

Review of fundamental object-oriented concepts

Lecture slides

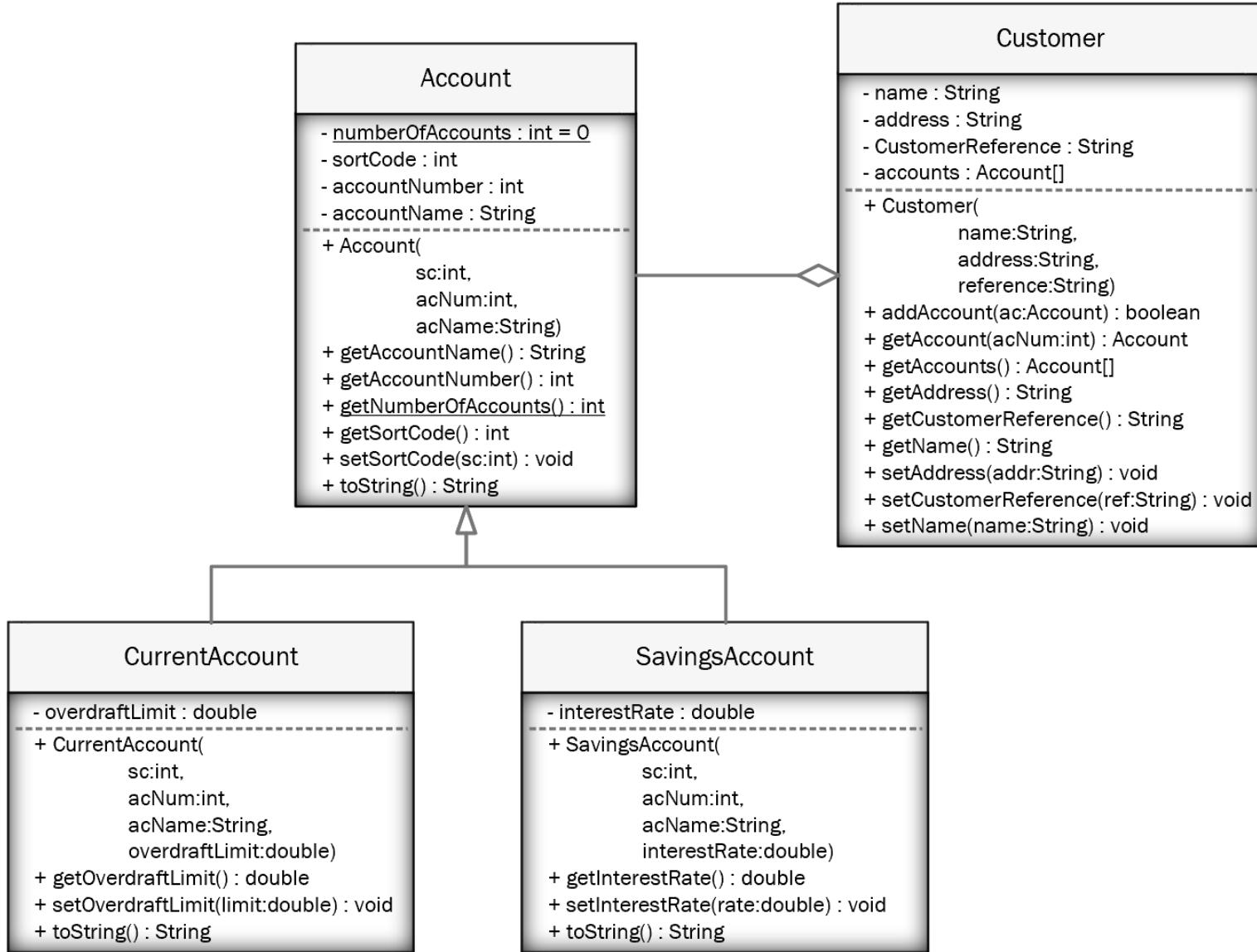
In this lecture

- You should have read the lecture notes before coming to this lecture
- We will:
 - Write a Java program that demonstrates the concepts in the lecture notes
 - Classes with static and non-static variables and methods
 - Association and aggregation
 - Inheritance and polymorphism
- Ask questions about anything you don't understand

Problem

- A bank has two types of account:
 - Current account
 - Savings account
- A customer can have many accounts at the bank
- Using NetBeans, write a Java application that...
 - Implements the UML class diagram on the next page
 - Has a main() method that creates two customers, each with two bank accounts
 - Displays the customer and account details

UML class diagram



Review of fundamental object-oriented concepts

Lecture notes

In these lecture notes

We will:

- Review fundamental concepts of object-orientation
 - Encapsulation
 - Static and non-static fields and methods
 - Association, aggregation and composition
 - Inheritance and polymorphism
- Review how to design and write OO classes

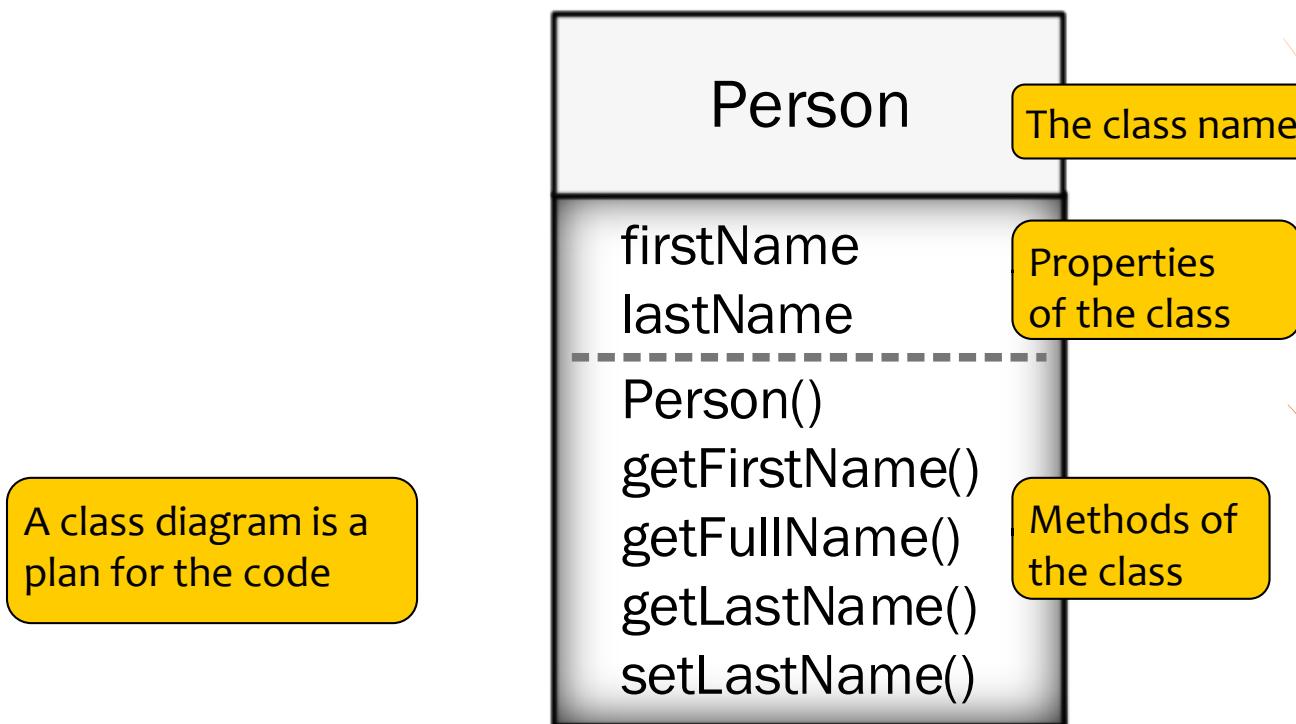
OO building blocks

- **Classes** define groups of **objects**
 - Objects and their interactions feature in everyday life
 - For example: kettle, cup, person, bank account
- Objects in the same class have the same encapsulation...
 - Data (variables), but with different values
 - Actions (methods)
 - Responsibilities (manipulate data; input-output)

Modelling with the UML

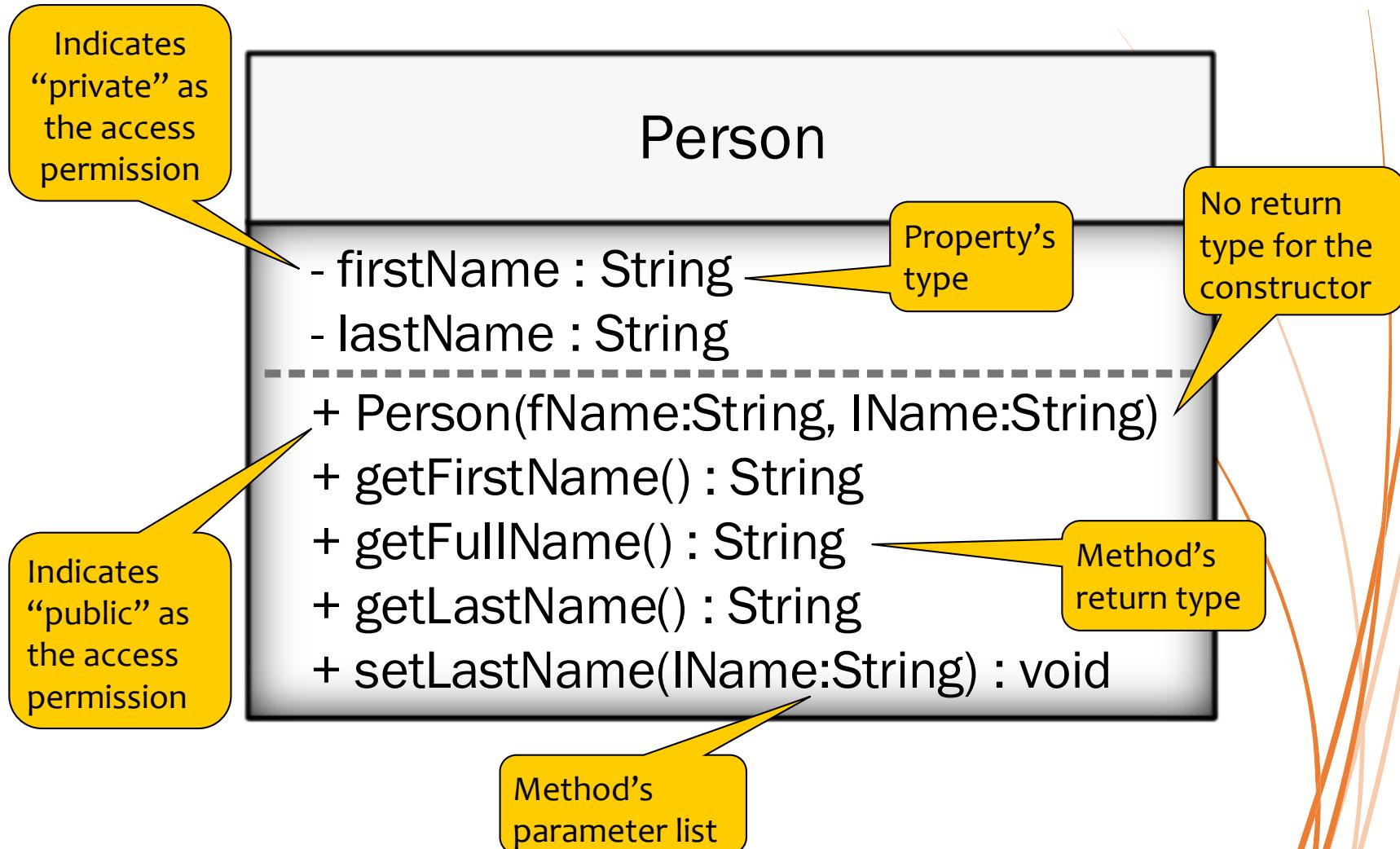
- The Unified Modelling Language (UML)
 - Uses diagrams for giving good overviews
 - Can use textual descriptions for precision
- UML has...
 - Class diagrams
 - Sequence diagrams
 - Many more diagrams
- Use UML to model...
 - Structure (static features)
 - Classes, relationships, responsibilities
 - Behaviour (dynamic features)
 - Interactions, collaborations, flow of control

UML class diagrams



- This tells us something about the class, but not enough to write the program code
 - Data types of the properties?
 - Parameters and return types for the methods?

