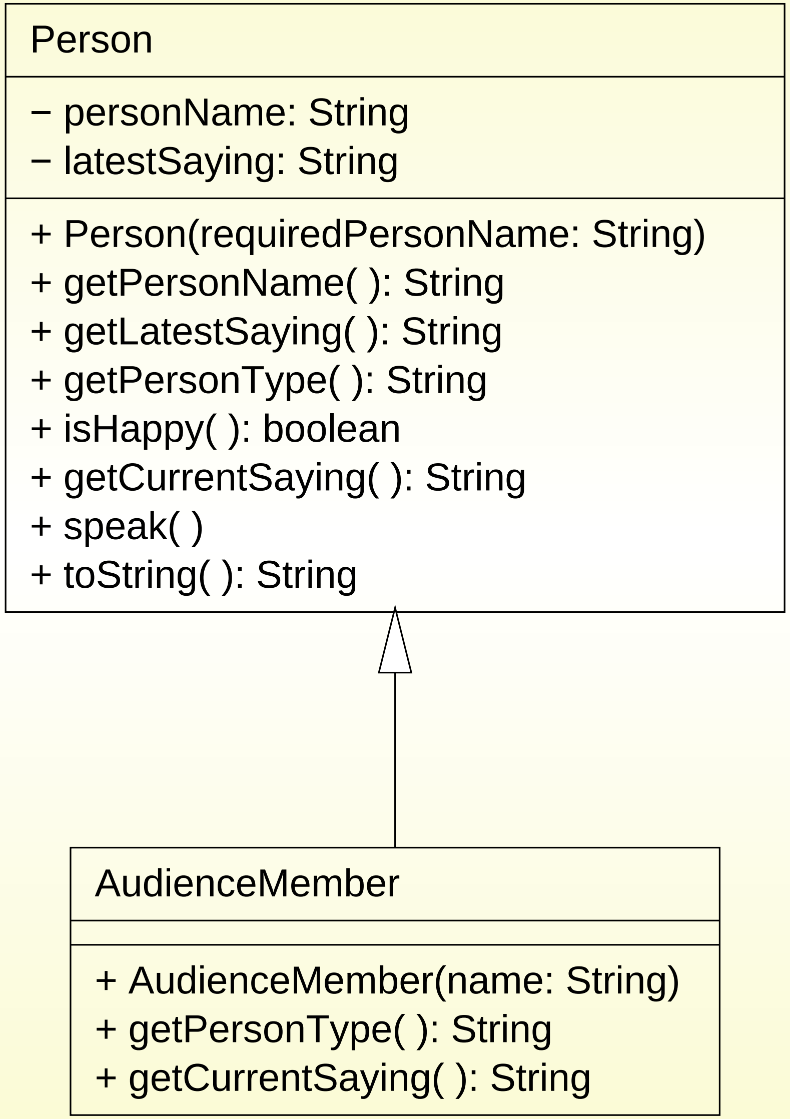
Design: UML

- **Unified Modelling Language** (**UML**)
  - collection of diagram types
  - can show relationships between entities
    - e.g. **object**s and **class**es.
- Used by many professional Java programmers for **design**s.

Design: UML: class diagram

- **UML class diagram** can be used to represent an **inheritance hierarchy**.
- Each **class** appears as box
  - with name
  - **variable**s
  - and **method**s
  - **private** items marked with -
  - **public** items with +.

The AudienceMember class



Person

− personName: String
− latestSaying: String

+ Person(requiredPersonName: String)
+ getPersonName( ): String
+ getLatestSaying( ): String
+ getPersonType( ): String
+ isHappy( ): boolean
+ getCurrentSaying( ): String
+ speak( )
+ toString( ): String

AudienceMember

+ AudienceMember(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

The AudienceMember class

Coffee time: How many **instance variable**s does an AudienceMember object have? Hint: an AudienceMember object is also a Person object.

The full AudienceMember code

```java
// Representation of an audience member watching the lottery.
public class AudienceMember extends Person
{
  // Constructor is given the person’s name.
  public AudienceMember(String name)
  {
    super(name);
  }  // AudienceMember


  // Returns the name of the type of Person.
  public String getPersonType()
  {
    return "Audience Member";
  } // getPersonType


  // Returns the Person’s current saying.
  public String getCurrentSaying()
  {
    return "Oooooh!";
  } // getCurrentSaying

} // class AudienceMember
```

The `TestAudienceMember` class

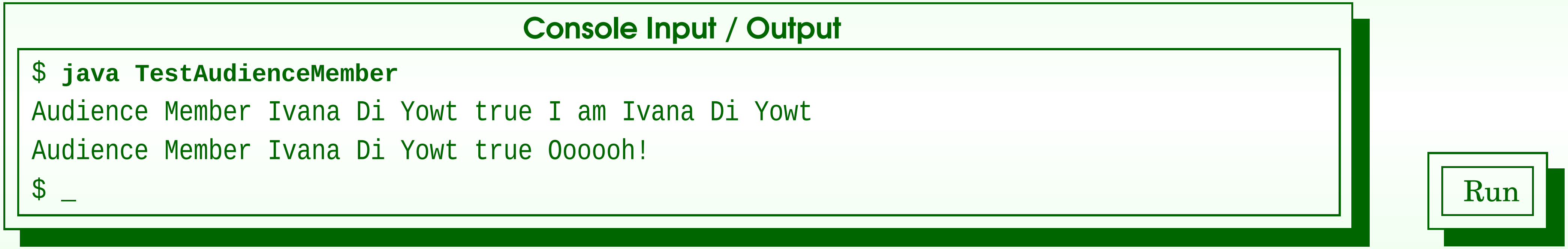
- Test in same way as `Person`.
- `toString()` implicitly used here
  - **inherit**ed from `Person`
  - invokes four other methods
    - `getPersonType()`, `getPersonName()`, `isHappy()` and `getLatestSaying()`.
  - `AudienceMember` **override**s `getPersonType()`, inherits others.
- Also use `speak()`
  - inherited from `Person`
  - invokes `getCurrentSaying()` overridden by `AudienceMember`.

The TestAudienceMember class

```java
// Create an AudienceMember and make them speak.
public class TestAudienceMember
{
  public static void main(String[] args)
  {
    AudienceMember audienceMember = new AudienceMember("Ivana Di Yowt");
    System.out.println(audienceMember);
    audienceMember.speak();
    System.out.println(audienceMember);
  } // main

} // class TestAudienceMember
```

Coffee time: Before looking at the test results in the next section, figure out what the output of the `TestAudienceMember` program should be.

Trying it



Console Input / Output

```
$ java TestAudienceMember
Audience Member Ivana Di Yowt true I am Ivana Di Yowt
Audience Member Ivana Di Yowt true Oooooh!
$ _
```

Coursework: Your first stock item!

(Summary only)

Write a **subclass** which **override**s some **instance method**s.

Section 5

The Punter class

Aim

AIM: To reinforce the ideas of **superclass**, **subclass**, **inheritance**, invoking the superclass **constructor method**, and **instance method**s that **override** another.

The Punter class

- Punters want to win, but not clever enough to play!
- Similar to `AudienceMembers`
  - but always unhappy.

```
// Representation of a person playing the lottery.
public class Punter extends Person
{
  // Constructor is given the person’s name.
  public Punter(String name)
  {
    super(name);
  }  // Punter


```

The Punter class

```
// Returns the name of the type of Person.
public String getPersonType()
{
  return "Punter";
} // getPersonType
```

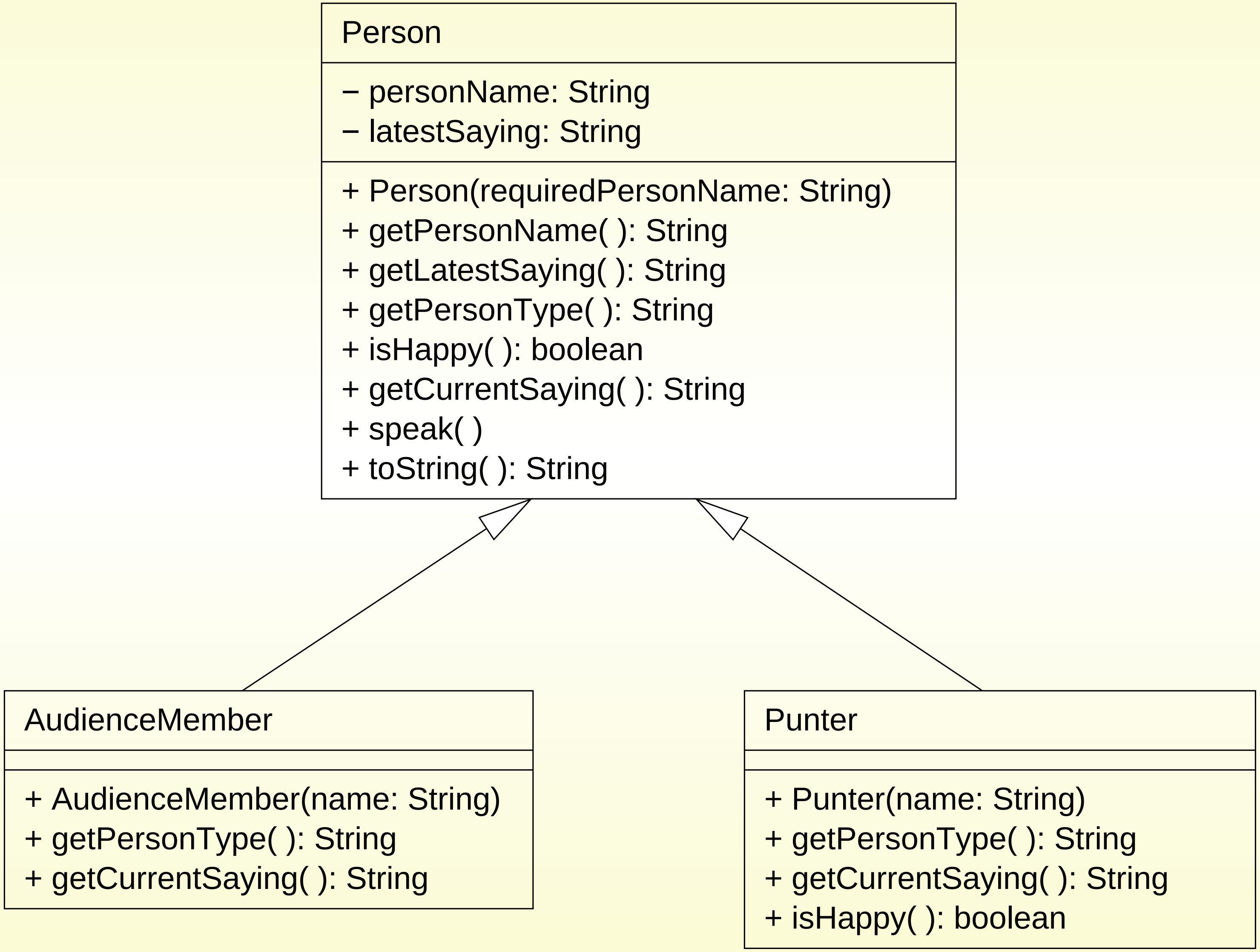
- Also **overrides** `isHappy()`.

```
// Returns whether or not the Person is happy.
public boolean isHappy()
{
  return false;
} // isHappy
```

The Punter class

```
  // Returns the Person’s current saying.
  public String getCurrentSaying()
  {
    return "Make me happy: give me lots of money";
  } // getCurrentSaying

} // class Punter
```

The Punter class



Person

− personName: String
− latestSaying: String

+ Person(requiredPersonName: String)
+ getPersonName( ): String
+ getLatestSaying( ): String
+ getPersonType( ): String
+ isHappy( ): boolean
+ getCurrentSaying( ): String
+ speak( )
+ toString( ): String

AudienceMember

+ AudienceMember(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Punter

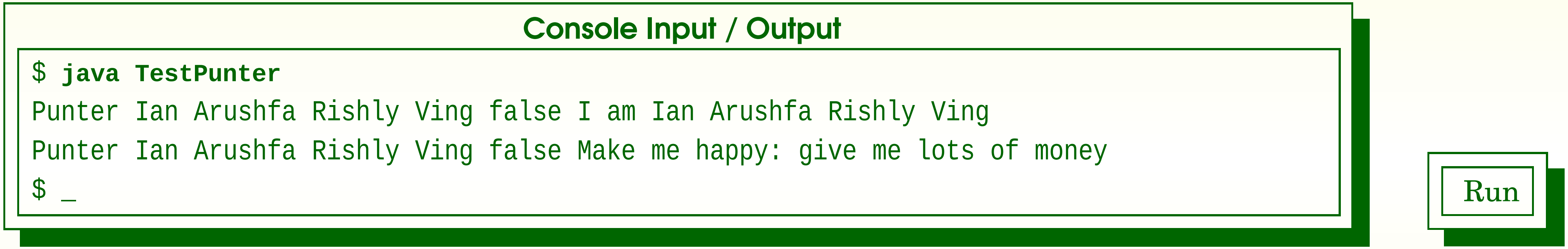
+ Punter(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String
+ isHappy( ): boolean

The TestPunter class

```java
// Create a Punter and make them speak.
public class TestPunter
{
  public static void main(String[] args)
  {
    Punter punter = new Punter("Ian Arushfa Rishly Ving");
    System.out.println(punter);
    punter.speak();
    System.out.println(punter);
  } // main

} // class TestPunter
```

Coffee time: Before looking at the test results in the next section, figure out what the output of the `TestPunter` program should be.

Trying it



Console Input / Output

```
$ java TestPunter
Punter Ian Arushfa Rishly Ving false I am Ian Arushfa Rishly Ving
Punter Ian Arushfa Rishly Ving false Make me happy: give me lots of money
$ _
```

- Notice he is not happy.

Coursework: Your catalogue

(Summary only)

Write another **subclass** which **override**s some **instance method**s.

Section 6

The Person abstract class

Aim

AIM: To introduce the concepts of **abstract class** and **abstract method**.

The `Person` abstract class

- Unsatisfactory aspects of what done so far
  - intend to make no **instance**s of `Person` directly
    - ∗ but what is stopping us do so in error?
  - Written code for **instance method**s `getPersonType()` and `getCurrentSaying()` in `Person`
    - ∗ yet every **subclass** will **override** them
    - ∗ so that code will never be used!

Inheritance: abstract class

- Can declare **class** as **abstract class**
  - no **instance**s can be made.
- Write **reserved word** **`abstract`** before **`class`** in heading.
- The **compiler** produces error if attempt to create direct instance.

Inheritance: abstract class

- E.g. likely do not want direct instances of `Vehicle`.

```
public abstract class Vehicle
{
  ...
} // class Vehicle

public class Bicycle extends Vehicle
{
  ...
} // class Bicycle
```

- This produces error.

```
Vehicle v = new Vehicle(...);
```

- This is allowed.

```
Bicycle b = new Bicycle(...);
```

The Person class

Coffee time: What about the following?

```
Vehicle v = new Bicycle(...);
```

The Person class

```java
// Representation of a person involved somehow in the lottery.
public abstract class Person
{
  // The name of the person.
  private final String personName;

  // The Person’s latest saying.
  private String latestSaying;


  // Constructor is given the person’s name.
  public Person(String requiredPersonName)
  {
    personName = requiredPersonName;
    latestSaying = "I am " + personName;
  } // Person


```

The Person class

```java
// Returns the Person's name.
public String getPersonName()
{
  return personName;
} // getPersonName


// Returns the Person's latest saying.
public String getLatestSaying()
{
  return latestSaying;
} // getLatestSaying
```

- Perhaps most valuable advantage of **abstract class** is **abstract methods**. . . .

Inheritance: abstract method

- An **abstract class** can have **abstract method**s
- These are **instance method**s which have
  - **modifier**s (not **`static`**)
    - but definitely **`abstract`**
  - **return type**
  - name
  - **method parameter**s
  - but no body
    - just semi-colon (`;`).
- This declares **method interface**
  - **method signature** and **return type**
- but not **method implementation**.

Inheritance: abstract method

- E.g. say there is no default way of determining if a vehicle can pass down a route.

```
public abstract class Vehicle
{
  ...
  public abstract boolean canPassDown(Route r);
  ...
} // class Vehicle
```

- Every **subclass** must
  - provide method implementation of all abstract methods
  - or be an abstract class.

Inheritance: abstract method

- An abstract method means
  - all non-abstract subclasses contain an instance method with this method interface
  - but method implementations provided by the subclasses.
- So no need to provide implementation that is never used if *every* subclass would **override** it.

Inheritance: abstract method

```
public class Bicycle extends Vehicle
{
  ...
  public boolean canPassDown(Route r)
  {
    ... Code for deciding if this bicycle can pass down the route.
  } // canPassDown
  ...
} // class Bicycle
```

Inheritance: abstract method

- When subclass defines non-abstract instance method also defined in **superclass**
  - we say it **override**s superclass one.
- When subclass defines non-abstract instance method declared as abstract method in superclass
  - we say it provides **method implementation**.
- Override is *replacing* method implementation from superclass.

The Person class

- No default implementation for `getPersonType()`.

```java
// Returns the name of the type of Person.
public abstract String getPersonType();
```

- Whereas most subclasses are always happy, so we have **default implementation** of `isHappy()`
  - **inherit**ed by most subclasses (happy ones)
  - others override it.

```java
// Returns whether or not the Person is happy.
public boolean isHappy()
{
  return true;
} // isHappy
```

The Person class

- Current saying *always* specific to kind of person.

```
  // Returns the Person’s current saying.
  public abstract String getCurrentSaying();
```

The Person class

- Rest is same as before.

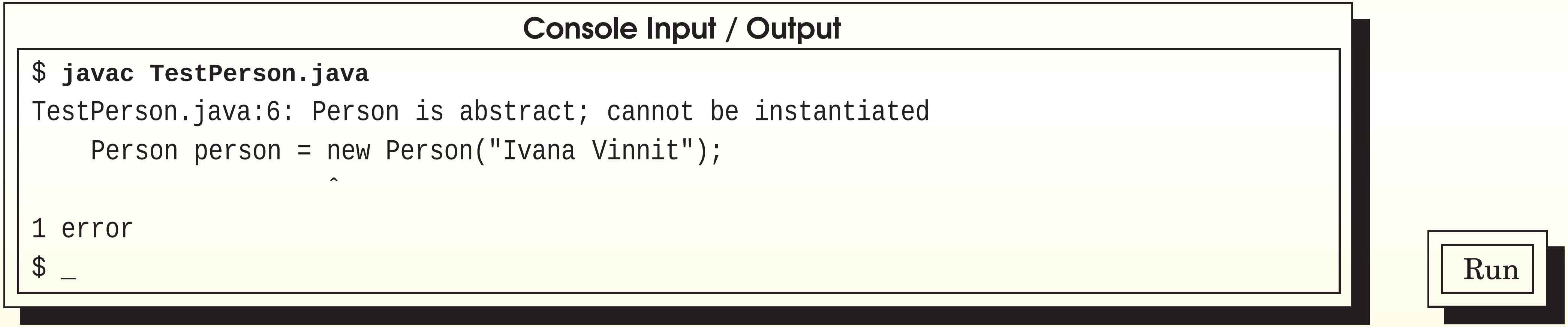
```
  // Causes the person to speak by updating their latest saying from
  // their current saying.
  public void speak()
  {
    latestSaying = getCurrentSaying();
  } // speak


  // Mainly for testing.
  public String toString()
  {
    return getPersonType() + " " + getPersonName()
           + " " + isHappy() + " " + getLatestSaying();
  } // toString

} // class Person
```

The AudienceMember and Punter classes

- No changes needed to `AudienceMember` and `Punter`.
- But no longer **override** `getPersonType()` and `getCurrentSaying()`
  - have **method implementation**s of them.
- `Punter` still overrides `isHappy()`.

Trying it

- Try re**compile** `TestPerson`.



Console Input / Output

```
$ javac TestPerson.java
TestPerson.java:6: Person is abstract; cannot be instantiated
    Person person = new Person("Ivana Vinnit");
                    ^
1 error
$ _
```

Coursework: An abstract stock item

(Summary only)

Make a **class** into an **abstract class**.

Section 7

The remaining simple subclasses of Person

Aim

AIM: To reinforce the concepts covered in the chapter so far, and introduce the ideas of **polymorphism** and **dynamic method binding**. We also meet **final class**es and **final method**s.

The remaining simple subclasses of `Person`

- Develop remaining `simple' **subclass**es
  - `Director`, `Psychic` and `TVHost`.
- Create `TestPersonSubclasses` to test all so far.

The Director class

```java
// Representation of a director of the lottery company.
public class Director extends Person
{
  // Constructor is given the person’s name.
  public Director(String name)
  {
    super(name);
  } // Director
```

The Director class

- Provide **method implementations** for **abstract methods**.

```
  // Returns the name of the type of Person.
  public String getPersonType()
  {
    return "Director";
  } // getPersonType


  // Returns the Person’s current saying.
  public String getCurrentSaying()
  {
    return "This business is MY pleasure";
  } // getCurrentSaying

} // class Director
```

Coffee time: Are directors happy or unhappy? (Daft question?)

The Psychic class

```java
// Representation of a psychic entertainer for the lottery.
public class Psychic extends Person
{
  // Constructor is given the person's name.
  public Psychic(String name)
  {
    super(name);
  }  // Psychic


  // Returns the name of the type of Person.
  public String getPersonType()
  {
    return "Psychic";
  } // getPersonType

```

The Psychic class

```

    // Returns the Person’s current saying.
    public String getCurrentSaying()
    {
      return "I can see someone very happy!";
    } // getCurrentSaying

} // class Psychic
```

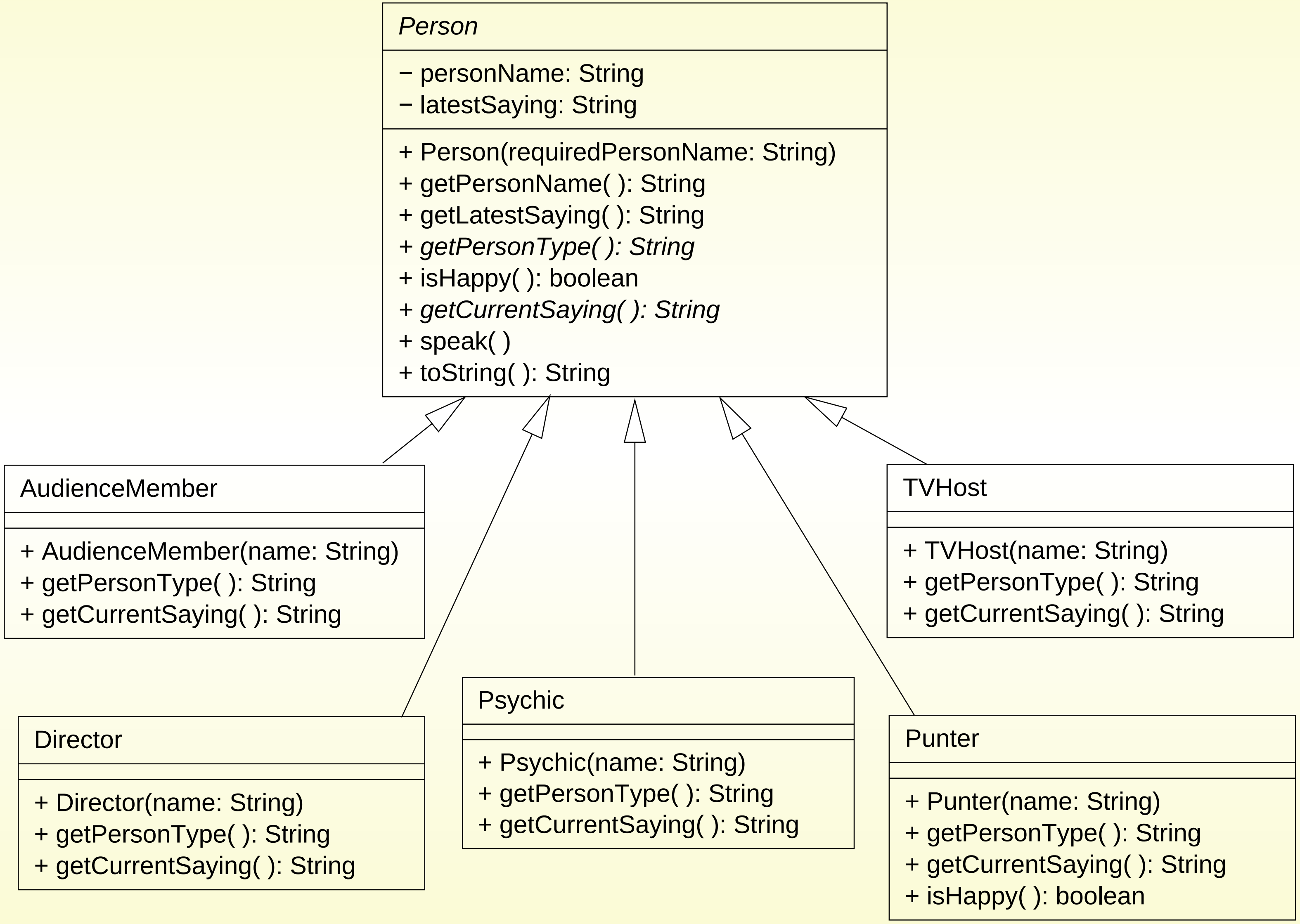
The TVHost class

```java
// Representation of a TV Host fronting the lottery TV programme.
public class TVHost extends Person
{
  // Constructor is given the person’s name.
  public TVHost(String name)
  {
    super(name);
  }  // TVHost


  // Returns the name of the type of Person.
  public String getPersonType()
  {
    return "TV Host";
  } // getPersonType

```

The TVHost class

```

  // Returns the Person’s current saying.
  public String getCurrentSaying()
  {
    return "Welcome, suckers!";
  } // getCurrentSaying

} // class TVHost
```

Latest inheritance hierarchy



Person

− personName: String
− latestSaying: String

+ Person(requiredPersonName: String)
+ getPersonName( ): String
+ getLatestSaying( ): String
+ *getPersonType( ): String*
+ isHappy( ): boolean
+ *getCurrentSaying( ): String*
+ speak( )
+ toString( ): String

AudienceMember

+ AudienceMember(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

TVHost

+ TVHost(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Director

+ Director(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Psychic

+ Psychic(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Punter

+ Punter(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String
+ isHappy( ): boolean

The TestPersonSubclasses class

- Tests needed for each **subclass** of Person are the same
  - create **instance**, print, speak, print.
- Have one program to test all
  - rather than one for each subclass.
- Have **array** of Person
  - containing one instance of each subclass
  - **loop** through array testing each of them.

The TestPersonSubclasses class

```java
// Create one of each type of person, and make them speak.
public class TestPersonSubclasses
{
  public static void main(String[] args)
  {
    Person[] persons =
      {
        new AudienceMember("Ivana Di Yowt"),
        new Director("Sir Lance Earl Otto"),
        new Psychic("Miss T. Peg de Gowt"),
        new Punter("Ian Arushfa Rishly Ving"),
        new TVHost("Terry Bill Woah B’Gorne")
      };
```

[a]

[a] This chapter is dedicated to Terry Wogan, 3rd August 1938 – 31st January 2016.

The `TestPersonSubclasses` class

- Instance of subclass is also instance of its **superclass**
  - e.g. first **array element** is (reference to) both an `AudienceMember` and a `Person`.
- Multiplicity of types – known as **polymorphism**
  - the **object**s are **polymorphic**.

Inheritance: polymorphism

- An **instance** of **subclass** is also instance of **superclass**.
  - E.g. **class** `Bicycle` is subclass of `Vehicle`
    - so instance of `Bicycle` **is a** `Bicycle`
    - also **is a** `Vehicle`
  - it has *both* these forms.
- Is **polymorphic** – means ‘has many forms’.
- Java **polymorphism** achieved by **inheritance**.

The TestPersonSubclasses class

```
    for (Person person : persons)
      testPerson(person);
  } // main


  // Make the given person speak, reporting the before and after toString.
  private static void testPerson(Person person)
  {
    System.out.println("--------------------------------------------------");
    System.out.println(person);
    person.speak();
    System.out.println(person);
  } // testPerson

} // class TestPersonSubclasses
```

The `TestPersonSubclasses` class

- Body of `testPerson()` calls `toString()` and `speak()` **instance method**s of its **method parameter**.

- `toString()` calls
  - `getPersonType()` – **method implementation** in subclass
  - `getPersonName()` – **inherit**ed by subclass
  - `isHappy()` – **inherit**ed by subclass except `Punter` which **override**s it
  - `getLatestSaying()` – **inherit**ed by subclass.

- `speak()` calls
  - `getCurrentSaying()` – **method implementation** in subclass.

- When **compiler** looks at `Person` code,
  cannot know which actual method will be used when program is **run**
  - different versions used at different moments for same **method call**s!

Inheritance: polymorphism: dynamic method binding

- A **class** might have **subclass**
  - which might **override** some of **instance methods**.
- And **abstract methods** are designed to have different **method implementations** in different subclasses.
- When **compiler** produces **byte code** for instance **method call**
  - does not know which actual **method implementation** will get used
  - same call can invoke different versions of method at different times
    - ∗ depending on **run time** value of **object reference**.

Inheritance: polymorphism: dynamic method binding

- E.g. Assume `PoshCar` does not override `emergencyStop()` but `Bicycle` does.

```
Vehicle funRide = Math.random() < 0.5 ? new PoshCar(...) : new Bicycle(...);
funRide.emergencyStop();
```

- At run time, reference stored in `funRide` refers either to
  - `PoshCar` object – `emergencyStop()` from `Vehicle` is called
  - or `Bicycle` object – `emergencyStop()` from `Bicycle` is used.
- Process of determining actual method at run time known as **dynamic method binding**.
- Consequence for programmers – our code might not behave as we expect in some subclass where some instance methods are replaced with ones that do something we did not expect.
- Our **private** instance methods are safe
  - cannot be overridden because not visible in any subclass.