UML class diagrams



Writing code from UML class diagrams

- There are four easy steps to convert a UML class diagram to code

1. Declare the class

- Copy the class name from the UML diagram

2. Declare the properties

- Copy each property's access permission, type and name from the UML diagram

3. Declare the methods

- Copy each method's signature from the UML diagram

The method's header e.g...
`public int getNum()`

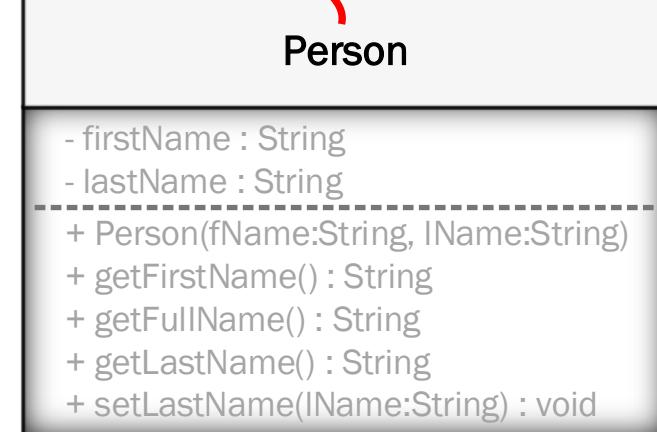
4. Write the code for the method bodies

- Use pseudo-code where provided

Writing code from UML class diagrams

- Declare the class

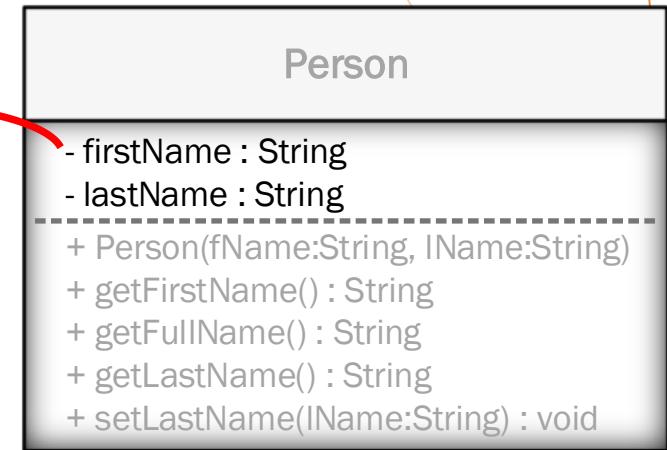
```
public class Person  
{  
}
```



Writing code from UML class diagrams

- Declare the properties

```
public class Person
{
    private String firstName;
    private String lastName;
}
```



Writing code from UML class diagrams

- Declare the methods

```
public class Person
{
    private String firstName;
    private String lastName;

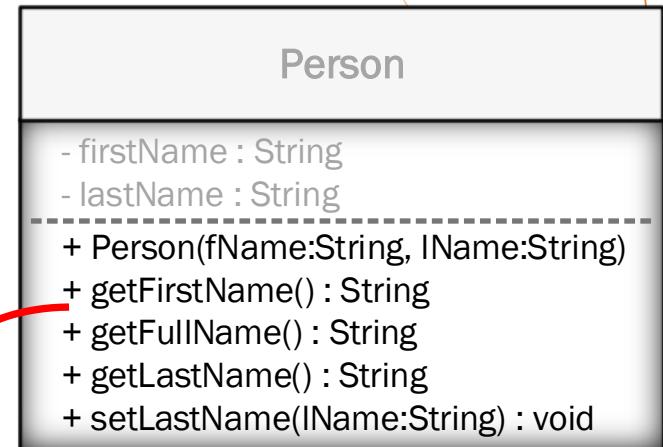
    public Person(String fName,
                  String lName)
    {
    }

    public String getFirstName()
    {
    }

    public String getLastName()
    {
    }

    public void setLastName(String lName)
    {
    }

    public String getFullName()
    {
    }
}
```



Writing code from UML class diagrams

- Write the method bodies
 - Understand **what** the method must do
 - Work out **how** to do it
 - Follow the pseudo-code if available
- Each method can access...
 - Its parameters
 - Any variables declared locally in the method
 - The object's properties

```
public class Person
{
    private String firstName;
    private String lastName;

    public Person(String fName,
                  String lName)
    {
        firstName = fName;
        lastName = lName;
    }
    // rest of code not shown
}
```

**this,
static
and
non-static**

Lecture notes



Java's reserved word `this`

- Java's reserved word `this` means this object or the current object
- When a method has a parameter with the same name as the object's variable, use `this` to distinguish between them

```
public Account(int sortcode,  
              int accountNumber,  
              String accountName)  
{  
    this.sortcode = sortcode;  
    this.accountNumber = accountNum;  
    this.accountName = accountName;  
}
```

The `sortcode` variable of this object

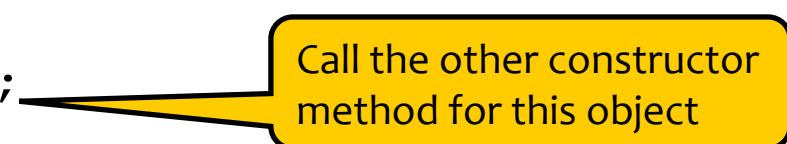
The `sortcode` parameter

Java's reserved word `this`

- `this` can be used in a constructor method to call an overloaded constructor for the current object

```
public class Account
{
    public Account()
    {
        this(-1, -1, "???" );
    }

    public Account(int sc, int acNum, String acName)
    {
        sortcode = sc;
        accountNumber = acNum;
        accountName = acName;
    }
}
```

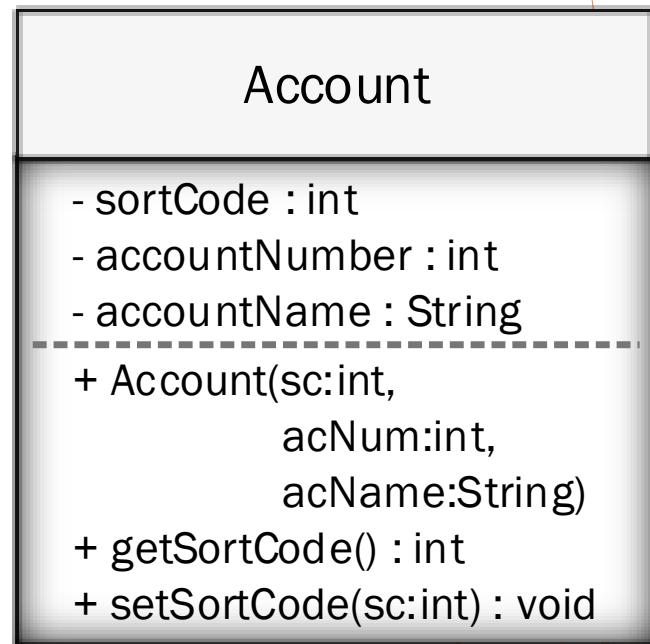
 Call the other constructor method for this object

 (Attribute declarations not shown)



Attributes that belong to objects

- So far, all attributes and methods have "belonged" to a specific object
 - store data for a specific object
 - modify the variables belonging to a specific object



- When an object is created...
 - It has its own version of all variables and methods
 - e.g. every `Account` object will have its own sort code, account number, and account name

Static data and behaviour

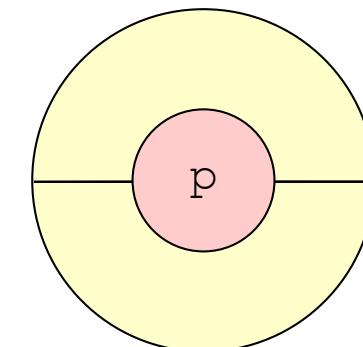
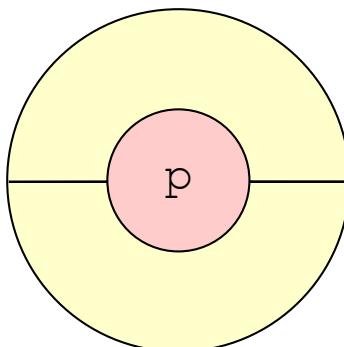
- Sometimes we need members that do not belong to a specific object
 - A method to perform a task independent of an object
 - e.g. `Math.random()`
 - A variable independent of a specific object
 - e.g. the total number of accounts created
 - A method to access such a variable
 - e.g. get the number of accounts created
- These members are declared as `static`
 - i.e. they belong to the whole class, not a specific object

Static members

- The reserved word `static` declares an attribute or method at the class level

```
private static int q;  
private int p;
```

- `static` ensures only one copy for the class, however many objects there are
 - Without `static` there is one copy per object

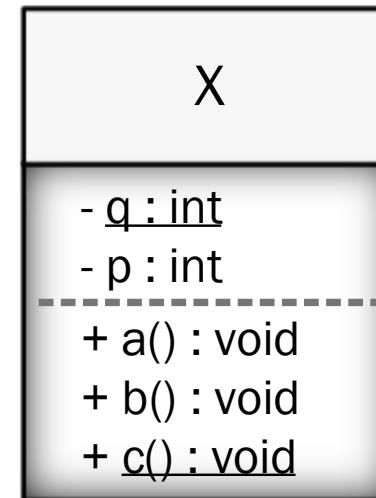


Activity



Coding a UML class diagram

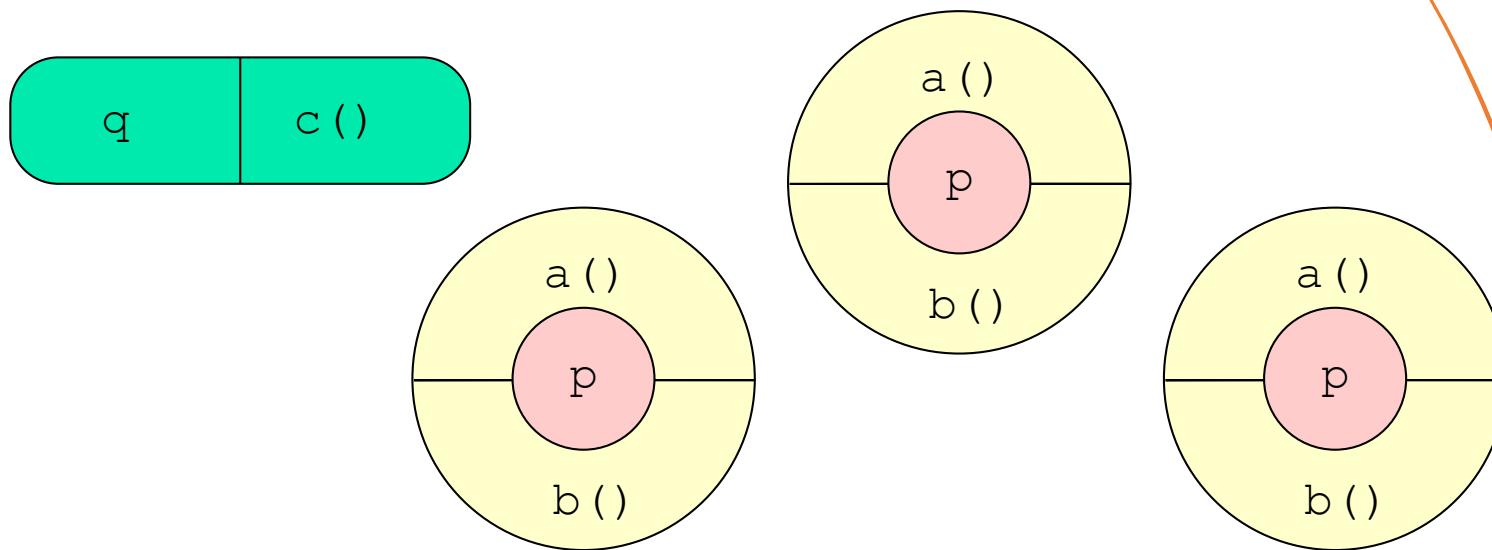
- In a class diagram, static attributes and methods are underlined



- What would the Java code look like for this class?

Static and non-static at run-time

- Assume that three objects of class X have been created
 - There is one copy of `q`, which is available to all objects of the class
 - There are three copies of `p`, one per object, each available only to its containing object



Class diagram for Account

Account

- numberOfAccounts : int = 0
- sortCode : int
- accountNumber : int
- accountName : String

+ Account(sc:int,
 acNum:int,
 acName:String)

+ getNumberOfAccounts() : int
+ getSortCode() : int
+ setSortCode(sc:int) : void

```
public Account(int sc,
               int acNum,
               String acName)
{
    sortcode = sc;
    accountNumber = acNum;
    accountName = acName;
    numberOfAccounts++;
}
```

```
public static int getNumberOfAccounts()
{
    return numberOfAccounts;
}
```

Using static members

- Object members belong to an object, therefore...
 - They do not exist until the object is created
 - They are accessed using the object name

```
Account ac1 = new Account(103546,  
                         12345678,  
                         "G. Mansfield");  
  
int sc = ac1.getSortCode();
```

- Static members belong to the class, therefore...
 - They exist even if no object has been created
 - They are accessed using the class name

```
int numAccounts = Account.getNumberOfAccounts();
```

Static methods

- A static method can only access
 - Static attributes in the same class
e.g. `q`
 - Public static attributes in other classes
e.g. `java.awt.Color.Red`
- A static method can only call
 - Static methods in the same class
e.g. `someStaticMethod()`
 - Public static methods in other classes
e.g. `System.out.println()`