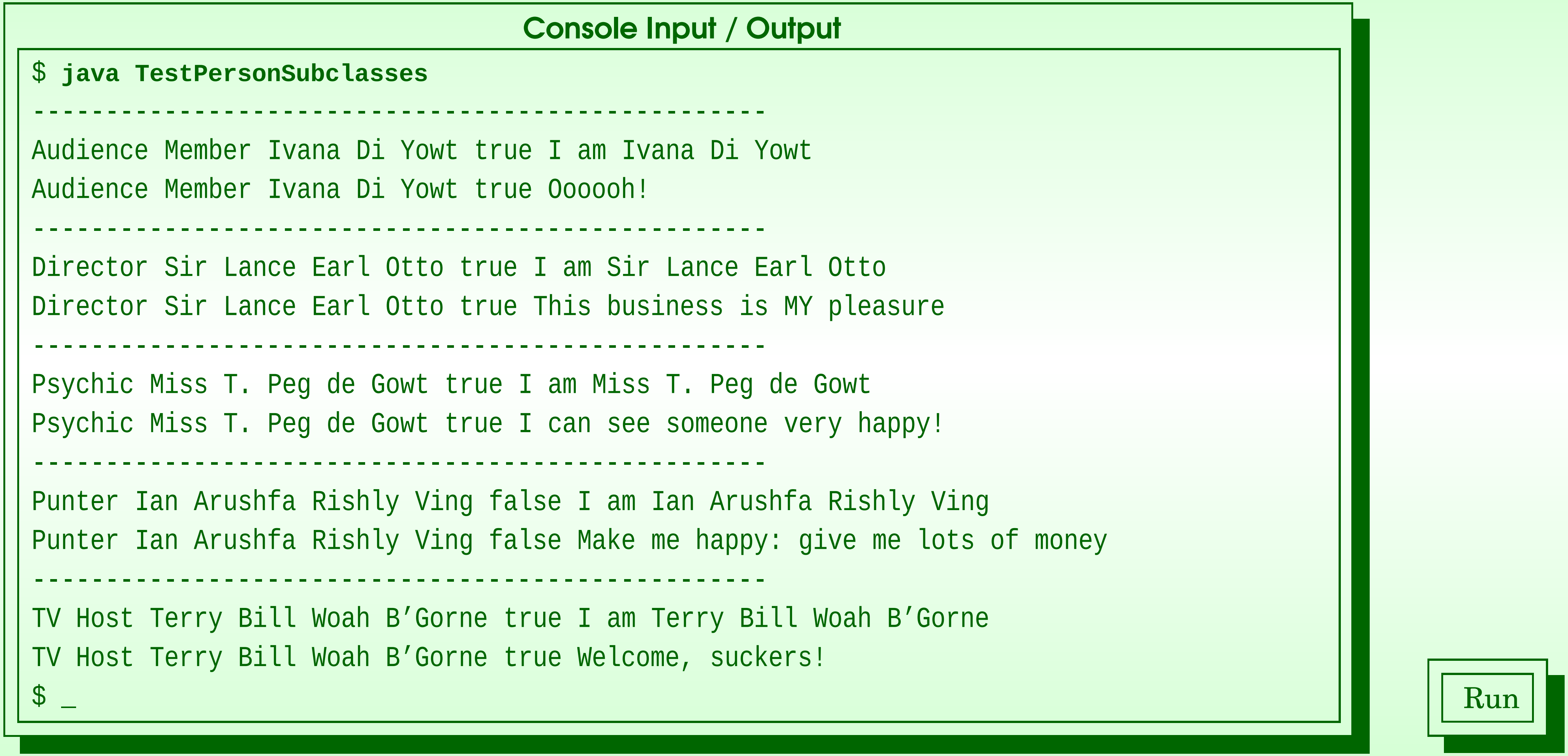
Inheritance: final methods and classes

- If wish that no **subclass** may **override** a **public instance method**
  - make it **final method** – include **reserved word `final`** in heading.
- Use with care: future requirements may mean subclass not yet written needs own version of instance method!
- Also can make a **class** into **final class**
  - write **`final`** in class heading
  - cannot have any subclasses.

The `TestPersonSubclasses` class

Coffee time: Look at the instance methods of the `Person` class and decide which might appropriately be declared as **final methods**. For example, will any subclass need to have its own version of `toString()`?

Trying it



```
$ java TestPersonSubclasses
--------------------------------------------------
Audience Member Ivana Di Yowt true I am Ivana Di Yowt
Audience Member Ivana Di Yowt true Oooooh!
--------------------------------------------------
Director Sir Lance Earl Otto true I am Sir Lance Earl Otto
Director Sir Lance Earl Otto true This business is MY pleasure
--------------------------------------------------
Psychic Miss T. Peg de Gowt true I am Miss T. Peg de Gowt
Psychic Miss T. Peg de Gowt true I can see someone very happy!
--------------------------------------------------
Punter Ian Arushfa Rishly Ving false I am Ian Arushfa Rishly Ving
Punter Ian Arushfa Rishly Ving false Make me happy: give me lots of money
--------------------------------------------------
TV Host Terry Bill Woah B'Gorne true I am Terry Bill Woah B'Gorne
TV Host Terry Bill Woah B'Gorne true Welcome, suckers!
$ _
```

Coursework: More stock items

(Summary only)

Make some more **subclass**es and explore **polymorphism** and **dynamic method binding**.

Section 8

The MoodyPerson classes

Aim

AIM: To introduce the ideas of adding more **object state** and **instance methods** in a **subclass**, testing for an **instance** of a particular **class**, and **cast**ing to a subclass. We also see how a **constructor method** can invoke another from the same class.

The MoodyPerson classes

- Coming up: less simple **subclass**es of Person.

Name	**Brief description**
Teenager	Just for fun – a person that can be made to be happy or unhappy at will.
CleverPunter	Someone who actually plays the lottery.
Worker	Someone who makes balls and fills up a lottery machine.
TraineeWorker	A worker who gets the ball numbers wrong sometimes.

- All neither always happy, nor always unhappy.
- Suggests another subclass of Person called MoodyPerson
  - above can be subclasses of MoodyPerson
  - **inherit** the mood changing properties.

The MoodyPerson classes

- Here develop `MoodyPerson` and `Teenager`
  - others interact with lottery games, so wait until those are done.
- Also add code to `TestPersonSubclasses` to test **instance**s of `MoodyPerson`.

The `MoodyPerson` class

- Don't want any direct instances of `MoodyPerson`....

```
// Representation of a person involved in the lottery
// who can change their happiness state.
public abstract class MoodyPerson extends Person
{
```

- Need **instance variable** to record if currently happy
  - adding more **object state**.

Inheritance: adding more object state

- A **subclass** is **extension** of its **superclass**.
  - in general can add more properties.
- One way of **extend**ing
  - add more **object state**
  - i.e. more **instance variable**s.

The MoodyPerson class

```java
// The state of the Person's happiness.
private boolean isHappyNow;
```

- Two **method parameters** for **constructor method**
  - one passed to **superclass constructor call**
  - one used here.

```java
// Constructor is given the person's name and initial happiness.
public MoodyPerson(String name, boolean initialHappiness)
{
  super(name);
  isHappyNow = initialHappiness;
} // MoodyPerson
```

Coffee time: Why must the call to **super** be the first statement?

The `MoodyPerson` class

- Also have second constructor
  - just takes name of person
  - assumes person initially happy.

Method: constructor methods: more than one: using this

- The **method parameter**s to **constructor method**s often values for **instance variable**s.
- When have several instance variables might have multiple constructors
  - some assume default values for some instance variables.
- E.g. Might allow **construct**ing a `Point` for origin by supplying no **method argument**s.

Method: constructor methods: more than one: using this

```
public class Point
{
  private final double x, y;

  public Point(double requiredX, double requiredY)
  {
    x = requiredX;
    y = requiredY;
  } // Point

  public Point()
  {
    x = 0;
    y = 0;
  } // Point

  ...
} // class Point
```

Method: constructor methods: more than one: using this

- Second constructor rather like wrapper around first.
- Could make relationship explicit
  - actually call first from second
  - using **reserved word** **`this`** with desired arguments.
- E.g.

```
public Point()
{
  this(0, 0);
} // Point
```

- Known as **alternative constructor call**
  - must be first **statement** in constructor body
  - class must have another constructor with matching parameters.

The `MoodyPerson` class

```java
// Alternative constructor is given the person's name
// and initial happiness is assumed to be true.
public MoodyPerson(String name)
{
  this(name, true);
}  // MoodyPerson
```

- Have **override** for `isHappy()`.

```java
// Returns whether or not the Person is happy.
public boolean isHappy()
{
  return isHappyNow;
} // isHappy
```

- Need method to set state of happiness. . . .

Inheritance: adding more instance methods

- Another way of **extend**ing **superclass**
  - add more **instance method**s.
- Especially likely if subclass also has additional **instance variable**s.

The MoodyPerson class

```
  // Sets the happiness of the person to the given state.
  public void setHappy(boolean newHappiness)
  {
    isHappyNow = newHappiness;
  } // setHappy

} // class MoodyPerson
```

The Teenager class

- Nothing to do with Lottery, per se, just for ‘fun’
  - end user can create `Teenager` to model big sister or brother.

```
// Representation of a teenager.
public class Teenager extends MoodyPerson
{
```

- Teenagers always start off being unhappy.
- Follow chain of constructor calls. . . .

```
  // Constructor is given the person’s name.
  public Teenager(String name)
  {
    super(name, false);
  } // Teenager
```

The Teenager class

- Provide **method implementations** `getPersonType()` and `getCurrentSaying()`.
- These **method interfaces inherited** from `MoodyPerson`
  - which inherited them from `Person` without implementing them.

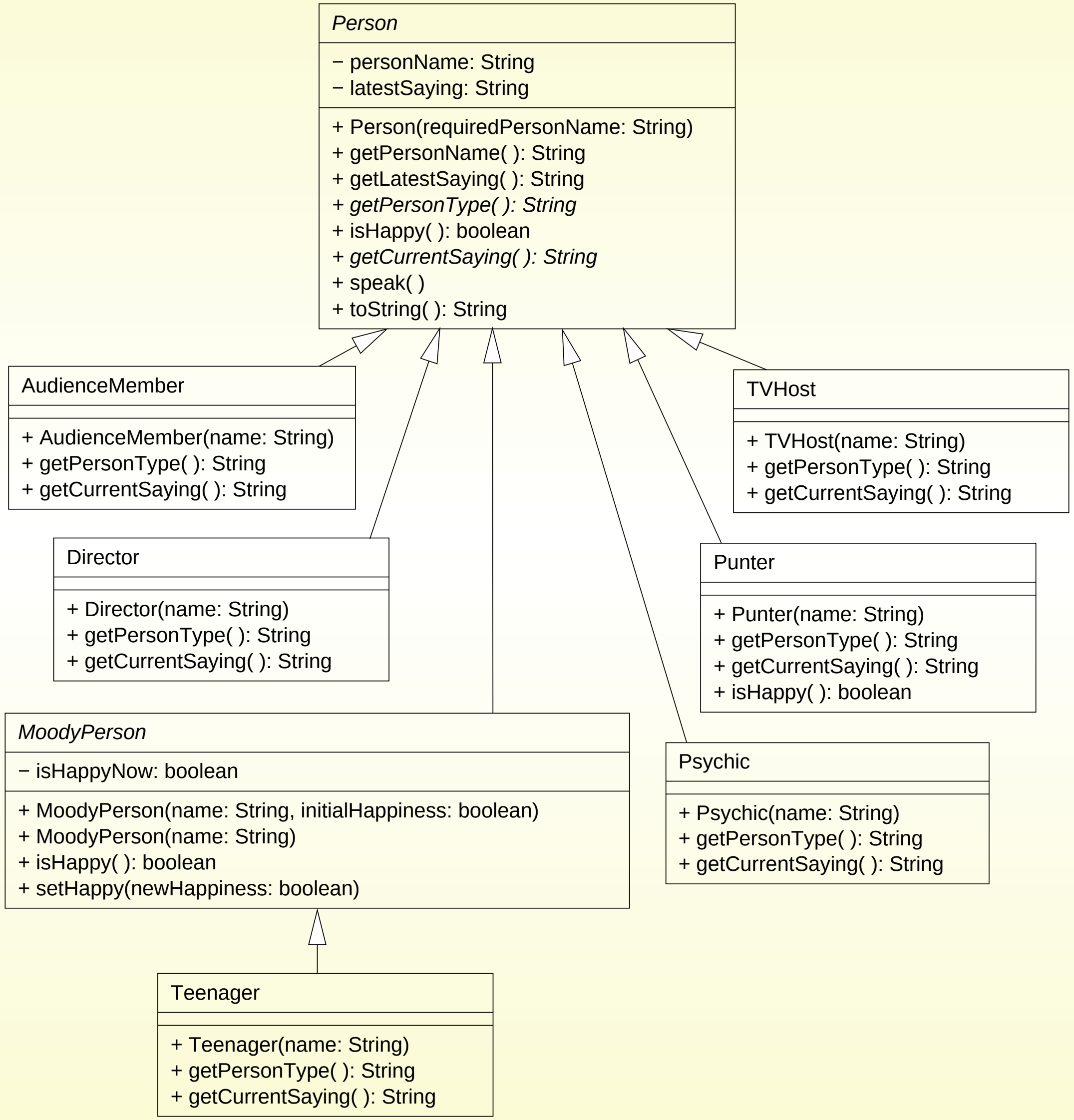
```java
// Returns the name of the type of Person.
public String getPersonType()
{
  return "Teenager";
} // getPersonType
```

The Teenager class

- Current saying depends on mood!

```java
  // Returns the Person's current saying.
  public String getCurrentSaying()
  {
    if (isHappy())
      return "Isn't life wonderful?";
    else
      return "It's not fair!";
  } // getCurrentSaying

} // class Teenager
```

The Teenager class



Person

− personName: String
− latestSaying: String

+ Person(requiredPersonName: String)
+ getPersonName( ): String
+ getLatestSaying( ): String
+ *getPersonType( ): String*
+ isHappy( ): boolean
+ *getCurrentSaying( ): String*
+ speak( )
+ toString( ): String

AudienceMember

+ AudienceMember(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

TVHost

+ TVHost(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Director

+ Director(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Punter

+ Punter(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String
+ isHappy( ): boolean

MoodyPerson

− isHappyNow: boolean

+ MoodyPerson(name: String, initialHappiness: boolean)
+ MoodyPerson(name: String)
+ isHappy( ): boolean
+ setHappy(newHappiness: boolean)

Psychic

+ Psychic(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Teenager

+ Teenager(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

The `Teenager` class

Coffee time: List the **instance method**s of the `Teenager` class, and for each identify where they originated. State whether they are **inherit**ed as is, **override** one from a **superclass**, or are a **method implementation** of an **abstract method**.

The TestPersonSubclasses class

```java
// Create one of each type of person, and make them speak.
public class TestPersonSubclasses
{
  public static void main(String[] args)
  {
    Person[] persons =
      {
        new AudienceMember("Ivana Di Yowt"),
        new Director("Sir Lance Earl Otto"),
        new Psychic("Miss T. Peg de Gowt"),
        new Punter("Ian Arushfa Rishly Ving"),
        new Teenager("Homer Nalzone"),
        new TVHost("Terry Bill Woah B'Gorne")
      };

    for (Person person : persons)
      testPerson(person);
  } // main
```

The `TestPersonSubclasses` class

- Alter `testPerson()`
  - if given `Person` also `MoodyPerson`
    - calls new method `testMoodyPerson()`.

Inheritance: testing for an instance of a class

- The **reserved word** **`instanceof`**
  - **binary infix operator**
  - left **operand** is **object reference**
  - right operand is **class** name.
  - yields **`true`**
    iff reference refers to object which **is an instance** of named class.
- E.g. if `Tandem` is subclass of `Bicycle`:

```
Vehicle vehicle = new Tandem(...);
... Code that might change what vehicle refers to.
if (vehicle instanceof Bicycle)
  ... Code that is only run if vehicle is still referring to a Bicycle,
  ... perhaps still the original Tandem.
```

Inheritance: casting to a subclass

- An **instance** of **subclass is a**n instance of **superclass** too.
- So item of subclass **type**
  can always be used wherever superclass type required.
- E.g.

```
    Vehicle vehicle1 = new Bicycle(...);
```

- But not every instance of superclass
  is instance of a particular subclass – obviously.
- So item of superclass type
  cannot automatically be used where subclass type is required.

Inheritance: casting to a subclass

- E.g. not permitted.

```
Vehicle vehicle1 = new Bicycle(...);
...
Bicycle bicycle1 = vehicle1;
```

- `vehicle1` is definitely type `Vehicle` – but value might not be a `Bicycle`.

- If sure is safe to treat item of superclass type as particular subclass type
  - can **cast** value to that subclass
    - ∗ precede value with subclass name in brackets.

- E.g. if sure after `...` that `vehicle1` is still **reference** to a `Bicycle`:

```
Vehicle vehicle1 = new Bicycle(...);
...
Bicycle bicycle1 = (Bicycle)vehicle1;
```

Inheritance: casting to a subclass

- The **compiler** accepts this on face value
  - but type cast is checked at **run time**
  - if value being cast is not reference to object of that type
    - `ClassCastException` object **throw**n.
- Note: **class** cast does not change object being cast
  - merely *checks* that object is already of stated type.
- Contrast with **primitive type** cast
  - e.g. convert **`double`** into **`int`**
  - creates new value from old one.

The TestPersonSubclasses class

```java
// Make the given person speak, reporting the before and after toString.
private static void testPerson(Person person)
{
  System.out.println("--------------------------------------------------");
  System.out.println(person);
  person.speak();
  System.out.println(person);
  if (person instanceof MoodyPerson)
    testMoodyPerson((MoodyPerson)person);
} // testPerson
```

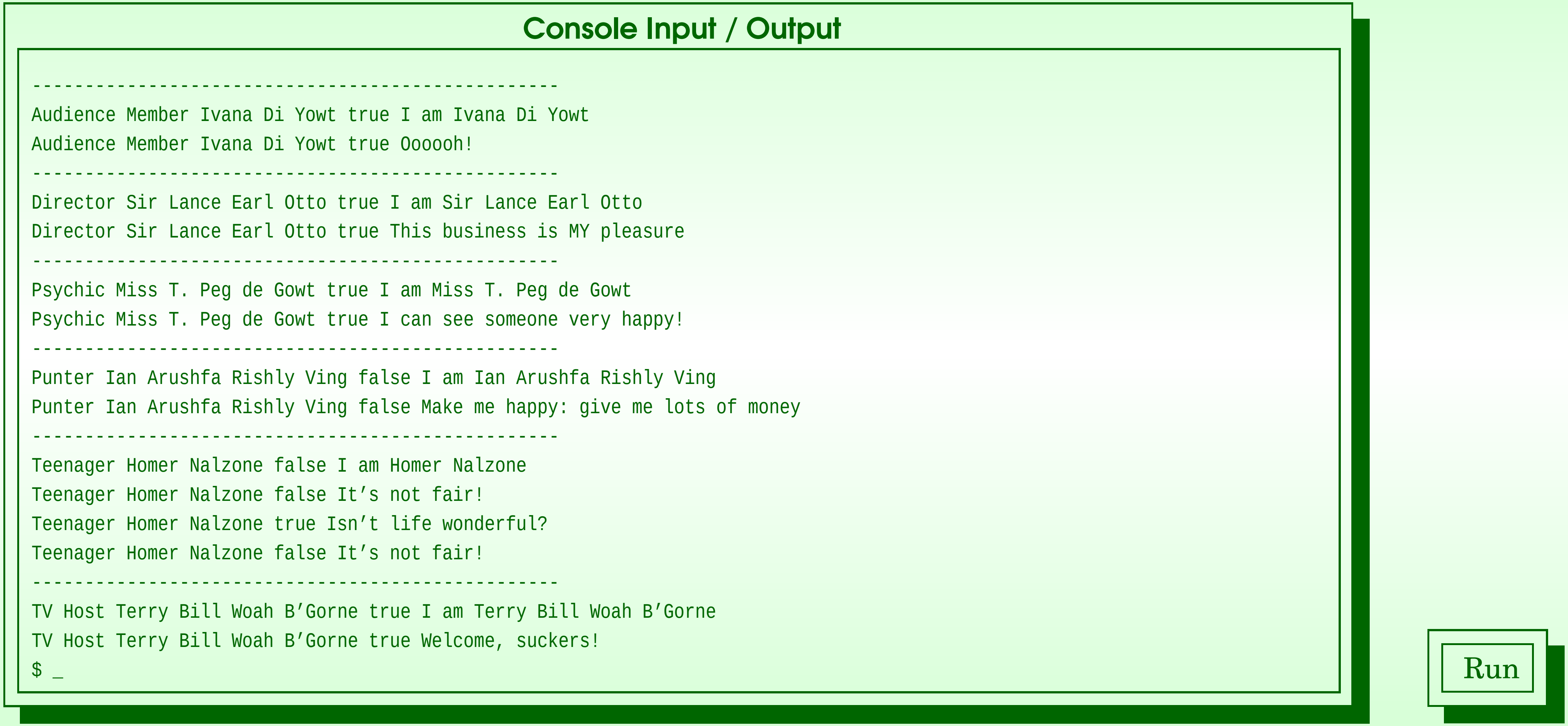
The TestPersonSubclasses class

```
  // Make the given moody person change happiness then speak,
  // reporting the after toString; all twice.
  private static void testMoodyPerson(MoodyPerson moodyPerson)
  {
    for (int count = 1; count <= 2; count++)
    {
      moodyPerson.setHappy(! moodyPerson.isHappy());
      moodyPerson.speak();
      System.out.println(moodyPerson);
    } // for
  } // testMoodyPerson

} // class TestPersonSubclasses
```

The `TestPersonSubclasses` class

Coffee time: In the code above, what would happen if we did not cast `person` to `MoodyPerson` when passing its value to `testMoodyPerson()`? What if that method parameter was declared to be of **type** `Person`?

Trying it



```
--------------------------------------------------
Audience Member Ivana Di Yowt true I am Ivana Di Yowt
Audience Member Ivana Di Yowt true Oooooh!
--------------------------------------------------
Director Sir Lance Earl Otto true I am Sir Lance Earl Otto
Director Sir Lance Earl Otto true This business is MY pleasure
--------------------------------------------------
Psychic Miss T. Peg de Gowt true I am Miss T. Peg de Gowt
Psychic Miss T. Peg de Gowt true I can see someone very happy!
--------------------------------------------------
Punter Ian Arushfa Rishly Ving false I am Ian Arushfa Rishly Ving
Punter Ian Arushfa Rishly Ving false Make me happy: give me lots of money
--------------------------------------------------
Teenager Homer Nalzone false I am Homer Nalzone
Teenager Homer Nalzone false It’s not fair!
Teenager Homer Nalzone true Isn’t life wonderful?
Teenager Homer Nalzone false It’s not fair!
--------------------------------------------------
TV Host Terry Bill Woah B’Gorne true I am Terry Bill Woah B’Gorne
TV Host Terry Bill Woah B’Gorne true Welcome, suckers!
$ _
```

Coursework: Lots of different mouse mats!

(Summary only)

Have additional state in some **subclass**es.

Section 9

The `Ball` class

Aim

AIM: This section is mainly for progressing the development of the program, however the `java.awt.Color` **class** is introduced.

The `Ball` class

- Away from **subclass**es of `Person` – onto lottery games.
- Balls have **integer** number, and colour.
- Use `java.awt.Color` to represent colour.

GUI API: `Color`

- `java.awt.Color` implements colours for **graphical user interface**s
  - each `Color` **object** has four values in range 0 to 255
    - ∗ red, green, blue and alpha (opacity).
- Class has **class constant**s
  containing **reference**s to `Color` objects for some common colours.

```
public static final Color black     = new Color(0,     0,   0, 255);
public static final Color white     = new Color(255, 255, 255, 255);
public static final Color red       = new Color(255,   0,   0, 255);
public static final Color green     = new Color(0,   255,   0, 255);
public static final Color blue      = new Color(0,   0,   255, 255);
```

GUI API: Color

```
public static final Color lightGray = new Color(192, 192, 192, 255);
public static final Color gray      = new Color(128, 128, 128, 255);
public static final Color darkGray  = new Color(64,   64,  64, 255);

public static final Color pink      = new Color(255, 175, 175, 255);
public static final Color orange    = new Color(255, 200,   0, 255);
public static final Color yellow    = new Color(255, 255,   0, 255);
public static final Color magenta   = new Color(255,   0, 255, 255);
public static final Color cyan      = new Color(0,   255, 255, 255);
```

- An **instance method** `getRGB()`
  - **return**s unique **`int`** for each **equivalent** colour
    - ∗ based on four component values.

The Ball class

```java
import java.awt.Color;

// Representation of a lottery ball, comprising colour and value.
public class Ball
{
  // The numeric value of the ball.
  private final int value;

  // The colour of the ball.
  private final Color colour;


  // A ball is constructed by giving a number and a colour.
  public Ball(int requiredValue, Color requiredColour)
  {
    value = requiredValue;
    colour = requiredColour;
  } // Ball


```

The Ball class

```
  // Returns the numeric value of the ball.
  public int getValue()
  {
    return value;
  } // getValue


  // Returns the colour of the ball.
  public Color getColour()
  {
    return colour;
  } // getColour


```

The Ball class

```java
// Compares this ball’s value with another, returning
// < 0 if this ball’s value is smaller than the other’s,
// > 0 if it is greater, or if the values are equal then
//                       compare the RGB numbers of the colours instead.
public int compareTo(Ball other)
{
  if (value == other.value)
    return colour.getRGB() - other.colour.getRGB();
  else
    return value - other.value;
} // compareTo


```

The Ball class

```
  // Mainly for testing.
  public String toString()
  {
    return "Ball " + value + " " + colour;
  } // toString

} // class Ball
```

Coffee time: Is an **instance** of Ball a **mutable object** or an **immutable object**?

Section 10

The BallContainer classes

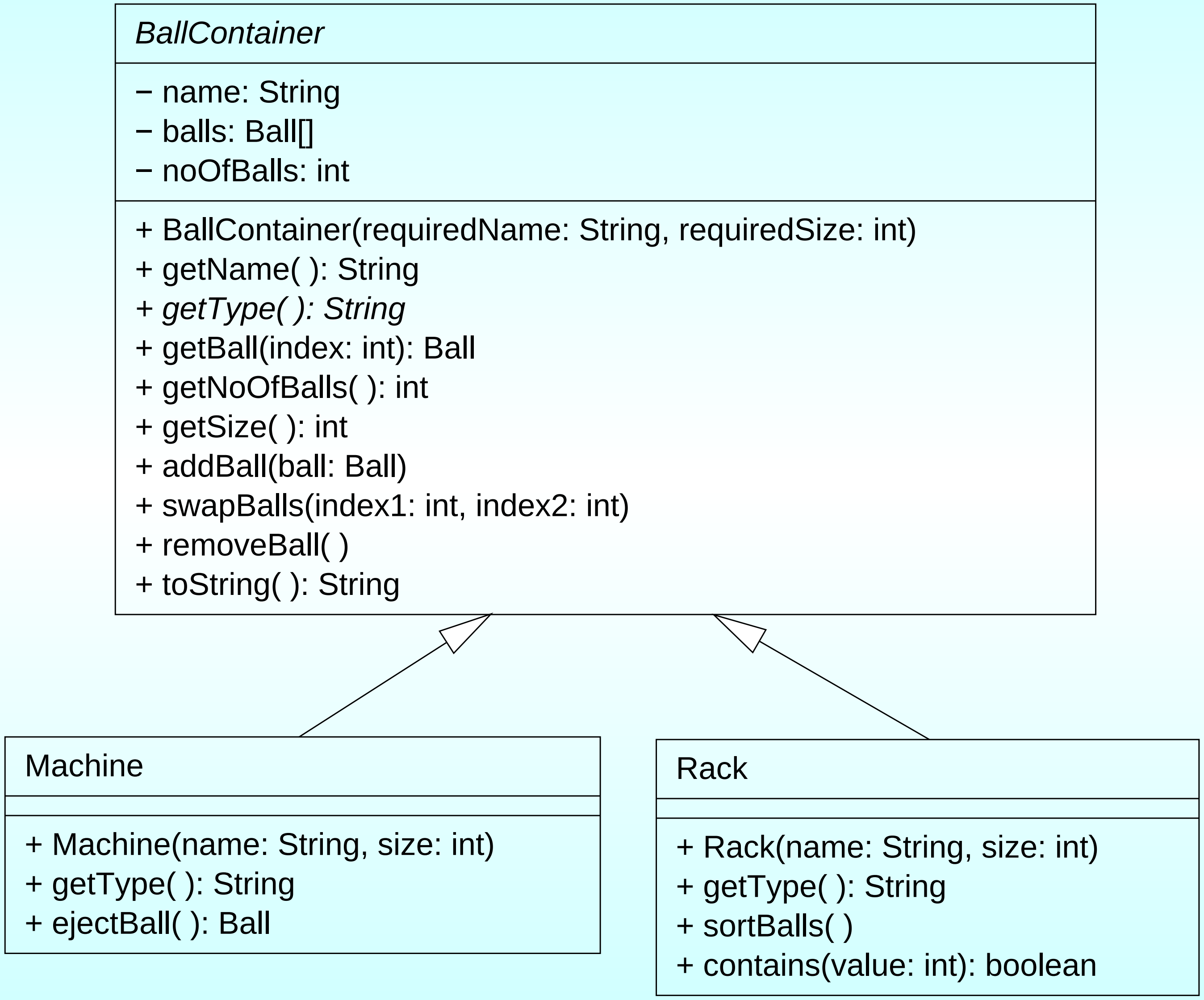
Aim

AIM: To show another example of **inheritance**. We also see how to delete an **array element** from an unsorted **partially filled array**.

The `BallContainer` classes

- Lottery games consist of machine and landing rack
  - both (can) contain balls
  - have some features in common
  - some features specific.
- Suggests **superclass** `BallContainer` for common features
  - two **subclass**es `Machine` and `Rack`
  - with specific features.
- We have `TestBallContainers` too, but do not show here.

The BallContainer classes



BallContainer

− name: String
− balls: Ball[]
− noOfBalls: int

+ BallContainer(requiredName: String, requiredSize: int)
+ getName( ): String
+ *getType( ): String*
+ getBall(index: int): Ball
+ getNoOfBalls( ): int
+ getSize( ): int
+ addBall(ball: Ball)
+ swapBalls(index1: int, index2: int)
+ removeBall( )
+ toString( ): String

Machine

+ Machine(name: String, size: int)
+ getType( ): String
+ ejectBall( ): Ball

Rack

+ Rack(name: String, size: int)
+ getType( ): String
+ sortBalls( )
+ contains(value: int): boolean

The BallContainer classes

- Note: places where we might sensibly **throw exceptions**
  - keep simple here
  - revisit during revisit to exceptions.

The BallContainer class

```java
// Representation of a container of balls for the lottery,
// with a fixed size and zero or more balls in a certain order.
public abstract class BallContainer
{
  // The name of the BallContainer.
  private final String name;

  // The balls contained in the BallContainer.
  private final Ball[] balls;

  // The number of balls contained in the BallContainer.
  // These are stored in balls, indexes 0 to noOfBalls - 1.
  private int noOfBalls;
```

The BallContainer class

Coffee time: Is a BallContainer always full? Does it have a fixed size?

```
// Constructor is given the name and size.
public BallContainer(String requiredName, int requiredSize)
{
  name = requiredName;
  balls = new Ball[requiredSize];
  noOfBalls = 0;
} // BallContainer
```

- The **accessor methods**. . . .