Association, Aggregation, Composition

Lecture notes



Association

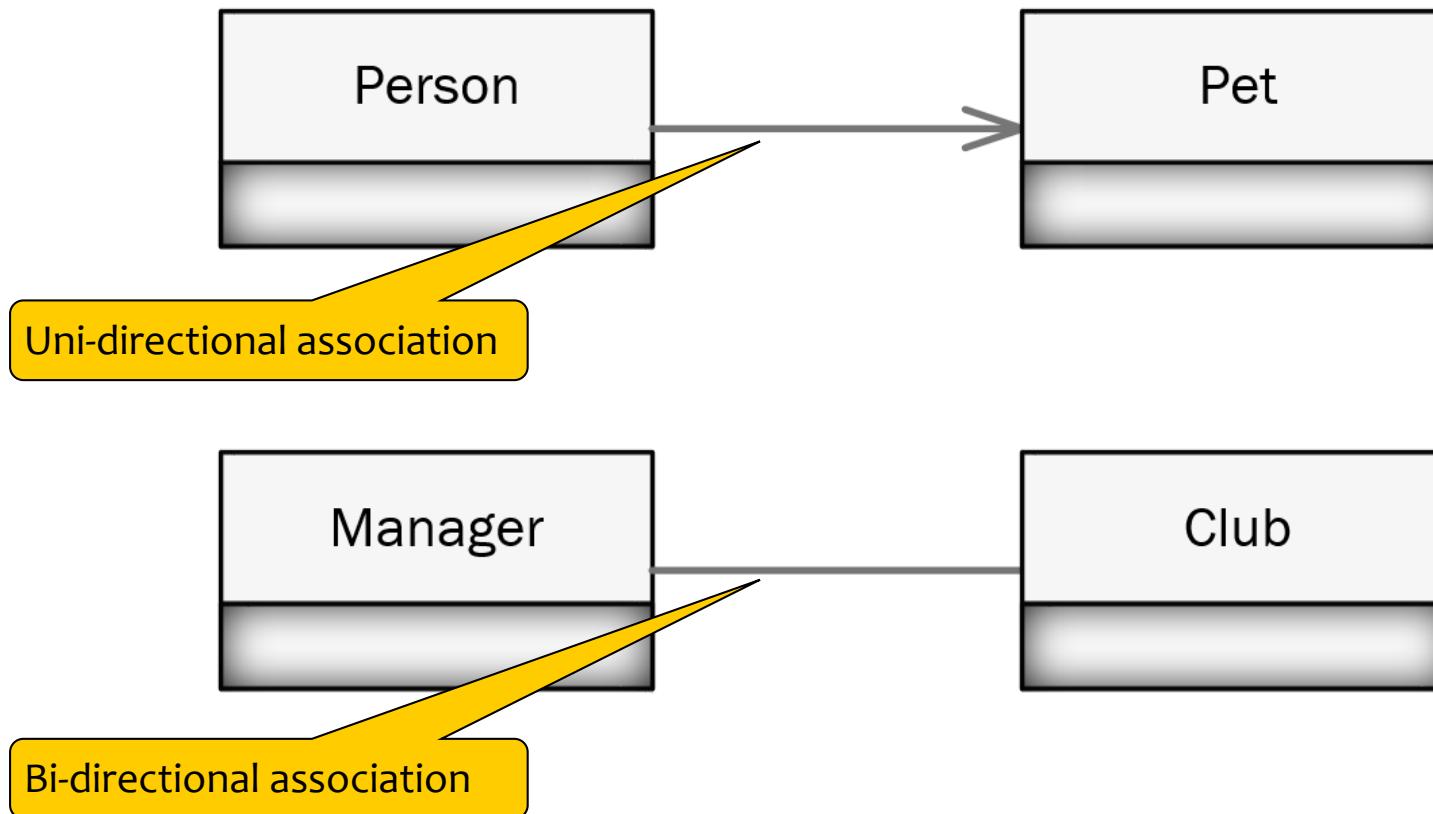
- An association indicates a HAS_A relationship between two classes
- Examples
 - A person HAS_A pet
or
A person owns a pet
 - A club HAS_A manager
or
A manager manages a club

Association

- Conceptually, one class is no more important than the other
 - A person is created (and destroyed) independently of a pet
 - The association begins when the person buys the pet
 - A manager exists without a club, and vice versa
 - The association begins when the club engages the manager

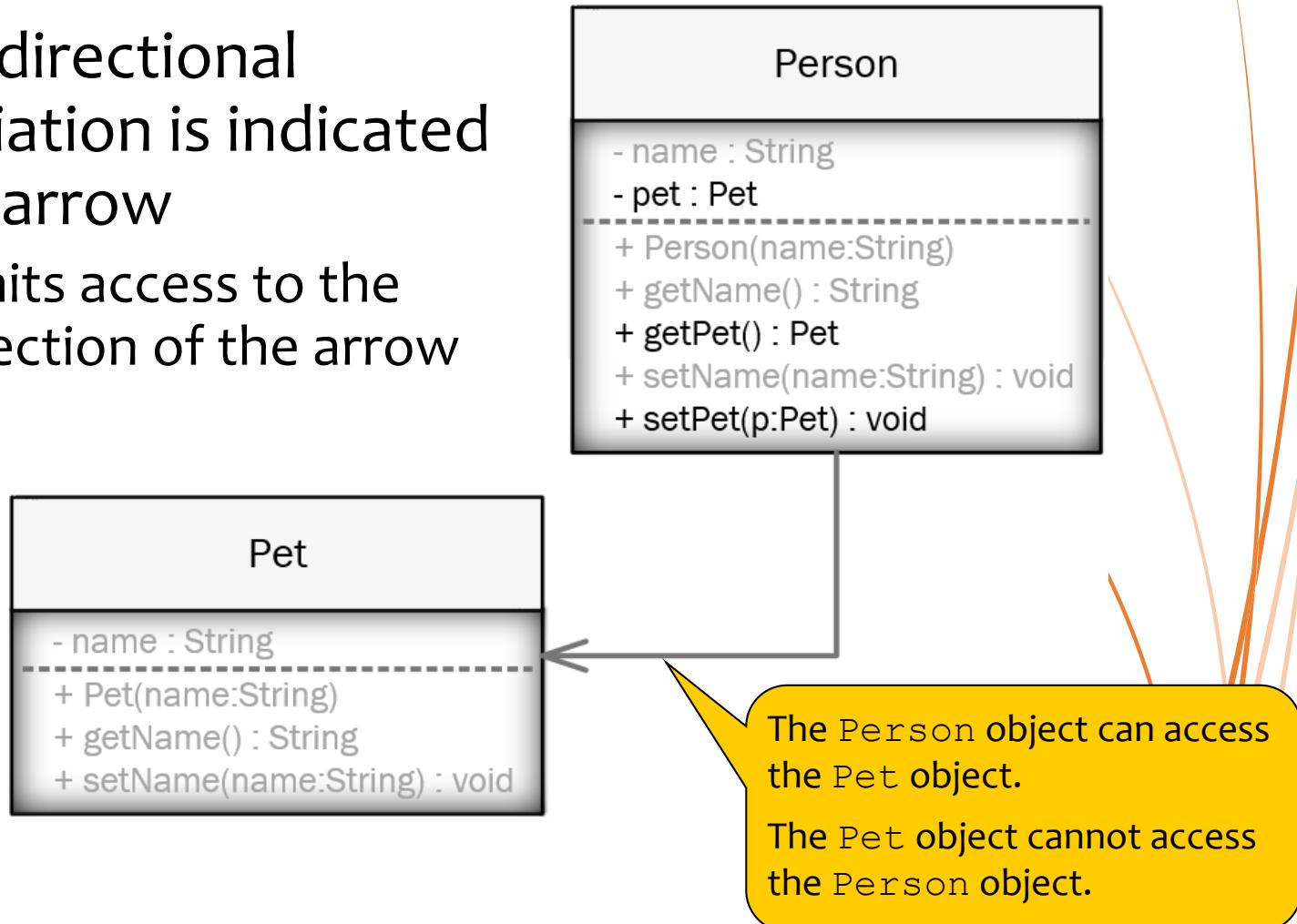
Association in UML

- An association is represented by a solid line between two classes



Uni-directional association

- A uni-directional association is indicated by an arrow
 - Limits access to the direction of the arrow



A good model

- Using an association is good modelling
 - The Pet and Person objects are created independently
 - They are associated (using the set method) after creation

```
Person tim = new Person("Tim");
Pet fluffy = new Pet("Fluffy");

tim.setPet(fluffy);
```

Activity



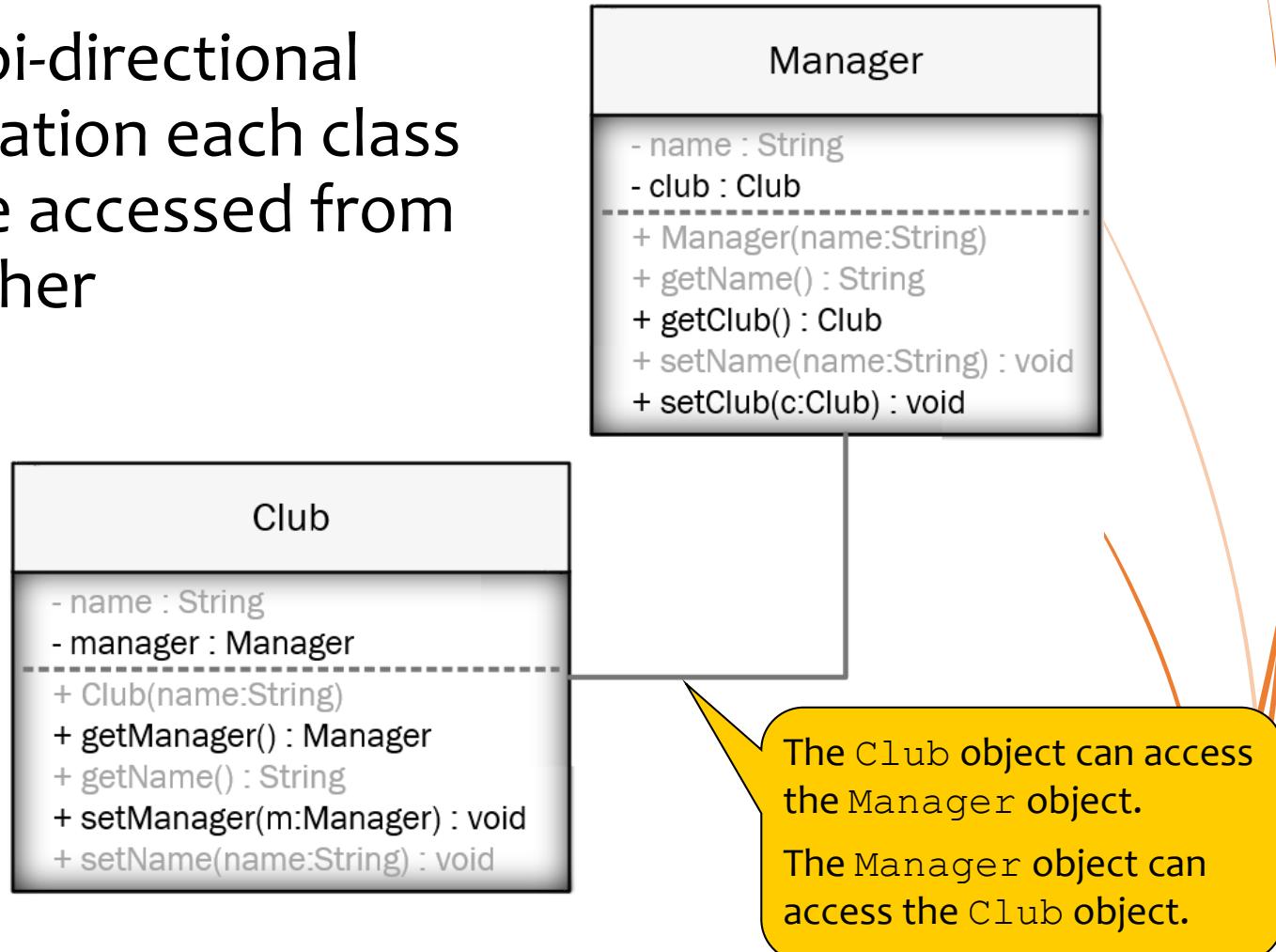
Coding

- Review the example code and watch out for:
 - how unidirectional association is implemented
 - how `printPersonWithPetDetails()` tests for the presence of a `Pet` object
- If you have any questions, ask

Project: UnidirectionalAssociation

Bi-directional association

- With bi-directional association each class can be accessed from the other



Bi-directional association: consistency

- In theory, the `setClub()` and `setManager()` methods could be called independently
 - This could lead to inconsistency in the association i.e. Club c might point to Manager m , but Manager m might not point to Club c
- Make one of the classes responsible for maintaining consistency
 - e.g. the `setManager()` method calls the `setClub()` method

Activity



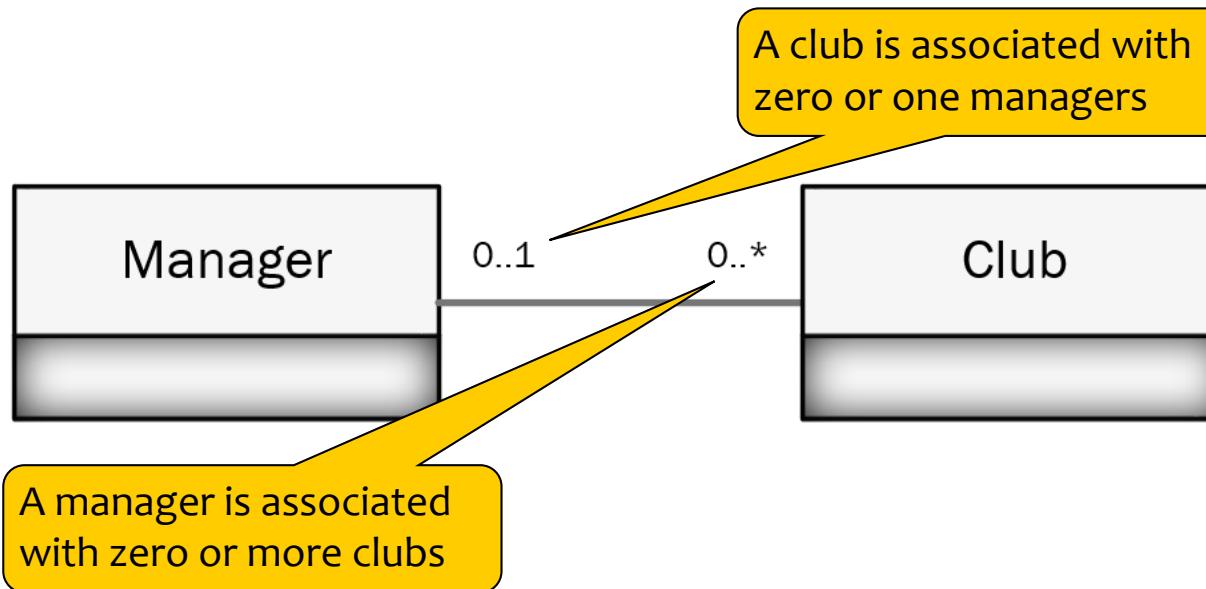
Coding

- Review the example code and watch out for:
 - how bidirectional association is implemented
 - how consistency is maintained in the bidirectional relational
- If you have any questions, ask

Project: BidirectionalAssociation

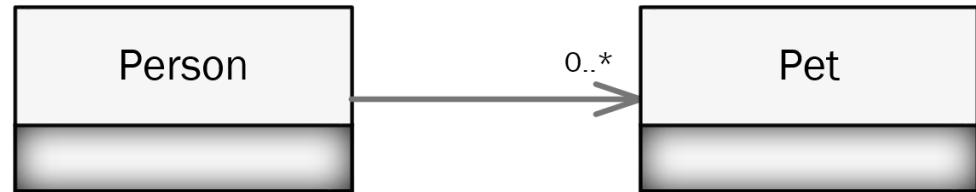
Multiplicity of an association

- Indicates how many objects are connected via the association
 - An exact number – e.g. 3
 - A range – e.g. 2..4
 - Many – e.g. 0..*



Implementing multiplicity

- $0..*$ can be implemented with a List object



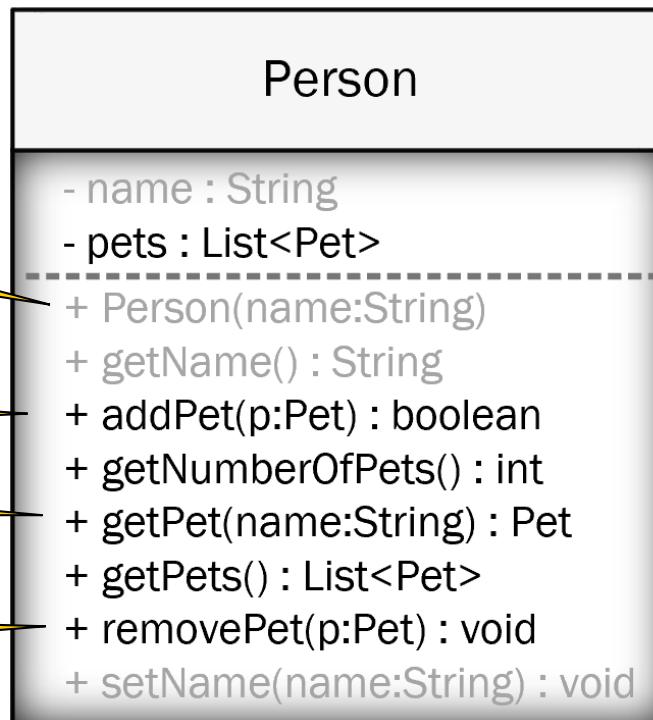
This is the pattern of methods needed for a collection of objects

The constructor creates an empty List

setPet () has been replaced by addPet ()

getPet () has been modified

If a pet can be added, it can also be removed



Activity



Coding

- Review the example code and watch out for:
 - how multiplicity is implemented
 - how to prevent the same object being added twice to the collection
- If you have any questions, ask

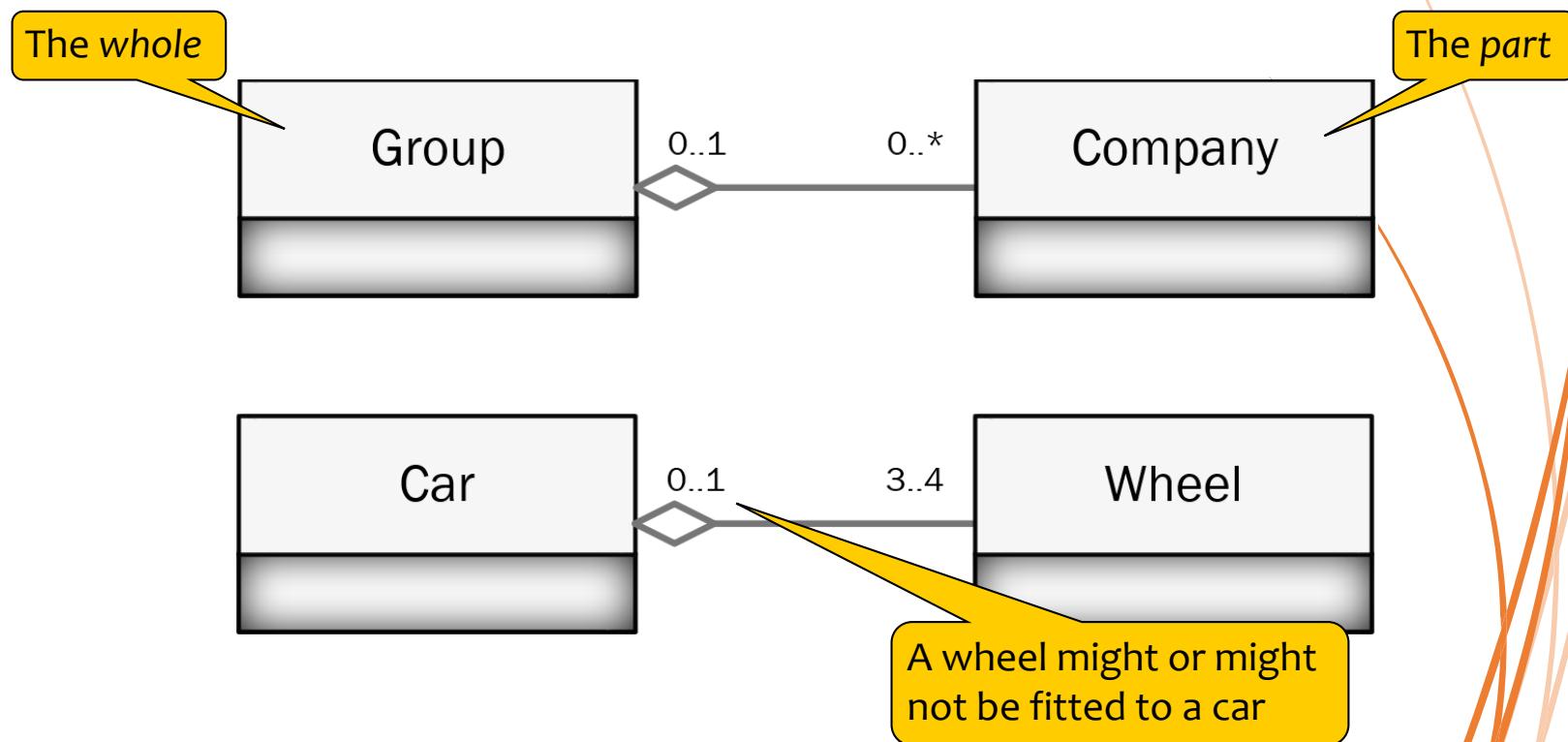
Project: Multiplicity

Aggregation

- An aggregation describes a *whole:part* relationship between two classes
- Examples
 - A commercial group has two or more companies
 - A car has three or four wheels
- This is an association with the addition of the *whole:part* concept
 - The lifetimes of the *whole* and the *part* are not linked
 - A car's wheel can be replaced by another wheel
 - The commercial group can survive a company, and vice versa
 - A new company could join the group years into its existence

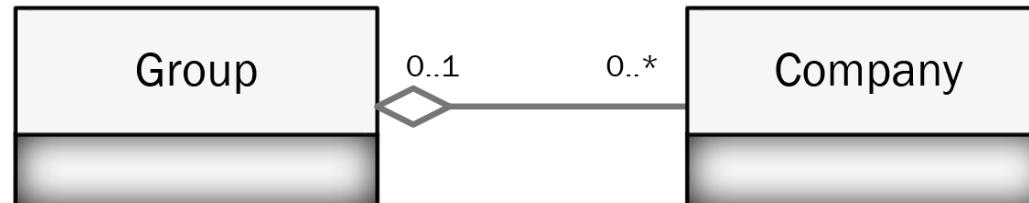
Aggregation in UML

- An aggregation is represented by a solid line between two classes with an open diamond at the whole end



Aggregation vs. association

- To distinguish between a plain association and an aggregation, ask “Is B part of A?”
 - Yes – it’s an aggregation
 - No – it’s an association
- Examples
 - Is Company part of Group? – Yes



- Is Pet part of Person? – No



Implementing aggregation

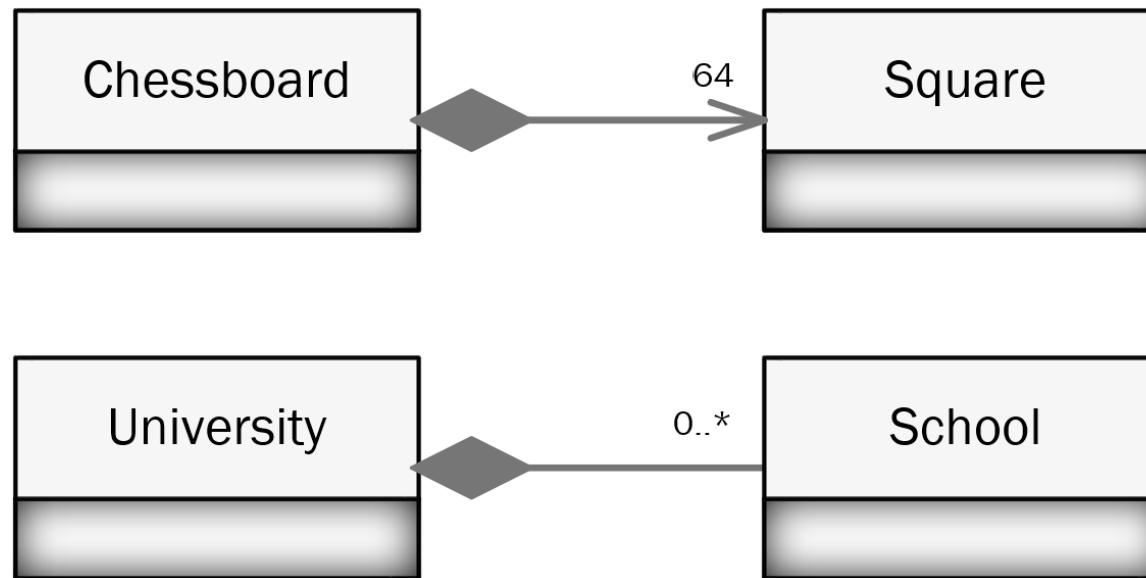
- There is no difference between implementing a “many” association and implementing an aggregation
 - Use a List
 - Suitable for any situation
 - Use an array
 - Probably only suitable if the number of items is fixed
- Aggregation is only conceptually different from association
 - When the aggregation symbol is used, it alerts the reader to the whole:part relationship
 - That's all

Composition

- A strong form of aggregation
 - The *part* objects **belong to** only one *whole* object at a time
- The *whole* object is responsible for the creation and destruction of its *part* objects
 - The *part* objects might be created later than the *whole* object, but they must be destroyed with it
- When the *whole* object is destroyed, the associated *part* objects are destroyed, too
 - *Part* objects may be removed prior to the destruction of the *whole* object

Composition in UML

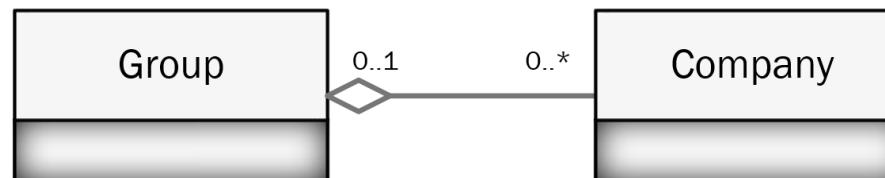
- A composition is represented by a solid line between two classes with a filled diamond at the whole end
 - A composition can be uni- or bi-directional



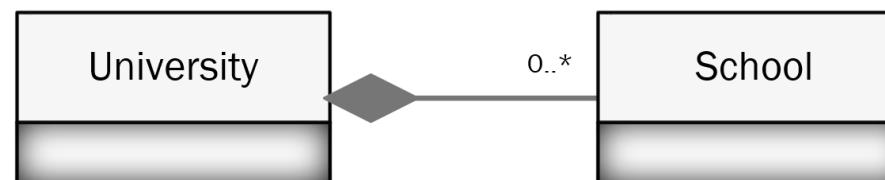
The part does not have an existence independent from the whole

Composition vs. aggregation

- To distinguish between a composition and an aggregation, ask “Is B created by A, and is B destroyed with A?”
 - Yes & Yes – it’s a composition
 - Otherwise – it’s an aggregation or association
- Examples
 - Is Company created by Group? – No



- Is Faculty created by University? – Yes
Is Faculty destroyed with University? – Yes



Implementing composition

- The *whole* object must create its *part* objects
 - Either when the *whole* object is created
i.e. in the constructor method
 - Or by a method in the *whole* object