The BallContainer class

```
  // Returns the BallContainer's name.
  public String getName()
  {
    return name;
  } // getName


  // Returns the name of the type of BallContainer.
  public abstract String getType();


  // Returns the Ball at the given index in the BallContainer,
  // or null if that index is not in the range 0 to noOfBalls - 1.
  public Ball getBall(int index)
  {
    if (index >= 0 && index < noOfBalls)
      return balls[index];
    else
      return null;
  } // getBall;
```

The BallContainer class

```java
// Returns the number of balls in the BallContainer.
public int getNoOfBalls()
{
  return noOfBalls;
} // getNoOfBalls


// Returns the size of the BallContainer.
public int getSize()
{
  return balls.length;
} // getSize
```

The BallContainer class

- And **mutator methods**.

```
// Adds the given ball into the BallContainer, at the next highest unused
// index position. Has no effect if the BallContainer is full.
public void addBall(Ball ball)
{
  if (noOfBalls < balls.length)
  {
    balls[noOfBalls] = ball;
    noOfBalls++;
  } // if
} // addBall


```

The BallContainer class

```
// Swaps the balls at the two given index positions.
// Has no effect if either index is not in the range 0 to noOfBalls - 1.
public void swapBalls(int index1, int index2)
{
  if (index1 >= 0 && index1 < noOfBalls
      && index2 >=0 && index2 < noOfBalls)
  {
    Ball thatWasAtIndex1 = balls[index1];
    balls[index1] = balls[index2];
    balls[index2] = thatWasAtIndex1;
  } // if
} // swapBalls;


```

The BallContainer class

```
  // Removes the Ball at the highest used index position.
  // Has no effect if the BallContainer is empty.
  public void removeBall()
  {
    if (noOfBalls > 0)
      noOfBalls--;
  } // removeBall
```

- And toString().

```
  // Mainly for testing.
  public String toString()
  {
    String result = getType() + " " + name + "(<=" + balls.length + ")";
    for (int index = 0; index < noOfBalls; index++)
      result += String.format("%n%d %s", index, balls[index]);
    return result;
  } // toString

} // class BallContainer
```

The Machine class

```java
// Representation of a lottery machine,
// with the facility for a randomly chosen ball to be ejected.
public class Machine extends BallContainer
{
  // Constructor is given the name and size.
  public Machine(String name, int size)
  {
    super(name, size);
  } // Machine
```

- A **method implementation** for **abstract method** `getType()`.

```java
  // Returns the name of the type of BallContainer.
  public String getType()
  {
    return "Lottery machine";
  } // getType
```

Array: partially filled array: deleting an element

- Simplest way to delete **array element** from **partially filled array** with arbitrary order
  - decrement the count
  - replace unwanted item with one at end.

```
int indexToBeDeleted = ...
noOfItemsInArray--;
anArray[indexToBeDeleted] = anArray[noOfItemsInArray];
```

The Machine class

```java
// Randomly chooses a ball in the machine, and ejects it.
// The ejected ball is returned. If the machine is empty then
// it has no effect, and returns null.
public Ball ejectBall()
{
  if (getNoOfBalls() <= 0)
    return null;
```

The Machine class

```
    else
    {
      // Math.random() * getNoOfBalls yields a number
      // which is >= 0 and < number of balls.
      int ejectedBallIndex = (int) (Math.random() * getNoOfBalls());

      Ball ejectedBall = getBall(ejectedBallIndex);

      swapBalls(ejectedBallIndex, getNoOfBalls() - 1);
      removeBall();

      return ejectedBall;
    } // else
  } // ejectBall

} // class Machine
```

The Rack class

```java
// Representation of a landing rack of balls for the lottery,
// with the facility for them to be sorted into order,
// and another to determine if it contains a ball of a given value.
public class Rack extends BallContainer
{
  // Constructor is given the name and size.
  public Rack(String name, int size)
  {
    super(name, size);
  } // Rack


  // Returns the name of the type of BallContainer.
  public String getType()
  {
    return "Landing rack";
  } // getType
```

The Rack class

```java
// Sorts the balls in the Rack into ascending order,
// using their compareTo() methods.
public void sortBalls()
{
  // Each pass of the sort reduces unsortedLength by one.
  int unsortedLength = getNoOfBalls();
  // If no change is made on a pass, the main loop can stop.
  boolean changedOnThisPass;
  do
  {
    changedOnThisPass = false;
    for (int pairLeftIndex = 0;
             pairLeftIndex < unsortedLength - 1; pairLeftIndex++)
      if (getBall(pairLeftIndex).compareTo(getBall(pairLeftIndex + 1)) > 0)
      {
        swapBalls(pairLeftIndex, pairLeftIndex + 1);
        changedOnThisPass = true;
      } // if
    unsortedLength--;
  } while (changedOnThisPass);
} // sortBalls
```

The Rack class

```
  // Return true if and only if the rack contains
  // a Ball with the given number.
  public boolean contains(int value)
  {
    boolean found = false;
    int index = 0;
    while (!found && index < getNoOfBalls())
    {
      found = getBall(index).getValue() == value;
      index++;
    } // while
    return found;
  } // contains

} // class Rack
```

Section 11

The `Game` class

Aim

AIM: To illustrate the difference between **is a** and **has a** relationships.

The Game class

- Games consist of machine and rack.
- Also create `TestGame` but not show here.
- A `Game` **has a** `Machine` and a `Rack`.
- A `Machine` **is a** `BallContainer`.

Inheritance: is a versus has a

- When **class** A is **subclass** of B
  - **object** of **type** A **is a** B.
- When C has **instance variable** of type D
  - object of type C **has a** D.

The Game class

```java
// Representation of a lottery game, comprising a machine and a rack.
public class Game
{
  // The machine for the game.
  private final Machine machine;

  // The rack for the game.
  private final Rack rack;


  // Constructor takes name and size of the machine, and the rack.
  public Game(String machineName, int machineSize,
              String rackName, int rackSize)
  {
    machine = new Machine(machineName, machineSize);
    rack = new Rack(rackName, rackSize);
  } // Game
```

The Game class

```java
// Return the size of the machine.
public int getMachineSize()
{
  return machine.getSize();
} // getMachineSize


// Return the size of the rack.
public int getRackSize()
{
  return rack.getSize();
} // getRackSize


// Return the number of balls in the rack.
public int getRackNoOfBalls()
{
  return rack.getNoOfBalls();
} // getRackNoOfBalls
```

The Game class

```java
// Add a ball into the machine
public void machineAddBall(Ball ball)
{
  machine.addBall(ball);
} // machineAddBall
```

The Game class

```java
// Eject a ball from the machine into the rack.
// Also return the rejected Ball.
public Ball ejectBall()
{
  if (machine.getNoOfBalls() > 0
      && rack.getNoOfBalls() < rack.getSize())
  {
    Ball ejectedBall = machine.ejectBall();
    rack.addBall(ejectedBall);
    return ejectedBall;
  } // if
  else
    return null;
} // ejectBall
```

The Game class

```
// Returns true if and only if the rack contains
// a Ball with the given number.
public boolean rackContains(int value)
{
  return rack.contains(value);
} // rackContains


// Sorts the balls in the Rack into ascending order.
public void rackSortBalls()
{
  rack.sortBalls();
} // rackSortBalls
```

The Game class

```
  // Mainly for testing.
  public String toString()
  {
    return String.format("%s%n%s", machine, rack);
  } // toString

} // class Game
```

Coursework: Shopping baskets

(Summary only)

Write a **class** each **instance** of which **has a** number of instances of another class stored in it.

Section 12

The Worker classes

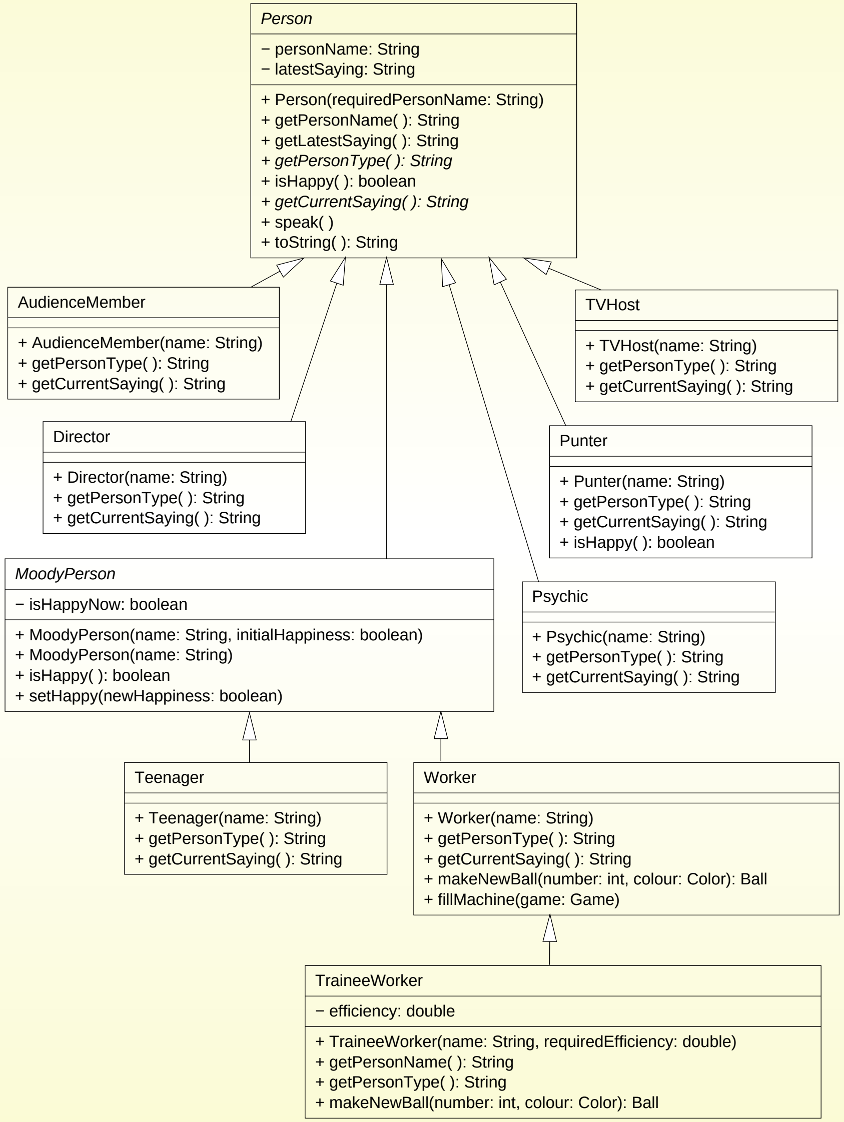
Aim

AIM: To show an example of a **superclass** which is (appropriately) not an **abstract class**. We also show how we can use an **instance method** defined in the superclass, from a **subclass** which **override**s it.

The Worker classes

- A Worker creates balls and fills lottery games.
  - A TraineeWorker is still learning to count
  - has efficiency rating:
    - probability of getting numbers right when creating balls!
- A TraineeWorker **is a** Worker.

The Worker classes



Person
− personName: String
− latestSaying: String
+ Person(requiredPersonName: String)
+ getPersonName( ): String
+ getLatestSaying( ): String
+ *getPersonType( ): String*
+ isHappy( ): boolean
+ *getCurrentSaying( ): String*
+ speak( )
+ toString( ): String

AudienceMember
+ AudienceMember(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

TVHost
+ TVHost(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Director
+ Director(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Punter
+ Punter(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String
+ isHappy( ): boolean

MoodyPerson
− isHappyNow: boolean
+ MoodyPerson(name: String, initialHappiness: boolean)
+ MoodyPerson(name: String)
+ isHappy( ): boolean
+ setHappy(newHappiness: boolean)

Psychic
+ Psychic(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Teenager
+ Teenager(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Worker
+ Worker(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String
+ makeNewBall(number: int, colour: Color): Ball
+ fillMachine(game: Game)

TraineeWorker
− efficiency: double
+ TraineeWorker(name: String, requiredEfficiency: double)
+ getPersonName( ): String
+ getPersonType( ): String
+ makeNewBall(number: int, colour: Color): Ball

The Worker classes

- Testing:
  - Worker and TraineeWorker are **subclass**es of MoodyPerson
    - add instances to TestPersonSubclasses – not shown here.
  - Have TestWorkers – do show that.

The Worker class

```
import java.awt.Color;

// Representation of a worker making balls
// and filling up machines in the lottery.
public class Worker extends MoodyPerson
{
  // Constructor is given the person’s name.
  public Worker(String name)
  {
    super(name);
  }  // Worker
```

The Worker class

```java
// Returns the name of the type of Person.
public String getPersonType()
{
  return "Worker";
} // getPersonType


// Returns the Person’s current saying.
public String getCurrentSaying()
{
  if (isHappy())
    return "Time for tea, I think";
  else
    return "Puff, pant, puff, pant";
} // getCurrentSaying
```

The Worker class

- Worker can fill Game with **new**ly created balls.
- Have separate **instance method** to create single ball
  - so TraineeWorker can **override** it.

```
// Returns a newly created Ball with the given number and colour.
public Ball makeNewBall(int number, Color colour)
{
  return new Ball(number, colour);
} // makeNewBall
```

- Ball colours similar to colours of rainbow
  - approximately evenly spread through balls from 1 to machine size. . . .

The Worker class

```
// Makes this Worker fill the machine of the given Game.
// The Balls are created as they are inserted into the Machine.
public void fillMachine(Game game)
{
  // Colours of balls are evenly spread between these colours,
  // in ascending order.
  Color[] colourGroupColours
    = new Color[] { Color.red, Color.orange, Color.yellow, Color.green,
                    Color.blue, Color.pink, Color.magenta };
  // This happiness change will show up when the GUI is added.
  setHappy(false);
  speak();

```

The Worker class

```java
    int noOfBalls = game.getMachineSize();
    for (int count = 1; count <= noOfBalls; count++)
    {
      // The colour group is a number from 0
      // to the number of colour groups - 1.
      // For the nth ball, we take the fraction
      // (n - 1) divided by the number of balls
      // and multiply that by the number of groups.
      int colourGroup = (int) ((count - 1.0) / (double)noOfBalls
                               * (double) colourGroupColours.length);
      Color ballColour = colourGroupColours[colourGroup];
      game.machineAddBall(makeNewBall(count, ballColour));
    } // for
    setHappy(true);
    speak();
  } // fillMachine

} // class Worker
```

The `TraineeWorker` class

- `TraineeWorker` is **subclass** of `Worker`
  - neither are **abstract class**es.
- `TraineeWorker` has name and efficiency
  - number between 0.0 (never concentrating) and 1.0 (always is).
  - When making ball, if trainee not concentrating
    - ball number is one less or one greater than desired.

The TraineeWorker class

```java
import java.awt.Color;

// Representation of a trainee lottery worker,
// who has an efficiency rating effecting accuracy of ball numbering.
public class TraineeWorker extends Worker
{
  // The efficiency of the TraineeWorker.
  private final double efficiency;


  // Constructor is given the person's name and the required efficiency.
  public TraineeWorker(String name, double requiredEfficiency)
  {
    super(name);
    efficiency = requiredEfficiency;
  } // TraineeWorker
```

The TraineeWorker class

- Want efficiency to be shown as part of person's name.
  - So **override** `getPersonName()`.
  - But need to use overridden version in new one!

Inheritance: using an overridden method

- When **override instance method** in **superclass**, may wish to **method call** *superclass* version in body of **subclass** version.
- Write **reserved word `super`** and dot then instance method name.
- E.g. Assume bicycle emergency stop based on general one.

```
public class Vehicle
{
  ...
  public void emergencyStop()
  {
    ... General code for most vehicles.
  } // emergencyStop
  ...
} // class Vehicle
```

Inheritance: using an overridden method

```
public class Bicycle extends Vehicle
{
  ...
  public void emergencyStop()
  {
    ... Specific code for bicycles.
    super.emergencyStop();
    ... More specific code for bicycles.
  } // emergencyStop
  ...
} // class Bicycle
```

- `super.` can be used in any instance method of subclass
  - not just overriding method.

The TraineeWorker class

```
// Returns the Person's name with the efficiency added in brackets.
public String getPersonName()
{
  return super.getPersonName() + " (" + efficiency + " efficiency)";
} // getPersonName
```

Coffee time: Was `getPersonName()` one of the instance methods which you decided ought to be declared as a **final method** in Section 81 on page 100? Oops?