Activity

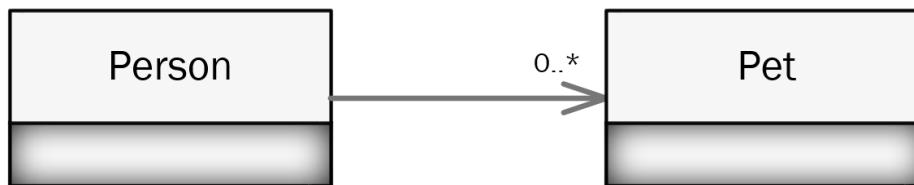


Coding

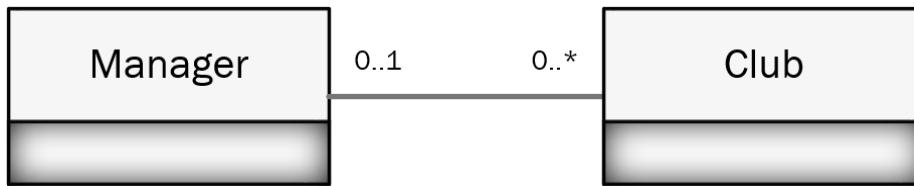
- Review the example code and watch out for:
 - how composition is implemented
 - how the whole creates the part
- If you have any questions, ask

Project: Composition

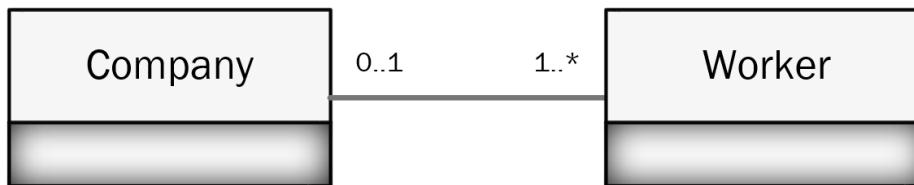
Association summary



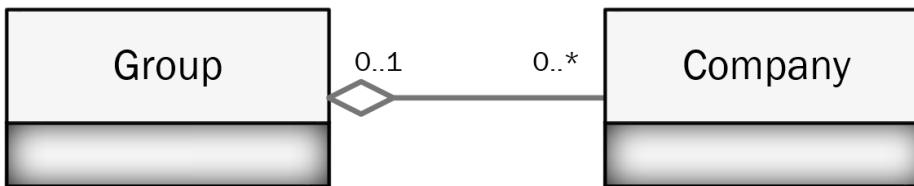
Uni-directional association



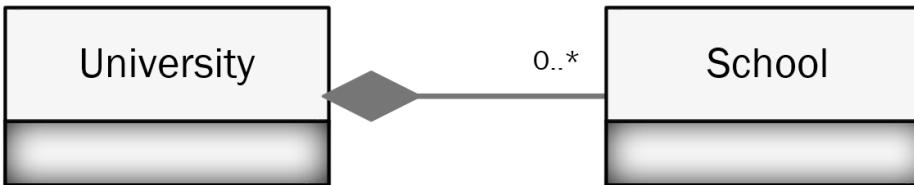
Bi-directional association



Multiple association



Aggregation
(whole:part)



Composition
(whole:part)

Inheritance and Polymorphism

Lecture notes

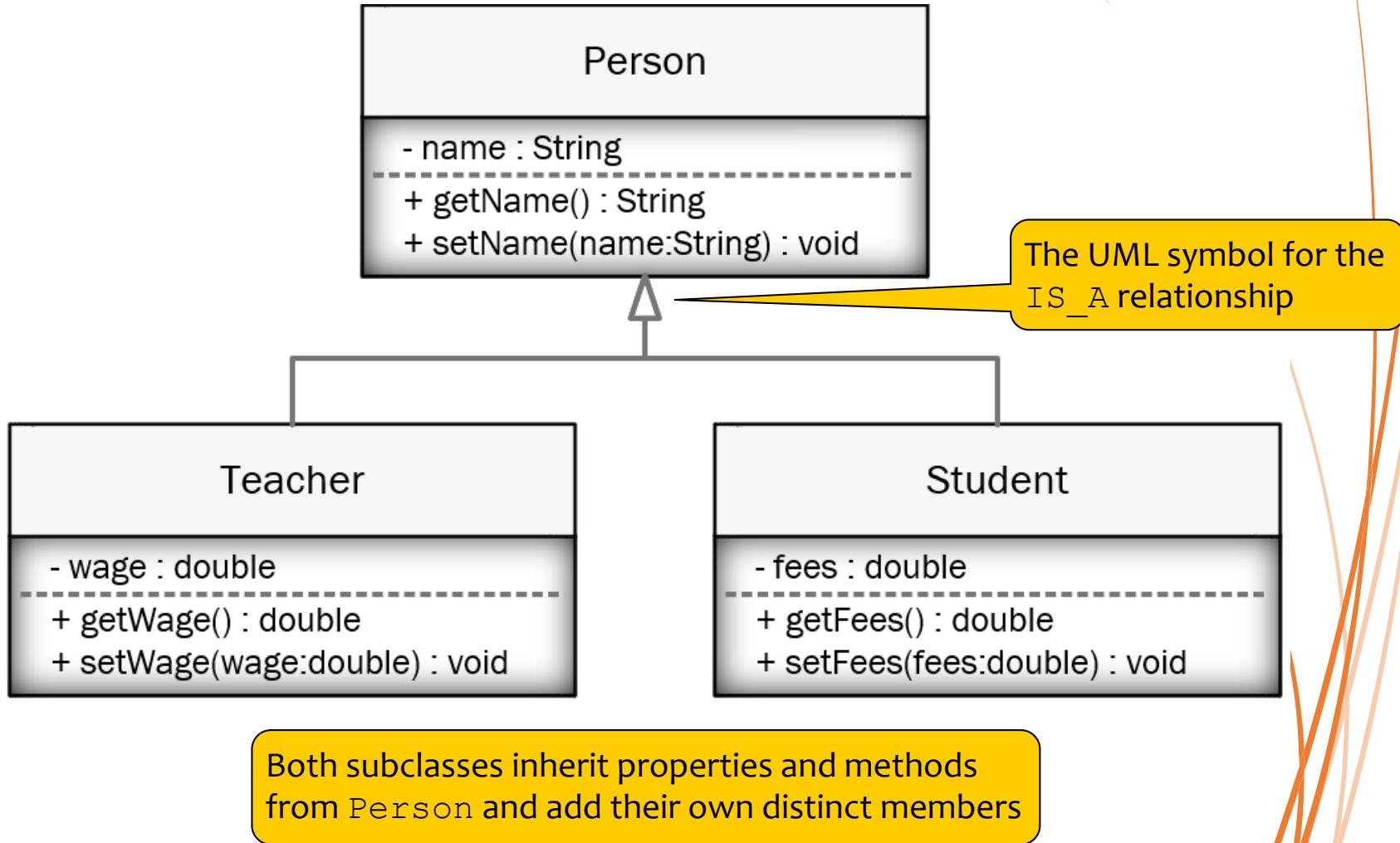
Inheritance

- To inherit
 - “To receive a characteristic from one’s parents”
(<http://www.thefreedictionary.com/inherit>)
- This principle is used extensively in OO
 - A class can inherit properties and methods from a parent
 - The parent class is called the **superclass**
 - The inheriting class is called the **subclass**
- In Java, a subclass may only inherit directly from a single superclass
 - In some OO languages (C++), a subclass may inherit directly from many super-classes

Inheritance: the IS_A relationship

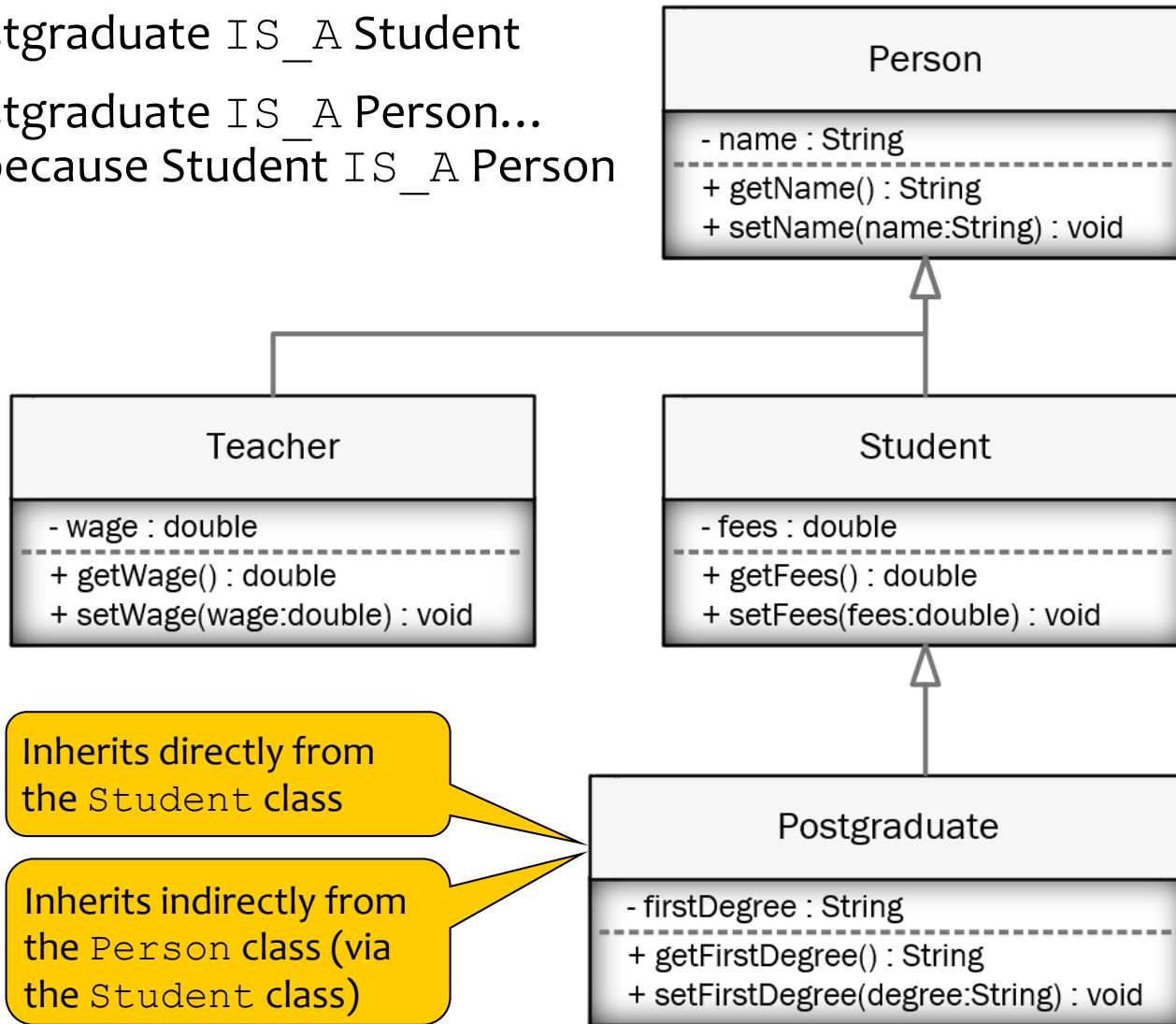
- The IS_A relationship
 - A teacher is a person
 - A student is a person
 - A postgraduate is a student
- Teachers and students have many traits in common
 - Describe the common characteristics in the Person class, whose properties and methods...
 - ...are inherited by the Teacher and Student subclasses
 - Similarly, the Postgraduate class inherits from the Student class

Inheritance in UML

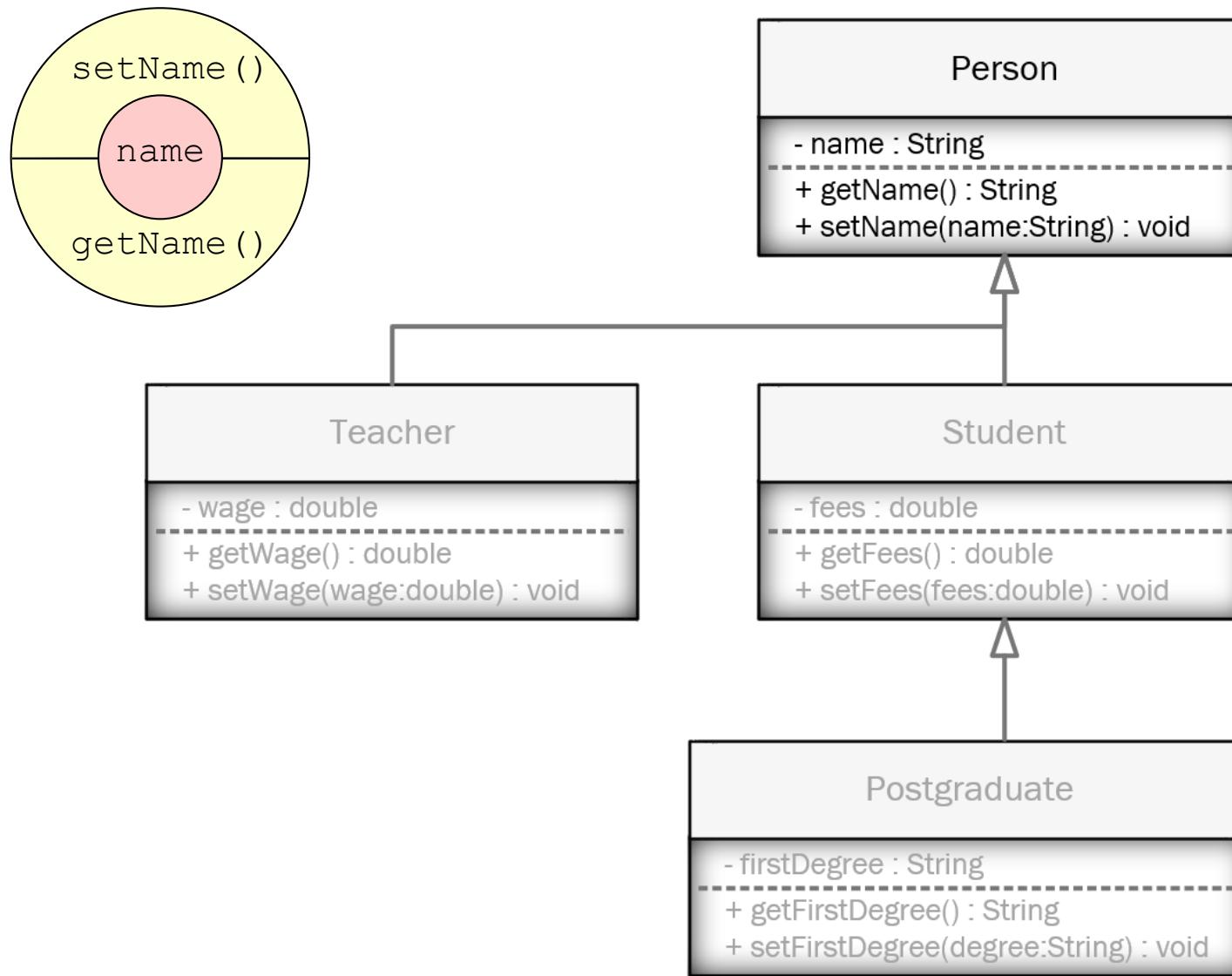


Inheritance hierarchies

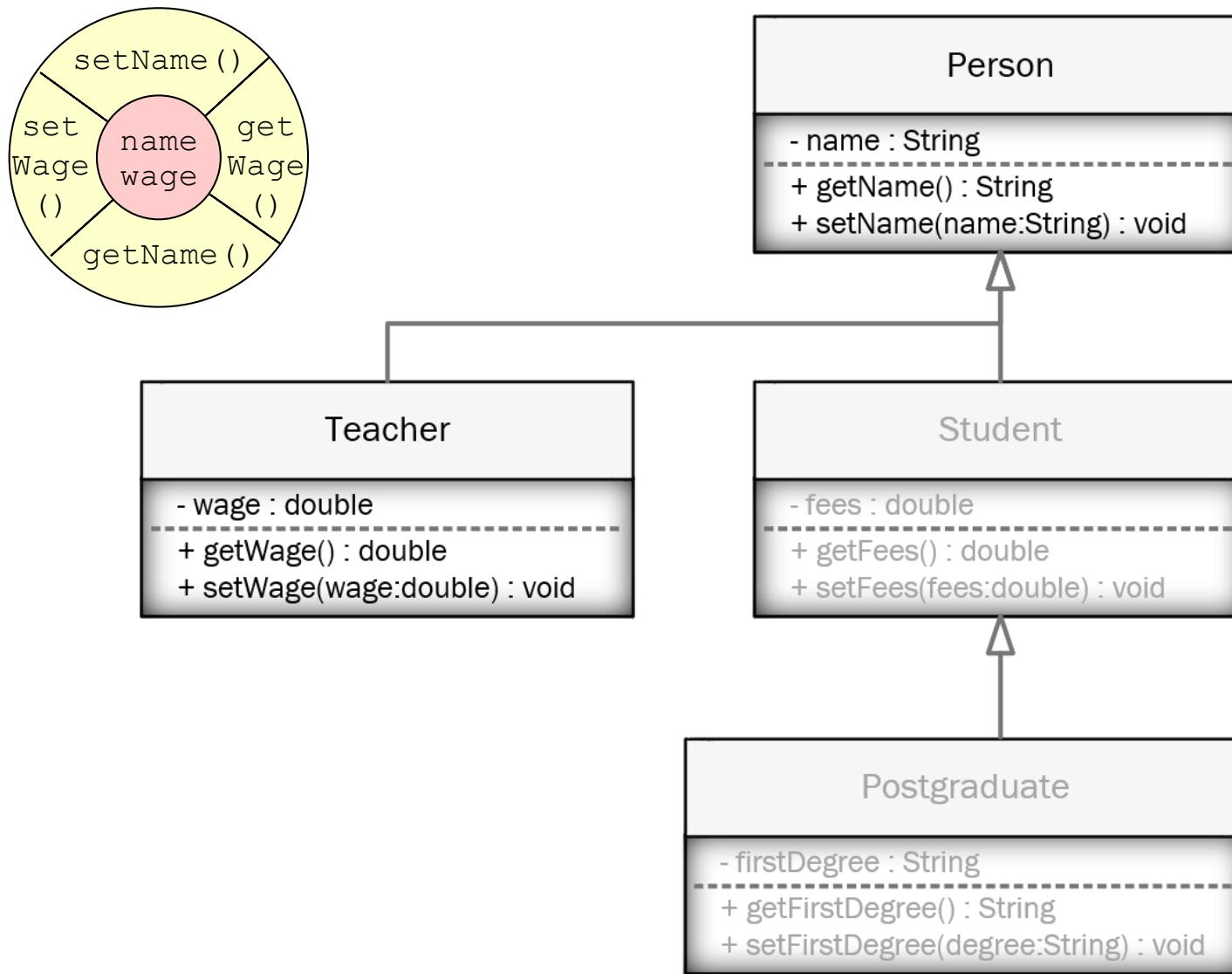
- Postgraduate IS_A Student
- Postgraduate IS_A Person...
...because Student IS_A Person



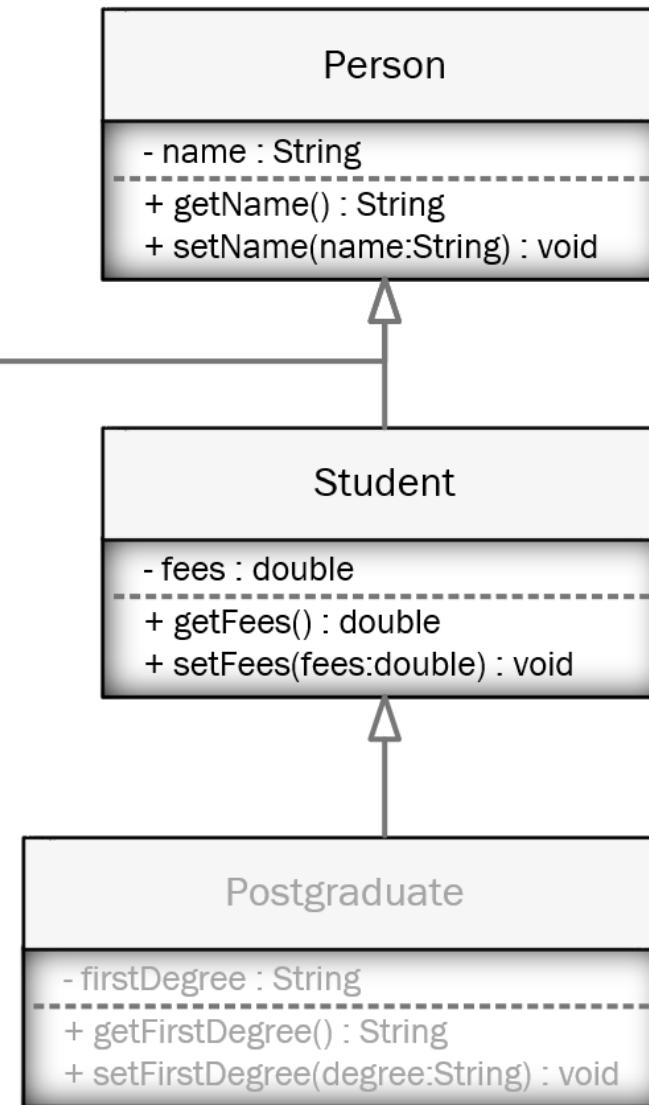
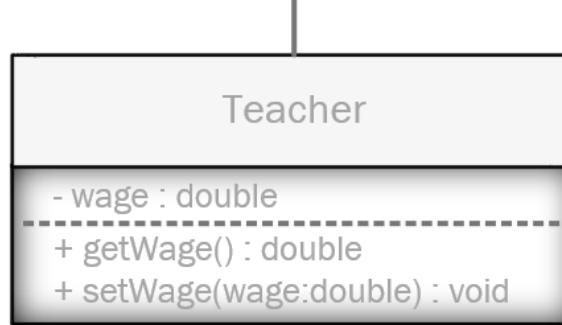
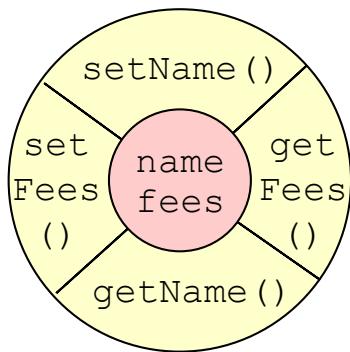
A Person object at run-time



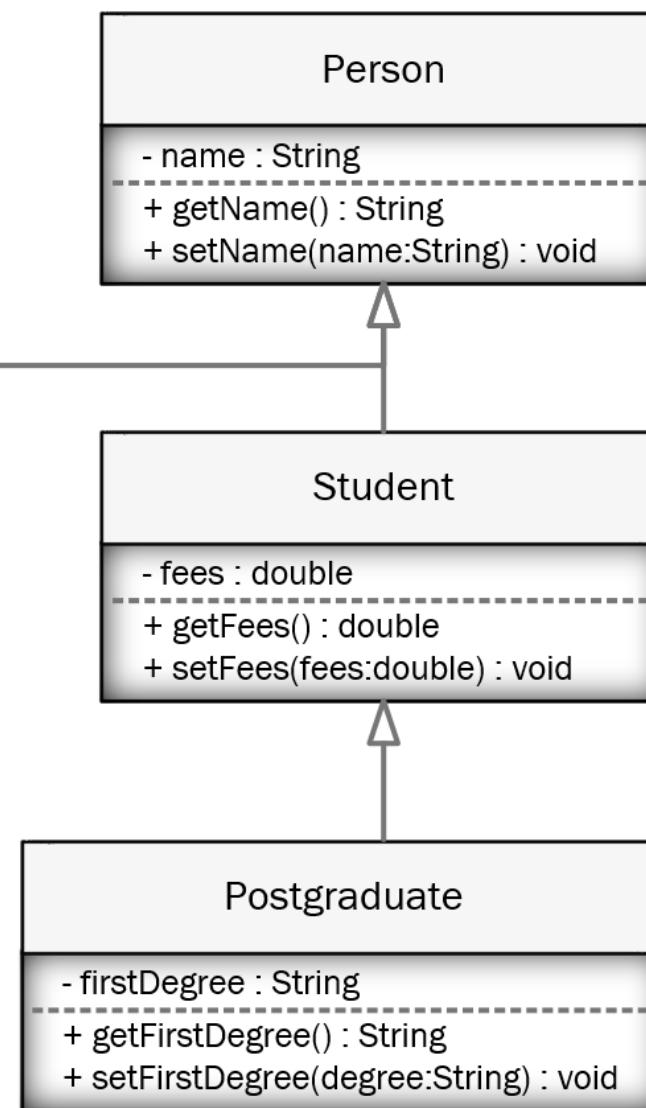
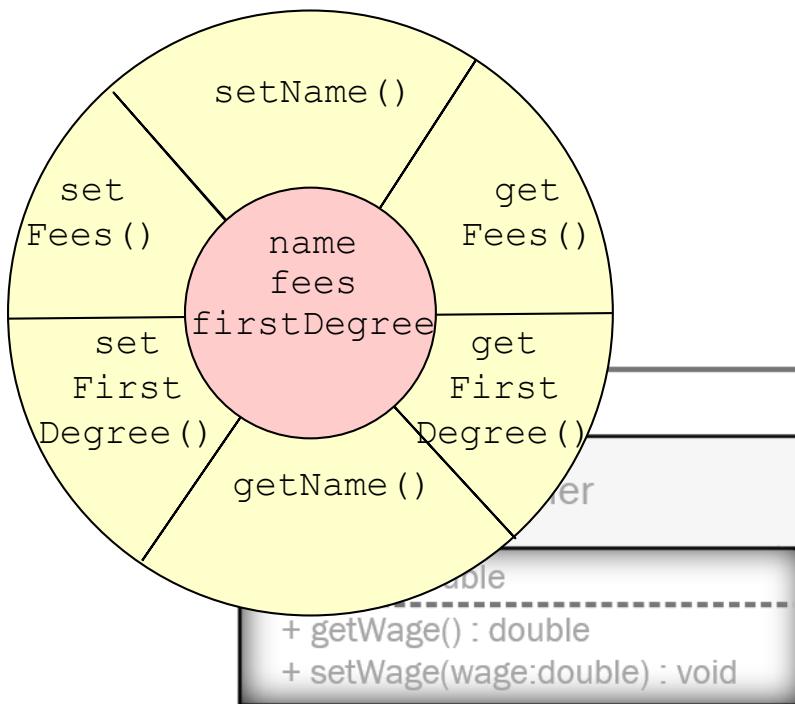
A Teacher object at run-time



A Student object at run-time



A Postgraduate object at run-time



To make a subclass in Java

- Use `extends` followed by the name of the super-class

```
public class Student extends Person
{
}
```

```
public class Postgraduate extends Student
{
}
```

- This immediately makes available to the subclass all the properties and methods of the super-class

Activity



Coding

- Review the example code and watch out for:
 - how the inheritance hierarchy is set up
 - calls to the subclasses' inherited methods
- If you have any questions, ask

Project: PersonInheritance

The Object class

- Every class in Java is a subclass of either...
 - A nominated superclass (using `extends`)
 - The `Object` class (if `extends` is not used)
- The `Object` class is in the `java.lang` package
 - It's automatically available to every class
 - There's no need to import it
- Therefore,
 - Person is a subclass of the `Object` class
 - `Object` is at the top of every inheritance hierarchy in Java

Constructors and inheritance

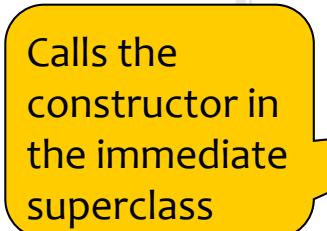
- An inheritance hierarchy dictates the order of object creation
- When a Student object is created, the following calls are made...
 - The constructor of the Student class, which immediately calls...
 - The constructor of the Person class, which immediately calls...
 - The constructor of the Object class
- Why?
 - To initialise the object's properties controlled by the relevant class

Which constructor is called?