The TraineeWorker class

```java
// Returns the name of the type of Person.
public String getPersonType()
{
  return "Trainee " + super.getPersonType();
} // getPersonType
```

The TraineeWorker class

```java
  // Returns a newly created Ball with the given number and colour.
  // The ball's number may be wrong depending on the efficiency.
  public Ball makeNewBall(int number, Color colour)
  {
    if (Math.random() >= efficiency)
      if (Math.random() < 0.5)
        number--;
      else
        number++;
    return new Ball(number, colour);
  } // makeNewBall

} // class TraineeWorker
```

The TestWorkers class

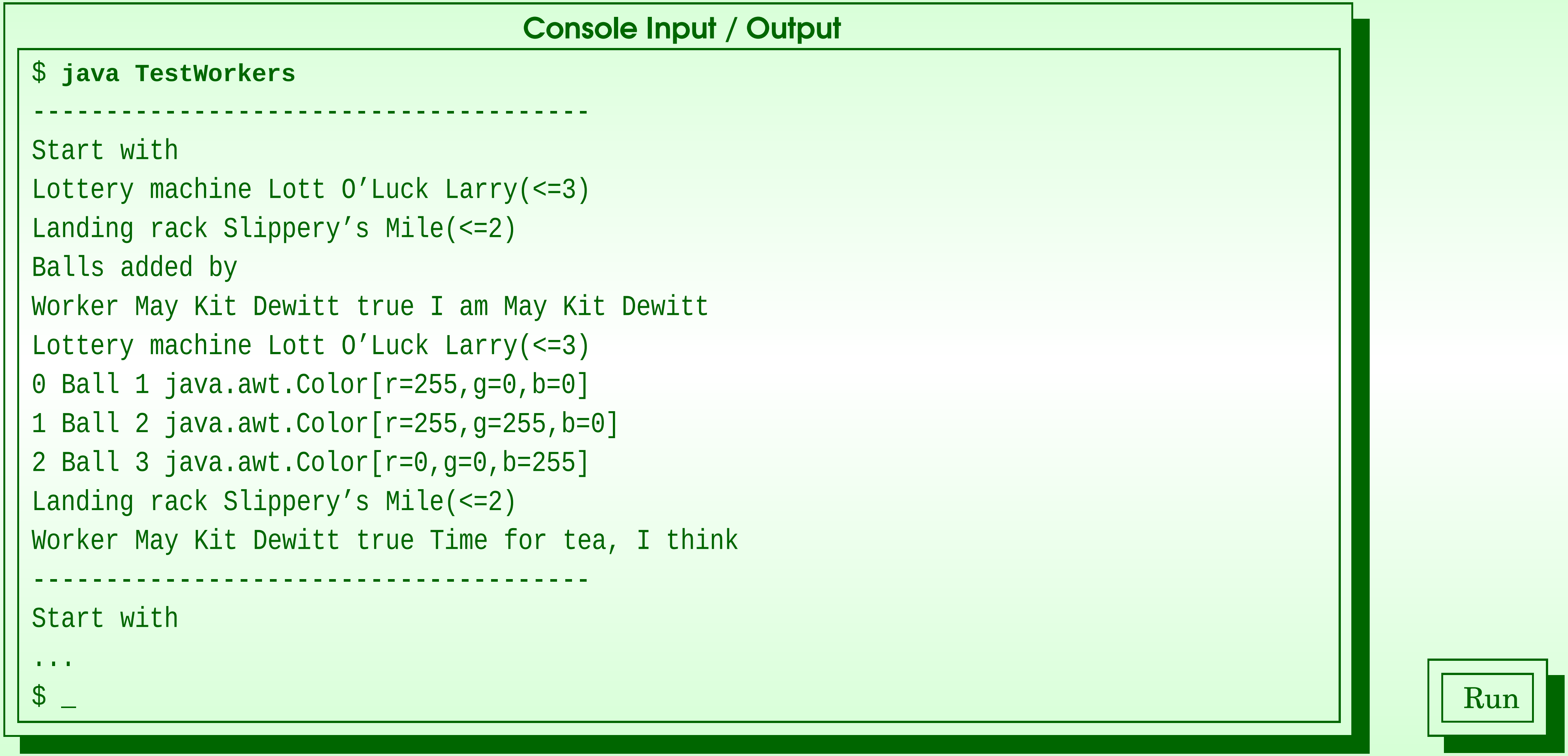
- (Not thorough test.)

```
// Create one of each type of worker,
// and get them to fill the machine of a game.
public class TestWorkers
{
  public static void main(String[] args)
  {
    testWorker(new Worker("May Kit Dewitt"),
               new Game("Lott O’Luck Larry", 3, "Slippery’s Mile", 2));
    testWorker(new TraineeWorker("Darwin Marbest", 0.75),
               new Game("13th Time Lucky", 5, "Oooz OK Lose", 2));
  } // main


```

The TestWorkers class

```
  // Make the given worker fill the given game,
  // reporting values before and after.
  private static void testWorker(Worker worker, Game game)
  {
    System.out.println("--------------------------------------");
    System.out.println("Start with");
    System.out.println(game);

    System.out.println("Balls added by");
    System.out.println(worker);

    worker.fillMachine(game);
    System.out.println(game);
    System.out.println(worker);
  } // testWorker

} // class TestWorkers
```

Trying it



```
$ java TestWorkers
--------------------------------------
Start with
Lottery machine Lott O’Luck Larry(<=3)
Landing rack Slippery’s Mile(<=2)
Balls added by
Worker May Kit Dewitt true I am May Kit Dewitt
Lottery machine Lott O’Luck Larry(<=3)
0 Ball 1 java.awt.Color[r=255,g=0,b=0]
1 Ball 2 java.awt.Color[r=255,g=255,b=0]
2 Ball 3 java.awt.Color[r=0,g=0,b=255]
Landing rack Slippery’s Mile(<=2)
Worker May Kit Dewitt true Time for tea, I think
--------------------------------------
Start with
...
$ _
```

Coursework: Loads of disc space

(Summary only)

To write a non-**abstract class** which has a **subclass**, and use an **instance method** defined in the **superclass** from a subclass which **override**s it.

Section 13

The `CleverPunter` class

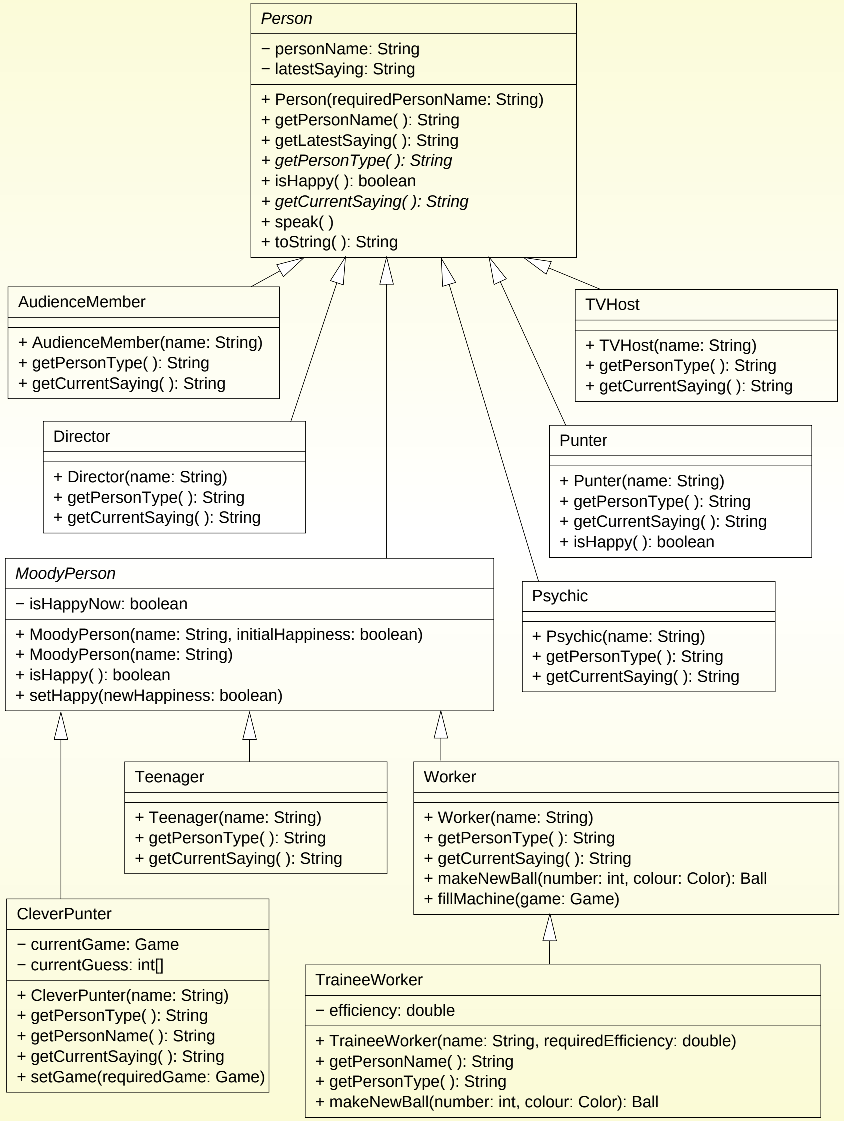
Aim

AIM: To reinforce **inheritance** concepts, and complete the model **class**es of the Notional Lottery program.

The CleverPunter class

- CleverPunter models kind of person that plays lottery games.
  - Is **subclass** of MoodyPerson.
- Develop CleverPunter and TestCleverPunter here.
- Add a CleverPunter to TestPersonSubclasses – not shown here.
- Our final **UML class diagram**. . . .

The CleverPunter class



Person

− personName: String
− latestSaying: String

+ Person(requiredPersonName: String)
+ getPersonName( ): String
+ getLatestSaying( ): String
+ *getPersonType( ): String*
+ isHappy( ): boolean
+ *getCurrentSaying( ): String*
+ speak( )
+ toString( ): String

AudienceMember

+ AudienceMember(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

TVHost

+ TVHost(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Director

+ Director(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Punter

+ Punter(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String
+ isHappy( ): boolean

MoodyPerson

− isHappyNow: boolean

+ MoodyPerson(name: String, initialHappiness: boolean)
+ MoodyPerson(name: String)
+ isHappy( ): boolean
+ setHappy(newHappiness: boolean)

Psychic

+ Psychic(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Teenager

+ Teenager(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String

Worker

+ Worker(name: String)
+ getPersonType( ): String
+ getCurrentSaying( ): String
+ makeNewBall(number: int, colour: Color): Ball
+ fillMachine(game: Game)

CleverPunter

− currentGame: Game
− currentGuess: int[]

+ CleverPunter(name: String)
+ getPersonType( ): String
+ getPersonName( ): String
+ getCurrentSaying( ): String
+ setGame(requiredGame: Game)

TraineeWorker

− efficiency: double

+ TraineeWorker(name: String, requiredEfficiency: double)
+ getPersonName( ): String
+ getPersonType( ): String
+ makeNewBall(number: int, colour: Color): Ball

The CleverPunter class

```java
// Representation of a clever person playing the lottery who actually knows
// enough to make some guesses and score them against a game.
public class CleverPunter extends MoodyPerson
{
  // The game which is currently being played.
  private Game currentGame = null;

  // The guess of what balls will come out.
  private int[] currentGuess = null;


  // Constructor is given the person’s name.
  public CleverPunter(String name)
  {
    super(name);
  }  // CleverPunter
```

The CleverPunter class

```java
// Returns the name of the type of Person.
public String getPersonType()
{
  return "Clever Punter";
} // getPersonType


// Returns the Person’s name, with the current guess included.
public String getPersonName()
{
  String result = super.getPersonName();
  if (currentGuess != null && currentGuess.length != 0)
  {
    result += "(guess " + currentGuess[0];
    for (int index = 1; index < currentGuess.length; index++)
      result += "," + currentGuess[index];
    result += ")";
  } // if
  return result;
} // getPersonName
```

The CleverPunter class

```java
// Returns the Person’s current saying.
public String getCurrentSaying()
{
  if (currentGame == null)
  {
    setHappy(false);
    return "I need a game to play!";
  } // if
  else
```

The CleverPunter class

```
    {
      int noOfMatches = getNoOfMatches();
      int noOfNonMatches = currentGame.getRackNoOfBalls() - noOfMatches;
      // Is happy if and only if there are no non-matches.
      setHappy(noOfNonMatches == 0);
      if (noOfMatches == currentGame.getRackSize())
        return "Yippee!! I've won the jackpot!";
      else if (noOfNonMatches != 0)
        return "Doh! " + noOfNonMatches + " not matched";
      else if (noOfMatches == 0) // I.e. the rack is still empty.
        return "I'm excited!";
      else
        return noOfMatches + " matched so far!";
    } // else
  } // getCurrentSaying
```

The `CleverPunter` class

```java


// Helper method to find out how many of the guesses currently match the
// game rack. Note: this does not get called if currentGuess is null.
private int getNoOfMatches()
{
  int noMatchedSoFar = 0;
  for (int oneNumber : currentGuess)
    if (currentGame.rackContains(oneNumber))
      noMatchedSoFar++;
  return noMatchedSoFar;
} // getNoOfMatches
```

- Next, observe **software reuse** – play a mock game to get the guess!. . .

The CleverPunter class

```java
  // Set the game being currently played.
  public void setGame(Game requiredGame)
  {
    currentGame = requiredGame;
    currentGuess = new int[currentGame.getRackSize()];
    // An easy way to obtain a guess is to play a mock game!
    Game mockGame = new Game("", currentGame.getMachineSize(),
                             "", currentGame.getRackSize());
    Worker mockWorker = new Worker("");
    mockWorker.fillMachine(mockGame);
    for (int index = 0; index < currentGame.getRackSize(); index++)
      currentGuess[index] = mockGame.ejectBall().getValue();
  } // setGame

} // class CleverPunter
```

The `CleverPunter` class

Coffee time: Whilst that may have been a bit 'clever' of us, was it really the best way to have software reuse for sharing code between `CleverPunter` and `Machine`? (Hint: some kind of number chooser?)