- A super-class might have more than one constructor method. Which is called?
 - The one called by the subclass using `super`

```
public class Student extends Person
{
    public Student(String name, double fees)
    {
        super(name);
        this.fees= fees;
    }
}
```

Calls the constructor in the immediate superclass



The code shows a `Student` class that extends `Person`. The constructor takes `String name` and `double fees` as parameters. It calls the constructor of the immediate superclass `Person` using `super(name)` and initializes the `fees` field. A yellow callout points to the `super(name)` line with the text "Calls the constructor in the immediate superclass". Two red circles highlight the word `super` in both the class declaration and the constructor call.

- If the subclass does not use `super`
 - The superclass' constructor is automatically called (i.e. the one without parameters)

Activity



Coding

- Review the example code and watch out for:
 - the constructors that have been added into the inheritance hierarchy
 - how the use of constructors makes the `Main()` method simpler
- If you have any questions, ask

Project: PersonInheritance2

Putting inheritance to work

- Inheritance permits assignments such as...

```
Postgraduate pg1 = new Postgraduate();  
Student s = pg1;
```

- Because...

- pg1 **IS_A** Postgraduate
- a Postgraduate **IS_A** Student
- Therefore, pg1 **IS_A** Student
 - and a Student variable can hold a reference to a Postgraduate object

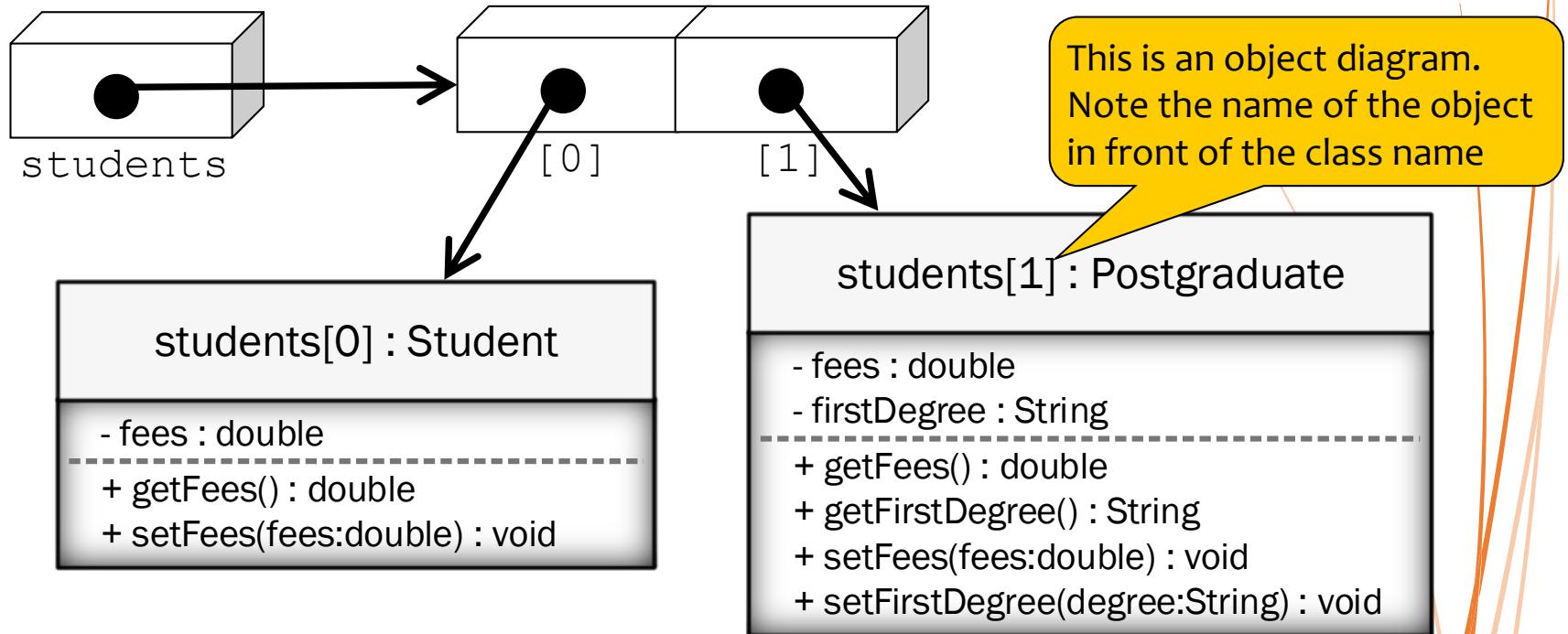
Putting inheritance to work

- If it can be done for a single variable, it can be done for an array

```
Student[] students = new Student[2];  
  
students[0] = new Student();  
  
students[1] = new Postgraduate();
```

- An array can hold objects of subclasses of its declared class
 - An array of Student objects can hold Postgraduate objects as well
 - Because a Postgraduate IS_A Student

There is a problem

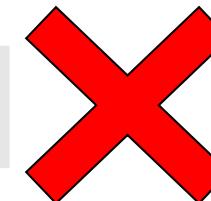


- Given `students[1]` is a Postgraduate

- We might expect this to work...

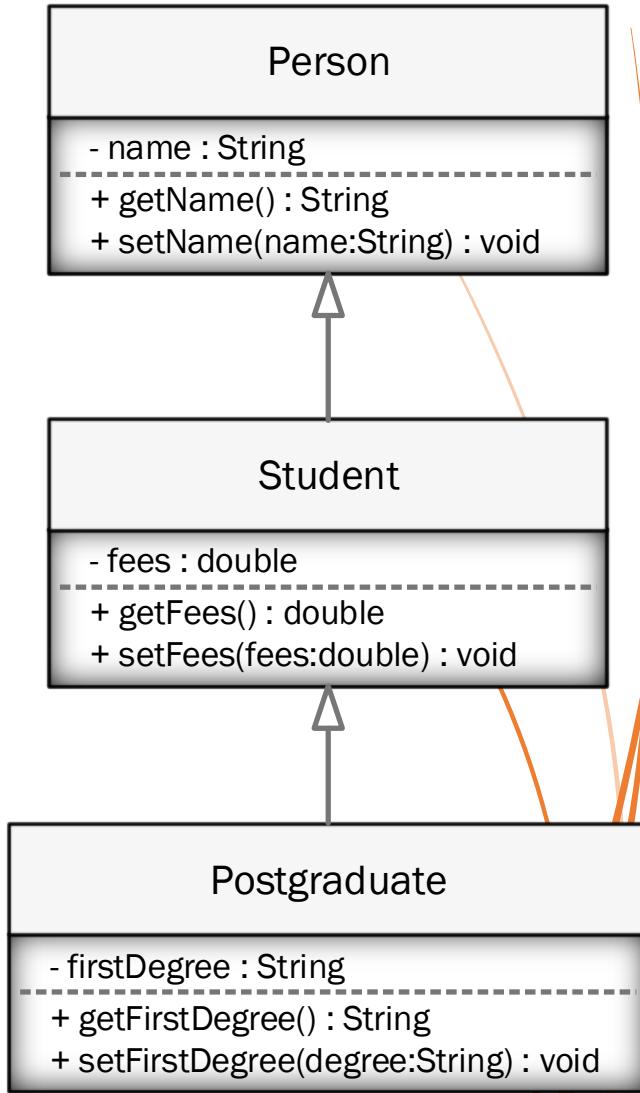
```
students[1].getFirstDegree();
```

- ...but it doesn't compile



Why not?

- A Student reference may store objects of subclasses (e.g. Postgraduate)...
 - but only the methods and properties from the Student class and its superclasses (e.g. Person) are available
- getFirstDegree () is a method declared in the Postgraduate class
 - it is not accessible via a reference of type Student



instanceof operator

- A logical operator of the form:
 - <object_reference> instanceof <class_name>
 - e.g. students[1] instanceof Postgraduate
- Returns true if the object is an instance of the nominated class
- Returns false otherwise

```
if (students[1] instanceof Postgraduate)
{
    // convert the student object to
    // a postgraduate object
}
```

This code will be executed
only if students [1] is a
Postgraduate object

`instanceof` operator

	<code>instanceof Person class</code>	<code>instanceof Teacher class</code>	<code>instanceof Student class</code>	<code>instanceof Postgraduate class</code>
<code>Person object</code>	✓	✗	✗	✗
<code>Teacher object</code>	✓	✓	✗	✗
<code>Student object</code>	✓	✗	✓	✗
<code>Postgraduate object</code>	✓	✗	✓	✓



Partial solution

- Type cast `students[1]` to a `Postgraduate`

```
if (students[1] instanceof Postgraduate)
{
    Postgraduate pg = (Postgraduate) students[1];
    System.out.println(pg.getFirstDegree());
}
```

- When `students[1]` is referred to by a variable of type `Postgraduate` ...
 - the `getFirstDegree()` method is available

Extending the concept

- If this concept works for the Student and Postgraduate classes...
 - It must work for all classes
- Therefore...
 - a Teacher IS_A Person
 - a Student IS_A Person
 - a Postgraduate IS_A Student
- The PersonInheritance example can be refined to hold all Person objects in an array
 - See the project PersonInheritance3

Overriding methods

- Sometimes a method inherited from a super-class does not do what the subclass needs
 - So, rewrite the method in the subclass using exactly the same signature as the inherited method
- When a method in a subclass has exactly the same signature as a method in one of its super-classes...
 - the subclass method **overrides** the super-class method
- Not to be confused with overloaded methods (multiple methods with the same name in the same class, but with different parameter lists)

Common example of overriding

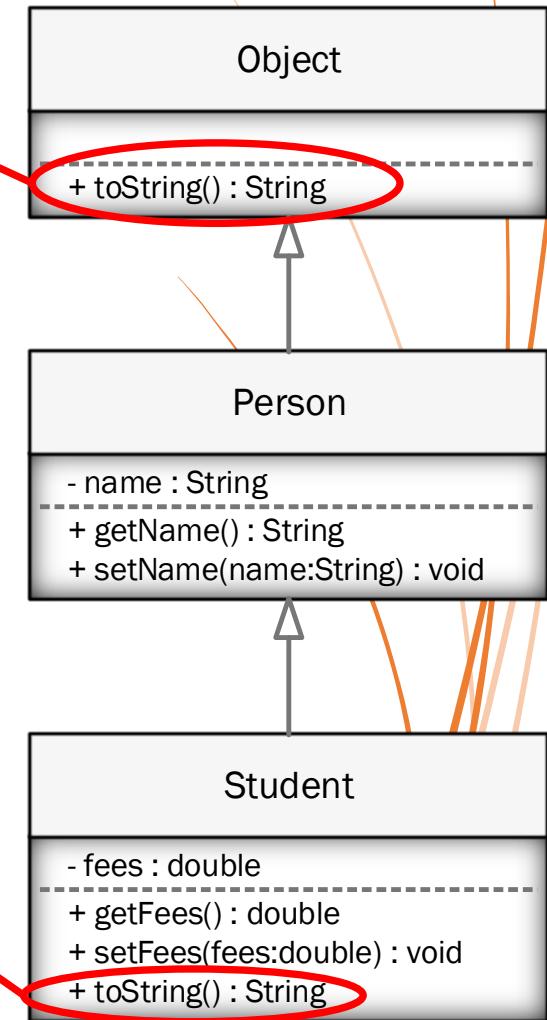
- Override the `toString()` method

- This method in the `Object` class returns a string of the form:

`<classname>@<address>`
e.g. `Object@23561287`

- A class can override this method to return something more meaningful

- The UML diagram shows that the `Student` class overrides the `toString()` method, whereas the `Person` class inherits from the `Object` class



Overriding the `toString()` method

- In the `Student` class...

```
@Override  
public String toString()  
{  
    return  
        System.out.printf("%s owes %1.2f GBP\n",  
                          Name,  
                          Fees);  
}
```