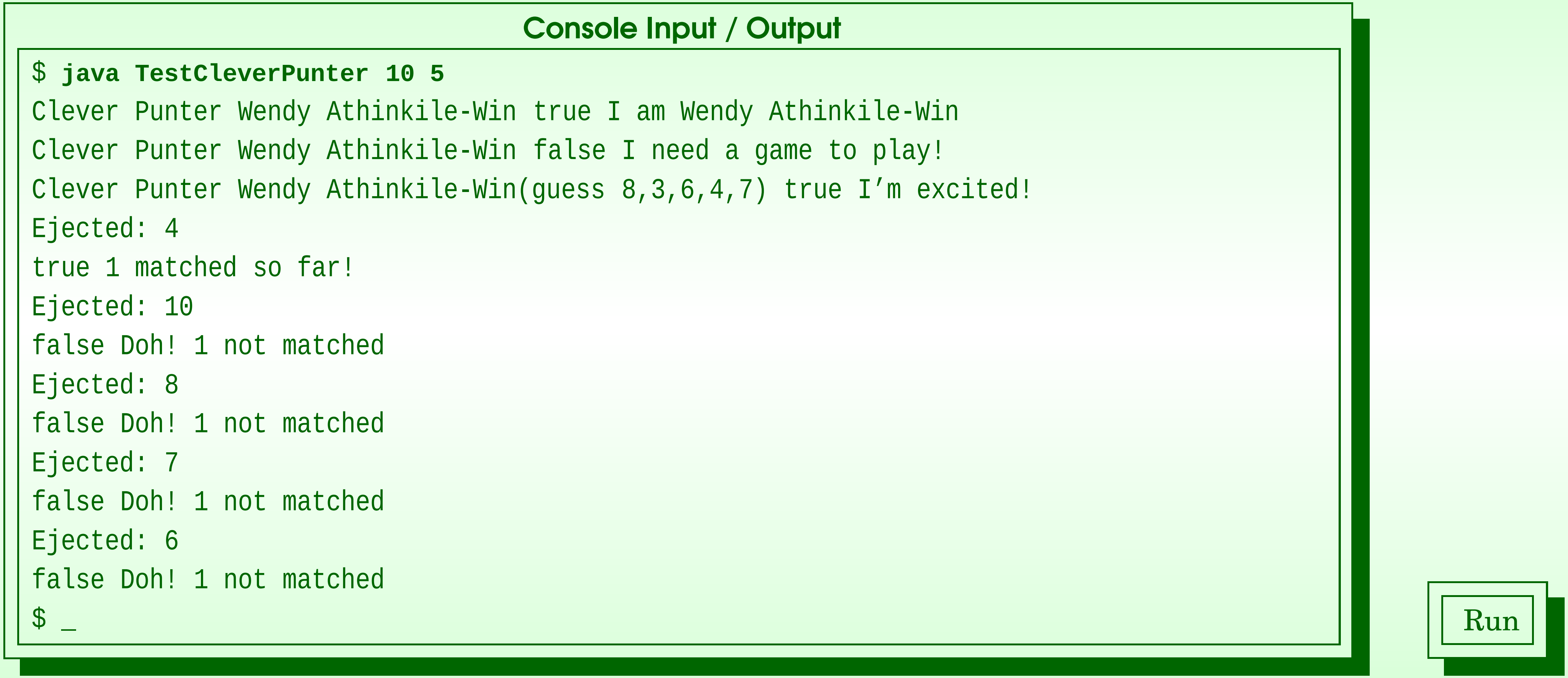
The TestCleverPunter class

```java
// Given a machine size and a rack size from the first two arguments,
// create a game and a clever punter to play it,
// reporting result as eject each ball.
public class TestCleverPunter
{
  public static void main(String[] args)
  {
    int machineSize = Integer.parseInt(args[0]);
    int rackSize = Integer.parseInt(args[1]);

    Game game = new Game("Lott O'Luck Larry", machineSize,
                         "Slippery's Mile", rackSize);
    Worker worker = new Worker("May Kit Dewitt");
    worker.fillMachine(game);

```

The TestCleverPunter class

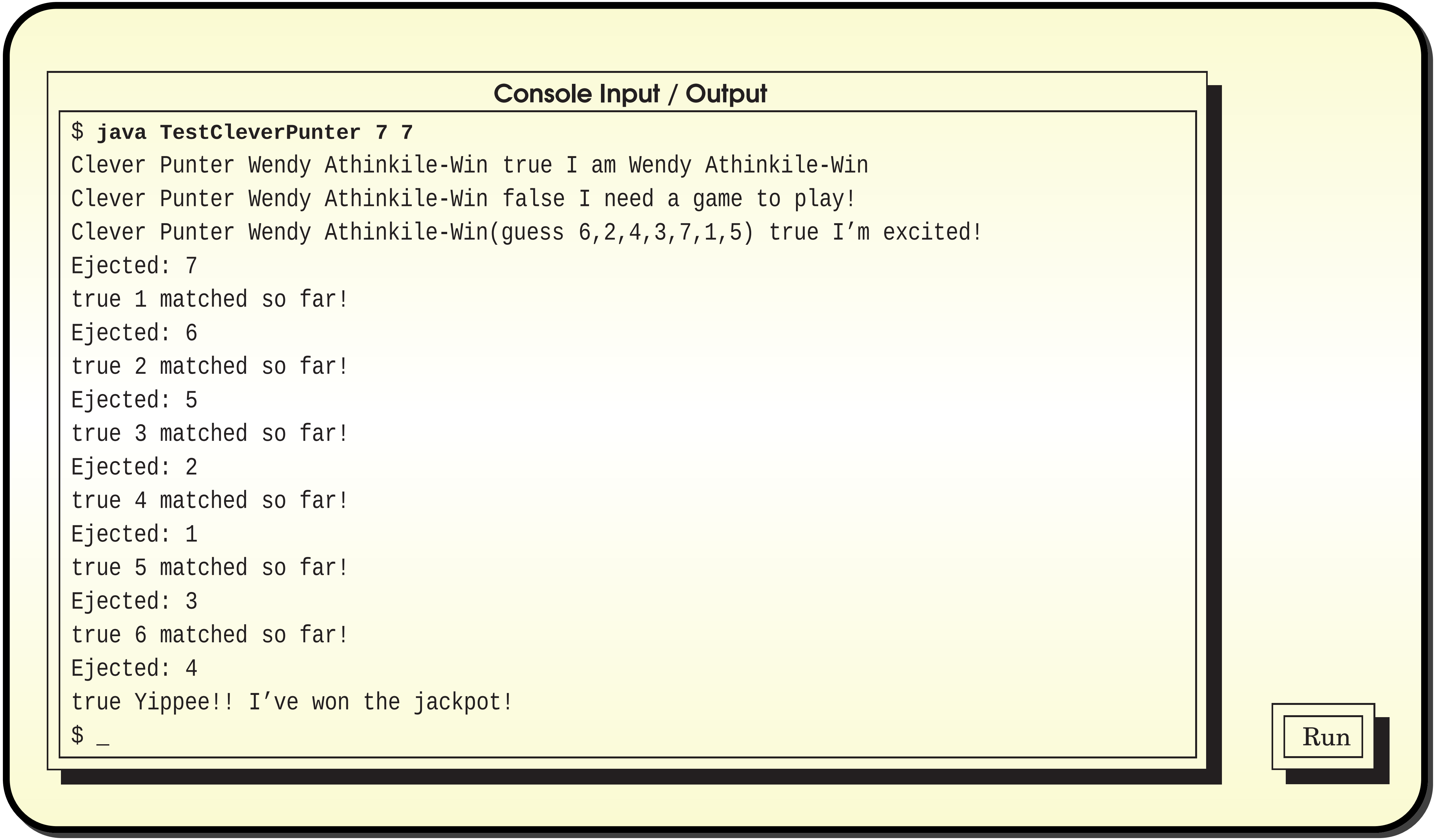
```java
    CleverPunter cleverPunter = new CleverPunter("Wendy Athinkile-Win");
    System.out.println(cleverPunter);
    cleverPunter.speak();
    System.out.println(cleverPunter);

    cleverPunter.setGame(game);
    cleverPunter.speak();
    System.out.println(cleverPunter);
    for (int count = 1; count <= game.getRackSize(); count++)
    {
      System.out.println("Ejected: " + game.ejectBall().getValue());
      cleverPunter.speak();
      System.out.println(cleverPunter.isHappy()
                         + " " + cleverPunter.getLatestSaying());
    } // for
  } // main

} // class TestCleverPunter
```

Trying it



```
$ java TestCleverPunter 10 5
Clever Punter Wendy Athinkile-Win true I am Wendy Athinkile-Win
Clever Punter Wendy Athinkile-Win false I need a game to play!
Clever Punter Wendy Athinkile-Win(guess 8,3,6,4,7) true I’m excited!
Ejected: 4
true 1 matched so far!
Ejected: 10
false Doh! 1 not matched
Ejected: 8
false Doh! 1 not matched
Ejected: 7
false Doh! 1 not matched
Ejected: 6
false Doh! 1 not matched
$ _
```

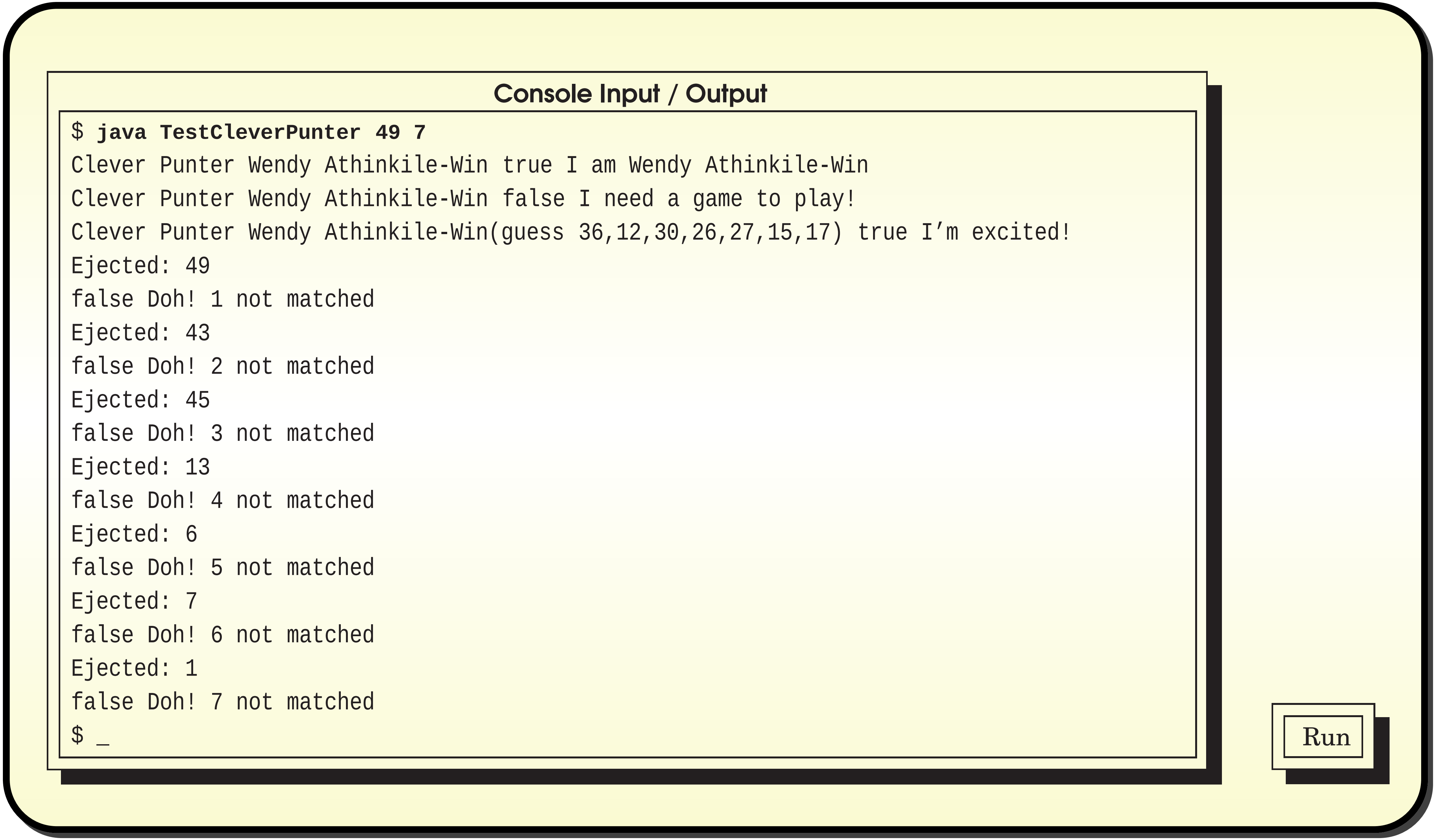
Trying it



Console Input / Output

```
$ java TestCleverPunter 7 7
Clever Punter Wendy Athinkile-Win true I am Wendy Athinkile-Win
Clever Punter Wendy Athinkile-Win false I need a game to play!
Clever Punter Wendy Athinkile-Win(guess 6,2,4,3,7,1,5) true I’m excited!
Ejected: 7
true 1 matched so far!
Ejected: 6
true 2 matched so far!
Ejected: 5
true 3 matched so far!
Ejected: 2
true 4 matched so far!
Ejected: 1
true 5 matched so far!
Ejected: 3
true 6 matched so far!
Ejected: 4
true Yippee!! I’ve won the jackpot!
$ _
```

Trying it



```
$ java TestCleverPunter 49 7
Clever Punter Wendy Athinkile-Win true I am Wendy Athinkile-Win
Clever Punter Wendy Athinkile-Win false I need a game to play!
Clever Punter Wendy Athinkile-Win(guess 36,12,30,26,27,15,17) true I’m excited!
Ejected: 49
false Doh! 1 not matched
Ejected: 43
false Doh! 2 not matched
Ejected: 45
false Doh! 3 not matched
Ejected: 13
false Doh! 4 not matched
Ejected: 6
false Doh! 5 not matched
Ejected: 7
false Doh! 6 not matched
Ejected: 1
false Doh! 7 not matched
$ _
```

Coursework: Making it more realistic

(Summary only)

Add more complexity to an **inheritance hierarchy** at appropriate places.

Section 14

The GUI classes

Aim

AIM: To characterize the rest of the Notional Lottery program development.

The GUI classes

- Second phase concerns **graphical user interface classes**
  - details would be distraction, so merely characterize.
- Class `LotteryGUI` to provide graphical user interface.
- Classes to provide images for model objects
  - `PersonImage`, `BallImage`,
  - and `BallContainerImage`
    - with **subclasses** `MachineImage` and `RackImage`.
- `Person` modified
  so each **instance has a** corresponding instance of `PersonImage`
  - created by **constructor method** of `Person`
  - stored in new **instance variable**.

The GUI classes

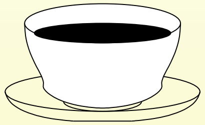
- Similarly `Ball`, `Machine` and `Rack` **objects**
  - each have corresponding `BallImage`, `MachineImage` and `RackImage` object.
- `PersonImage` has `update()` **instance method**
  - ensures image on screen reflects state of `Person`
  - `Person` modified to invoke `update()` whenever state changes.
- E.g. `MoodyPerson setHappy()`:

MoodyPerson.java

```
...
// Sets the happiness of the person to the given state.
public void setHappy(boolean newHappiness)
{
  isHappyNow = newHappiness;
  getImage().update();
} // setHappy
...
```

The GUI classes

- Similar relationship for other model classes with corresponding image class.
- Have classes `SpeedController` and `SpeedControllerGUI` to control speed of game.
- `Person` and `Ball` have `flash()` instance method
  - causes their image objects to flash on screen
  - invoked at various points in model
    - e.g. just before ball is ejected from machine.
- Each kind of `Person` has different coloured face in image
  - `getColour()` instance method added to `Person`.



Coffee time: How would we add `getColour()` to the `Person` model classes, so that each type of person has a different colour?

Section 15

The `Object` class and constructor chaining

Aim

AIM: To introduce the **class** `Object` and the fact that the **constructor method** of the **superclass** is invoked implicitly by default. We also take a more thorough look at **constructor chaining**.

Standard API: `Object`

- All **object**s are also **instance**s of `java.lang.Object`.
- If class not declared to **extend** some other class
  - implicitly extends `Object` directly.
- ALL classes reside in single **inheritance hierarchy**
  - `Object` at root.
- Every class has one **superclass**
  - except `Object`.

Standard API: `Object`

- `Object` has one **constructor method**.

```
public class Object
{
  ...
  public Object()
  {
    ... Code here to actually create an object,
    ... allocating memory for it, etc..
  } // Object
  ...
} // class Object
```

Inheritance: invoking the superclass constructor: implicitly

- In **constructor method**, if first **statement**
  - is not **superclass constructor call**
  - nor **alternative constructor call**
  - then implicit call `super()` assumed.
- The first work done by constructor must be to actually create the **object**
  - allocate memory for it
  - done inside constructor of `java.lang.Object`.

The `Object` class and constructor chaining

- E.g. `Person` constructor we saw previously.

Person.java

```
...
public Person(String requiredPersonName)
{
  personName = requiredPersonName;
  latestSaying = "I am " + personName;
}  // Person
...
```

The Object class and constructor chaining

- Treated as though has call to constructor of superclass of `Person`
  - which is `Object`.

Person.java-WITH-SUPER

```
...
public Person(String requiredPersonName)
{
  super();
  personName = requiredPersonName;
  latestSaying = "I am " + personName;
}  // Person
...
```

Inheritance: constructor chaining

- When **constructor method** invoked, first thing is
  - either call to another constructor in same **class**
  - or call to constructor method in **superclass**.
- This does the same
  - all way up **inheritance hierarchy**
  - until constructor of `java.lang.Object` is called.
- Known as **constructor chaining**.
- Constructor chaining must always be possible for every class
  - else could not have **object**s created at **run time**
    - constructor method of `Object` actually creates object.
- One rule
  - at least one constructor must *not* call another of same class!

The `Object` class and constructor chaining

- E.g. see `TraineeWorker` being created.

```
Person person = new TraineeWorker("Justin de Neaushob", 0.0);
```

TraineeWorker.java

```
...
public TraineeWorker(String name, double requiredEfficiency)
{
  super(name);
  efficiency = requiredEfficiency;
}  // TraineeWorker
...
```

Worker.java

```java
...
  public Worker(String name)
  {
    super(name);
  }  // Worker
...
```

MoodyPerson.java

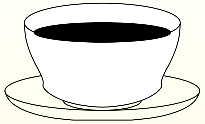
```java
...
public MoodyPerson(String name, boolean initialHappiness)
{
  super(name);
  isHappyNow = initialHappiness;
} // MoodyPerson
...
public MoodyPerson(String name)
{
  this(name, true);
} // MoodyPerson
...
```

Person.java

```java
...
public Person(String requiredPersonName)
{
  personName = requiredPersonName;
  latestSaying = "I am " + personName;
}  // Person
...
```

The `Object` class and constructor chaining

- And finally that implicitly calls constructor of `Object`.



Coffee time: Suppose a (non-abstract) class does not have a constructor method defined by the programmer. Can it still be instantiated? How does this fit in with constructor chaining?

Method: constructor methods: default

- If **class** does not include **constructor method**
  - Java assumes **default constructor**
  - **public** empty one, no **method argument**s.
- E.g. for class called `FabulousThing`.

```
public FabulousThing()
{
} // FabulousThing
```

- which is same as:

```
public FabulousThing()
{
  super();
} // FabulousThing
```

Method: constructor methods: default

- Default constructor only assumed for classes with no explicitly defined constructor
  - so not every class has constructor method with no arguments.
- E.g. `VeryFabulousThing` does not.

```
public class VeryFabulousThing
{
  ... Some code, but no more constructor methods.
  public VeryFabulousThing(String name)
  {
    ...
  } // VeryFabulousThing
  ... Some code, but no more constructor methods.
} // class VeryFabulousThing
```

Method: constructor methods: default

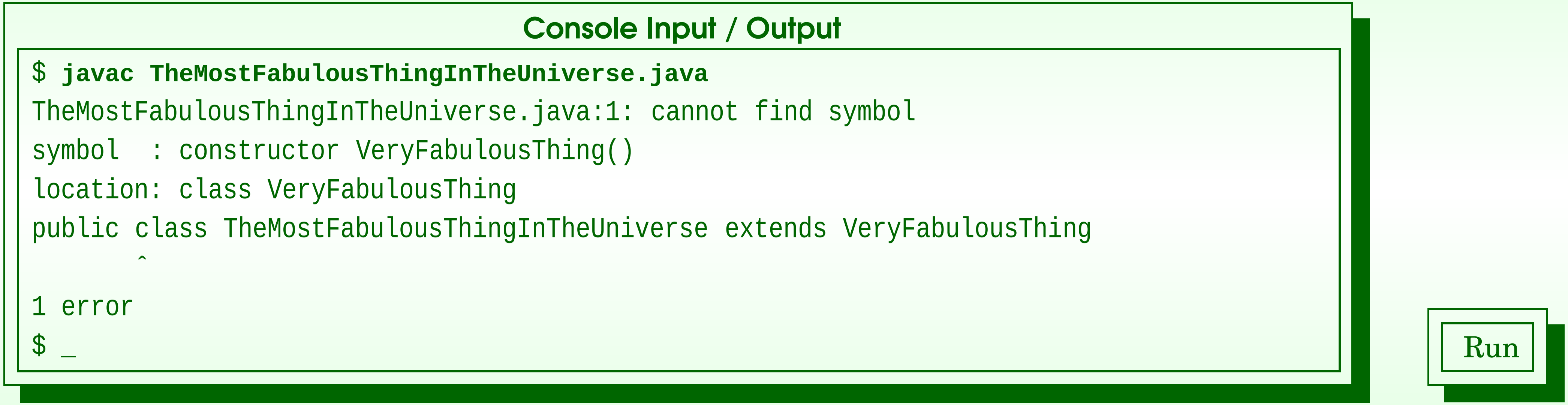
- So this is illegal!

```
public class TheMostFabulousThingInTheUniverse extends VeryFabulousThing
{
  ... Code here, but no constructor method.
} // class TheMostFabulousThingInTheUniverse
```

Coffee time: Why?

- Default constructors not often what we want.
- Recommend: *always* explicitly write at least one constructor in classes intended to have **instance**s
  - even when that constructor is empty
  - shows that is deliberately empty rather than been omitted.

Trying it



```
$ javac TheMostFabulousThingInTheUniverse.java
TheMostFabulousThingInTheUniverse.java:1: cannot find symbol
symbol  : constructor VeryFabulousThing()
location: class VeryFabulousThing
public class TheMostFabulousThingInTheUniverse extends VeryFabulousThing
       ^
1 error
$ _
```

Coursework: Exploring constructor chaining

(Summary only)

Add tracing to existing **constructor method**s in order to explore **constructor chaining**.

Section 16

Overloaded methods versus override

Aim

AIM: To take a closer look at **overloaded methods** and in particular how an intended **override** can accidentally become an overload. We revisit the overloaded methods `System.out.println()`, and look at `toString()` from the `Object` **class**.

Overloaded methods versus override

- Can have **overloaded methods**
  - more than one **method** with same name in same **class**
  - including those **inherit**ed from a **superclass**.
- Can be confused with **instance methods** that **override** another.

Does an `int` match a `double`?

```java
public class WhoAmI
{
  public static void identify(int arg)
  {
    System.out.println("I am an int: " + arg);
  } // identify

  public static void identify(double arg)
  {
    System.out.println("I am a double: " + arg);
  } // identify

```

Does an int match a double?

```
  public static void identifyToo(double arg)
  {
    System.out.println("I too am a double: " + arg);
  } // identifyToo

  public static void main(String[] args)
  {
    identify(10);       // An int argument is surely an int.
    identify(20.0);     // A double argument is surely a double.
    identifyToo(30);    // An int argument is surely an int.
  } // main

} // class WhoAmI
```

Does an `int` match a `double`?

Console Input / Output

```
$ java WhoAmI
I am an int: 10
I am a double: 20.0
I too am a double: 30.0
$ _
```

Run

- First **method call**, **method argument** is **`int`**
  - two **method**s match
  - **compiler** picks most specific one.
- Second method call, one method matches
  - **`int`** method argument matches **`double`** **method parameter**,
    - ∗ but not vice-versa.
- Third method call, one method matches
  - **`int`** method argument automatically **cast** to **`double`**.

Does an `int` match a `double`?

Coffee time: What would happen if we had the following?

```
public static void m(int i, double d) { ... }
public static void m(double d, int i) { ... }
  ... m(10, 10);
```

Standard API: `System`: `out.println()`: with any argument

- `java.lang.System` has **overloaded methods** `out.println()` & `out.print()` for
  - every **primitive type** of **method argument**
  - and `java.lang.Object`.
- Each treats argument, `(arg)`, as `("" + arg)`
  - **`int`** output in decimal
  - non-null **object reference** has `toString()` used
  - etc..
- Another version of `System.out.println()` / `System.out.print()` takes a **character array** and print the characters in it.

Standard API: `Object: toString()`

- `java.lang.Object` has `toString()` **instance method**
  - `String` representation of **type** of **object** followed by `'@'` and **hexadecimal** number (hash code).
- Classes which do not provide own version **inherit** this default one.

`System.out.println()` and inheritance

- Previously said **array**s are objects
  - **superclass** of every array **type**: `java.lang.Object`.
  - So **inherit** default `toString()`.
- What is result of following?. . .

System.out.println() and inheritance

```java
public class PrintlnOverloadDemo
{
  private static char[] vowels = {’a’, ’e’, ’i’, ’o’, ’u’};

  public static void main(String[] args)
  {
    System.out.println("Printing vowels as a char[]");
    System.out.println(vowels);
    System.out.println();
    System.out.println("Printing vowels as an Object");
    System.out.println((Object)vowels);
  } // main

} // class PrintlnOverloadDemo
```

System.out.println() and inheritance

- Two versions of System.out.println() match first call
  - one takes a char[], one takes an Object
  - **compiler** chooses most specific
    - ∗ so vowels are printed as string of **character**s.
- For second call, **cast** tells compiler to treat array as Object
  - so get version of System.out.println() that takes an Object
  - uses toString() of array – inherited from Object.

Console Input / Output

```
$ java PrintlnOverloadDemo
Printing vowels as a char[]
aeiou

Printing vowels as an Object
[C@1a46e30
$ _
```

Run

Accidental overload

- The **compiler** produces **byte code** to call a **method** with particular **method interface**
  - based on **type**s of **method argument**s.
- Where there is choice of matching methods
  - chooses most specific one
  - decision made at **compile time**.
- Then **dynamic method binding** chooses correct **method implementation** at **run time**.
- Common error: intended **override** results in **overloaded method**.
- E.g. contrived example: police inspectors
  - interrogating other police inspectors. . .

Accidental overload

```java
public class Inspector
{
  private final String name;

  public Inspector(String requiredName)
  {
    name = requiredName;
  } // Inspector

  public String getName()
  {
    return name;
  } // getName

```

Accidental overload

```java
public void interrogate(Inspector suspect)
{
  System.out.println("I am Inspector " + getName()
                     + ", who are you? " + suspect);
} // interrogate

public String toString()
{
  return "I am Inspector " + getName() + "!";
} // toString
```

Accidental overload

- A **class method** to arrange interrogation.

```
  public static void makeInspection(Inspector inspectingOfficer,
                                    Inspector suspect)
  {
    inspectingOfficer.interrogate(suspect);
  } // makeInspection

} // class Inspector
```

Overloaded methods versus override

```java
public class ChiefInspector extends Inspector
{
  public ChiefInspector(String name)
  {
    super(name);
  } // ChiefInspector

  public void interrogate(ChiefInspector suspect)
  {
    System.out.println("I am Chief Inspector " + getName()
                       + ", who are you? " + suspect);
  } // interrogate

  public String toString()
  {
    return "I am Chief Inspector " + getName() + "!";
  } // toString
```

Overloaded methods versus override

```java
  public static void main(String[] args)
  {
    Inspector clouseau = new Inspector("Clouseau");
    ChiefInspector dreyfus = new ChiefInspector("Dreyfus");

    Inspector.makeInspection(clouseau, dreyfus);
    Inspector.makeInspection(dreyfus, clouseau);
    Inspector.makeInspection(dreyfus, dreyfus);
    System.out.println();
    clouseau.interrogate(dreyfus);
    dreyfus.interrogate(clouseau);
    dreyfus.interrogate(dreyfus);
  } // main

} // class ChiefInspector
```

Overloaded methods versus override

Coffee time: Before reading on, predict what the output will be. In particular, do you expect the results of the first three interrogations to be the same as the second three?

Overloaded methods versus override

Console Input / Output

```
$ java ChiefInspector
I am Inspector Clouseau, who are you? I am Chief Inspector Dreyfus!
I am Inspector Dreyfus, who are you? I am Inspector Clouseau!
I am Inspector Dreyfus, who are you? I am Chief Inspector Dreyfus!

I am Inspector Clouseau, who are you? I am Chief Inspector Dreyfus!
I am Inspector Dreyfus, who are you? I am Inspector Clouseau!
I am Chief Inspector Dreyfus, who are you? I am Chief Inspector Dreyfus!
$ _
```

Run

- In some outputs Chief Inspector Dreyfus is wrongly titled Inspector.
- Look carefully at `ChiefInspector` code
  - instance method intended to **override**
  - instead is **overloaded method**.

Inheritance: overriding a method: @Override annotation

- Since Java 5.0 – **annotation**s
  - allow us to provide additional information to **compiler**.
- The **override annotation**, `@Override`
  - written immediately before **instance method** heading
  - says we believe **override**s one from **superclass**,
  - or is **method implementation** of **abstract method** in superclass.
- Compiler will complain if not true
  - protecting us from getting **method signature** wrong
    - ∗ misspelling method name
    - ∗ or differently ordering **method parameter type**s.

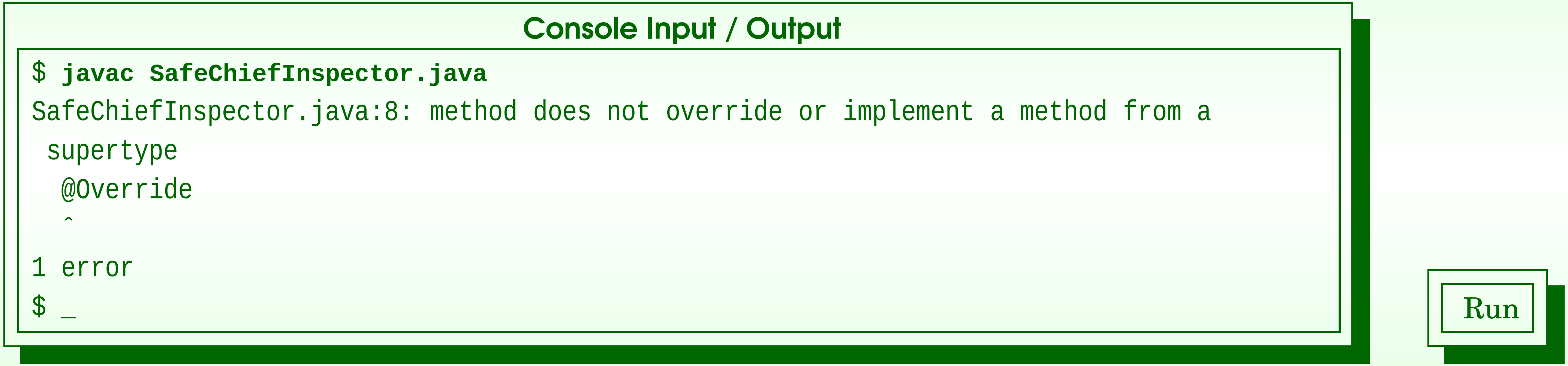
Overloaded methods versus override

- Copy of `ChiefInspector`
  - called `SafeChiefInspector`
  - and has **override annotation**.

SafeChiefInspector.java-WITH-HIDE

```java
public class SafeChiefInspector extends Inspector
{
...
  @Override
  public void interrogate(SafeChiefInspector suspect)
  {
    System.out.println("I am Chief Inspector " + getName()
                       + ", who are you? " + suspect);
  } // interrogate
...
```

Overloaded methods versus override



Console Input / Output

```
$ javac SafeChiefInspector.java
SafeChiefInspector.java:8: method does not override or implement a method from a
 supertype
  @Override
  ^
1 error
$ _
```

Coursework: Using the `@Override` annotation

(Summary only)

Add to your **instance methods** that **override** another, an **annotation** which helps protect against errors.

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.