Exactly the same signature
as in the `Object` class

Project: PersonInheritance4

Activity

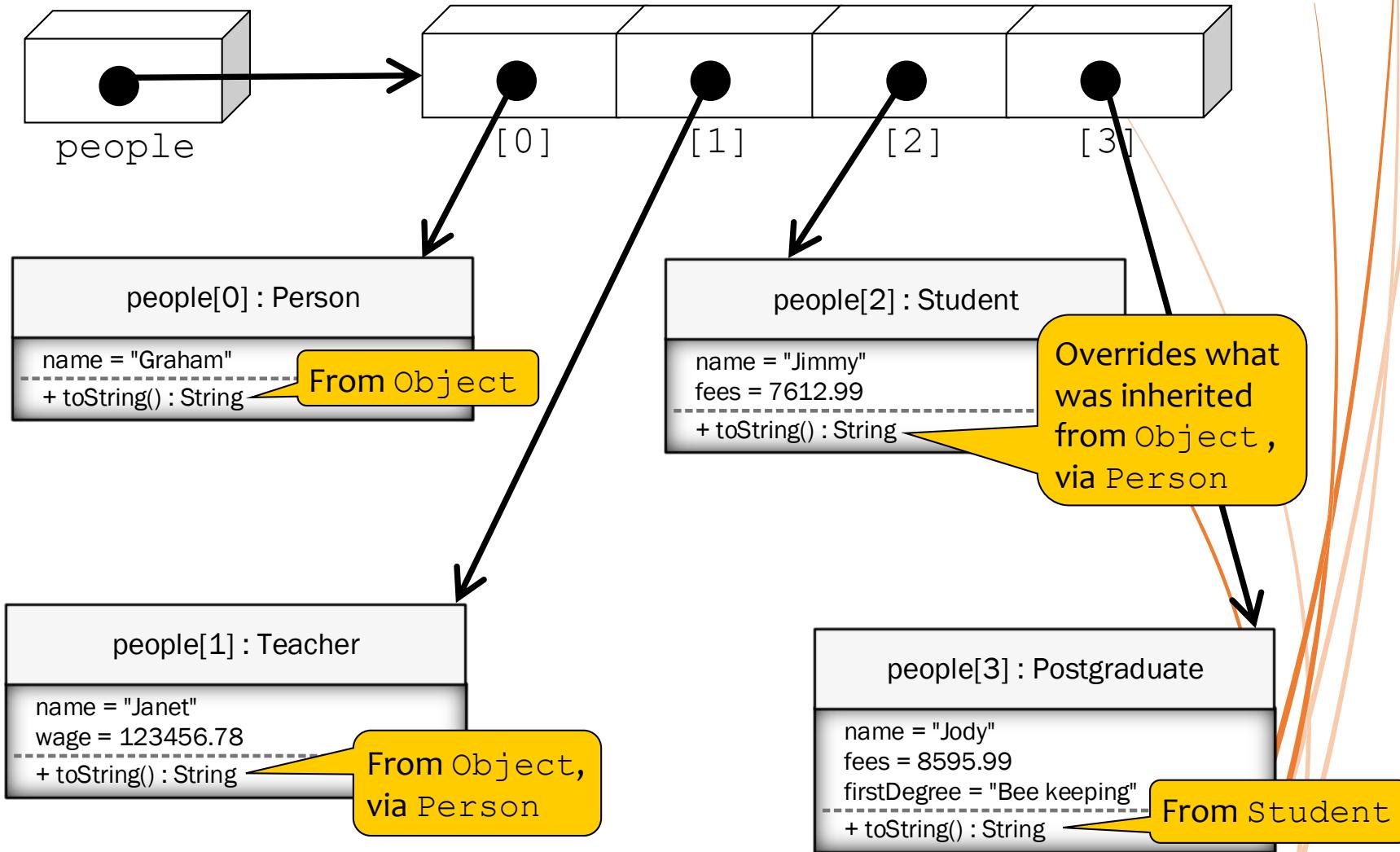


Coding

- Review the example code and watch out for:
 - how the `toString()` method has been overridden in the `Student` class
 - how the `main()` method in `Program` outputs the string returned by `toString()` for each object
- If you have any questions, ask

Project: PersonInheritance4

Inherit and override `toString()`



Which version to call?

- In the Main () method, p.ToString () produced different results according to the actual method called at run-time...

When p is a	Output	Execute <code>toString()</code> in
Person object	PersonInheritance4.Person	Object class
Teacher object	PersonInheritance4.Teacher	Object class (inherited from Person class)
Student object	Jimmy owes 7612.99 GBP	Student class (overrides inheritance from Person class)
Postgraduate object	Jody owes 8595.99 GBP	Student class

Project: PersonInheritance4

It all works out

- When a method has been overridden, the runtime environment works out which version of the method to call

```
System.out.println("toString(): " + p.toString());
```

- This is polymorphism

- If p refers to a Person or Teacher object, the `toString()` method inherited from the Object class is called
- If p refers to a Student or Postgraduate object, the `toString()` method from the Student class is called

Polymorphism – *having multiple forms*

- Allows objects of subclasses to be stored in variables declared for a superclass

```
Person p = new Student();
```

- The variable p can store *multiple forms* of Person object

```
Person p = new Student();
p = new Teacher();
p = new Postgraduate();
```

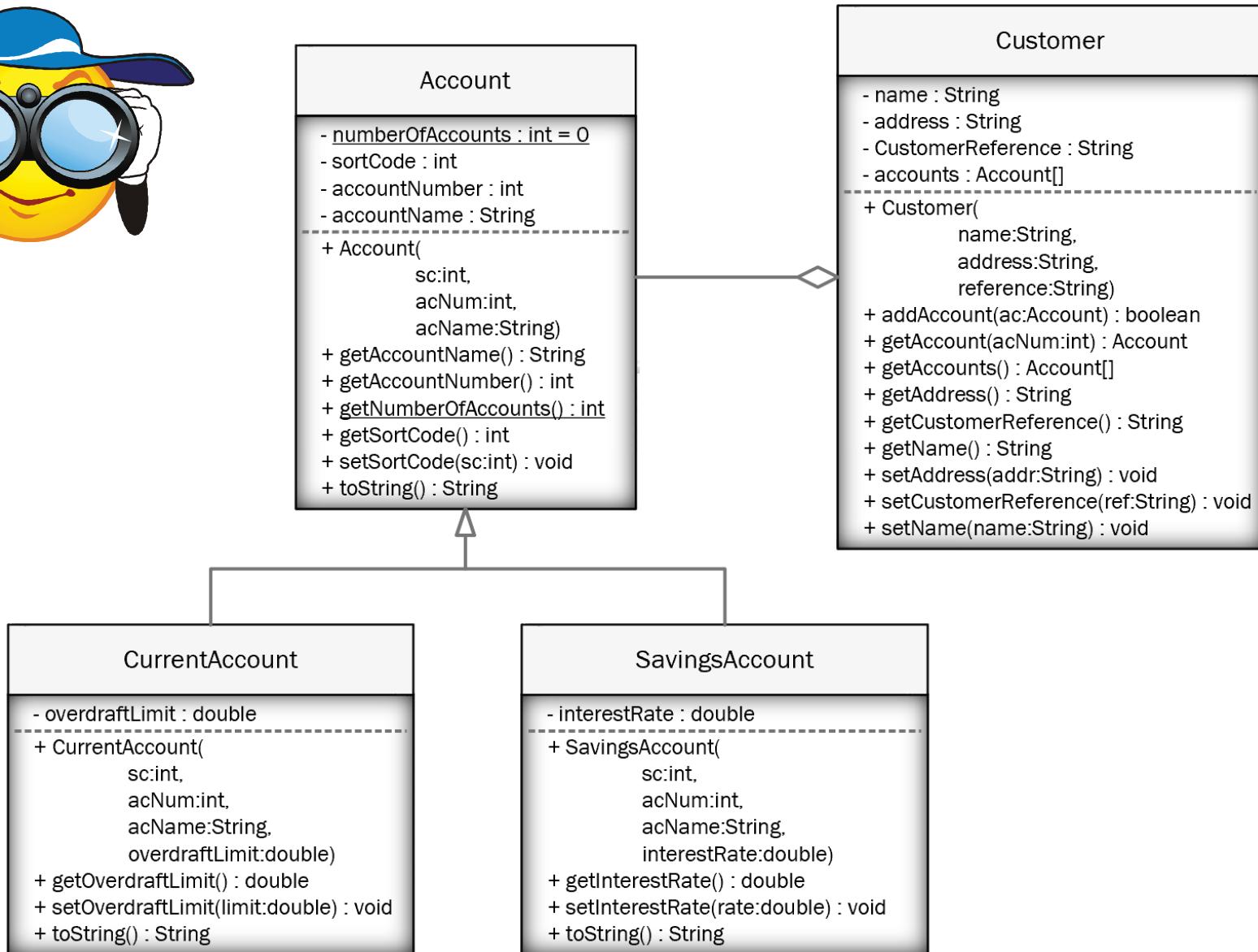
Polymorphism – *having multiple forms*

- Allows an overriding method to be called instead of the inherited method
 - Which method to execute is decided at runtime
 - This is called **dynamic binding** of the method definition to the method invocation
 - A method call can invoke *multiple forms* of method

```
Person p = new Teacher();  
p.toString();
```

```
Person p = new Student();  
p.toString();
```

Activity – where is the polymorphism?



Input & Output in Java

03

Input stream

- Input typically comes from:
 - keyboard
 - file
- Program code can read data from an **input stream**
- Input stream includes:
 - numbers
 - characters
 - white space
 - characters that represent horizontal or vertical space
 - e.g. [space bar], [Tab] and [Enter]
- White-space delimits tokens in the input stream

Input in Java

- Scanner is used to read input
- To use a Scanner we need to:
 - Import the Scanner
 - Create a Scanner
 - Use the input methods the Scanner provides

Import the Scanner

- Scanner is found in a **package** called `java.util`
- Packages are external to the program
- Packages must be imported into applications so the compiler can find them
- `import` statement must appear before the class header

```
import java.util.Scanner;  
public class ReadExample  
{  
}
```

Create a Scanner

- One Java statement to do three things

```
import java.util.Scanner;

public class ReadExample
{
    public static void main (String[ ] args)
    {
        Scanner kybd = new Scanner(System.in);

        //use the Scanner
    }
}
```

Tell the Scanner to use
System.in (the input
stream from the keyboard)

Use the Scanner's input methods

Input method	Comments
nextInt()	input integer eg. int i = kybd.nextInt();
nextDouble()	input double eg. double x = kybd.nextDouble();
next()	input string – up to next space or Enter eg. String s1 = kybd.next();
nextLine()	input string – up to next Enter eg. String s2 = kybd.nextLine();

Inputting numbers with Scanner

- White space delimits input
- `nextInt()` and `nextDouble()` read numbers up to, but not including, next white space
- Input data can be separated by white space

```
3 4 5<enter>
```

```
3<enter>
4<enter>
5<enter>
```

```
3 4<enter>
5<enter>
```

- If wrong type of data is input, program will throw an exception and crash

Inputting strings example – Details

- Problem – Details:
 - Prompt the user to input their name, age and address
 - Output each input item with some explanatory text
- What data is used?
 - name: String input by user
 - age: positive integer input by user
 - address: String input by user
- What operations are performed, and in what order?
 - Create Scanner to read in data
 - Prompt user for name, age and address
 - Input name, age and address
 - Output name, age and address

```
//input and output name, age and address

import java.util.Scanner;

public class Details
{
    public static void main (String[] args)
    {
        //create Scanner to read in data
        Scanner myKeyboard = new Scanner(System.in);

        //prompt user for input

        System.out.print("Enter name: ");
        String name = myKeyboard.nextLine();

        System.out.print("Enter age: ");
        int age = myKeyboard.nextInt();

        myKeyboard.nextLine(); //flush buffer

        System.out.print("Enter address: ");
        String address = myKeyboard.nextLine();

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Address: " + address);

    }
}
```

See
<http://youtu.be/bnNHKnHMbt4>

Output in Java

- No packages need to be imported for output
- System.out sends output to the screen
 - Unformatted output
 - print()
 - println()
 - Formatted output
 - printf()
- In unformatted output, strings can be concatenated using +
 - Joins individual strings and the values of variables into one string

Use the Scanner's output methods

Output method	Comments
print ()	output contents of brackets immediately following the previous output eg. System.out.print ("Hello");
println ()	output contents of brackets followed by new line eg. System.out.println ("Hello");
printf ()	output the data formatted as specified by the first parameter in the brackets

lower case L,
not 1

Unformatted output examples

- All examples assume name = "Bob", i = 3, x = 6.2

```
System.out.print("Hello ");  
System.out.print(name);
```

Hello Bob

```
System.out.println("Hello ");  
System.out.println(name);
```

Hello
Bob

```
System.out.print("Hello " + name);
```

Hello Bob

```
System.out.print("Hello ");  
System.out.print("name");
```

Hello name

name in
quotes

Unformatted output examples

```
System.out.print("Hello " + i + x);
```

Hello 36.2

converts each
variable to a string

```
System.out.print("Hello " + (i + x));
```

Hello 9.2

does calculation
then concatenates

```
System.out.print("Hello " + "i + x");
```

Hello i + x

calculation in quotes
is not a calculation!

Formatted output

- `printf()` has two parts:
 1. Format string containing **placeholders** for data items (shown by % in format string)
 2. list of **items** to be output separated by commas
 - number of placeholders and items must be same
- Placeholder specifies:
 - Type
 - d: decimal integer
 - f: floating point
 - s: string
 - Optional number of columns
 - Justification (default is right-justified)

Formatted output examples

```
System.out.printf("%d", i);
```

3

```
System.out.printf("%f", x);
```

6.200000

defaults to 8 columns for value

```
System.out.printf("%s", "Bob");
```

Bob

```
System.out.printf("%s adds %d to %1.1f",
                  name, i, x);
```

Bob adds 3 to 6.2

three placeholders;
so three data items

Formatted output examples

```
System.out.printf("%8s", "Bob");
```

— — — — Bob

specifies width of field

```
System.out.printf("%-8s", name);
```

Bob — — — —

changes alignment using -

```
System.out.printf("%7.3f", x);
```

— — 6.200

sets number of decimal places

Tab and newlines

- New lines and tab characters can be included in print statements

- \t tab to next print area
- \n output new line
- \r\n output return character with new line
(needed for some applications like Notepad)

```
System.out.print("\nHello\n" + name);
```

Hello
Bob

outputs newline before and after

```
System.out.printf("\t%d\n\t%3.1f\n",
                  i, x);
```

----- 3
----- 6.2

uses tabs and newlines to format

Advanced Java class concepts

Lecture slides

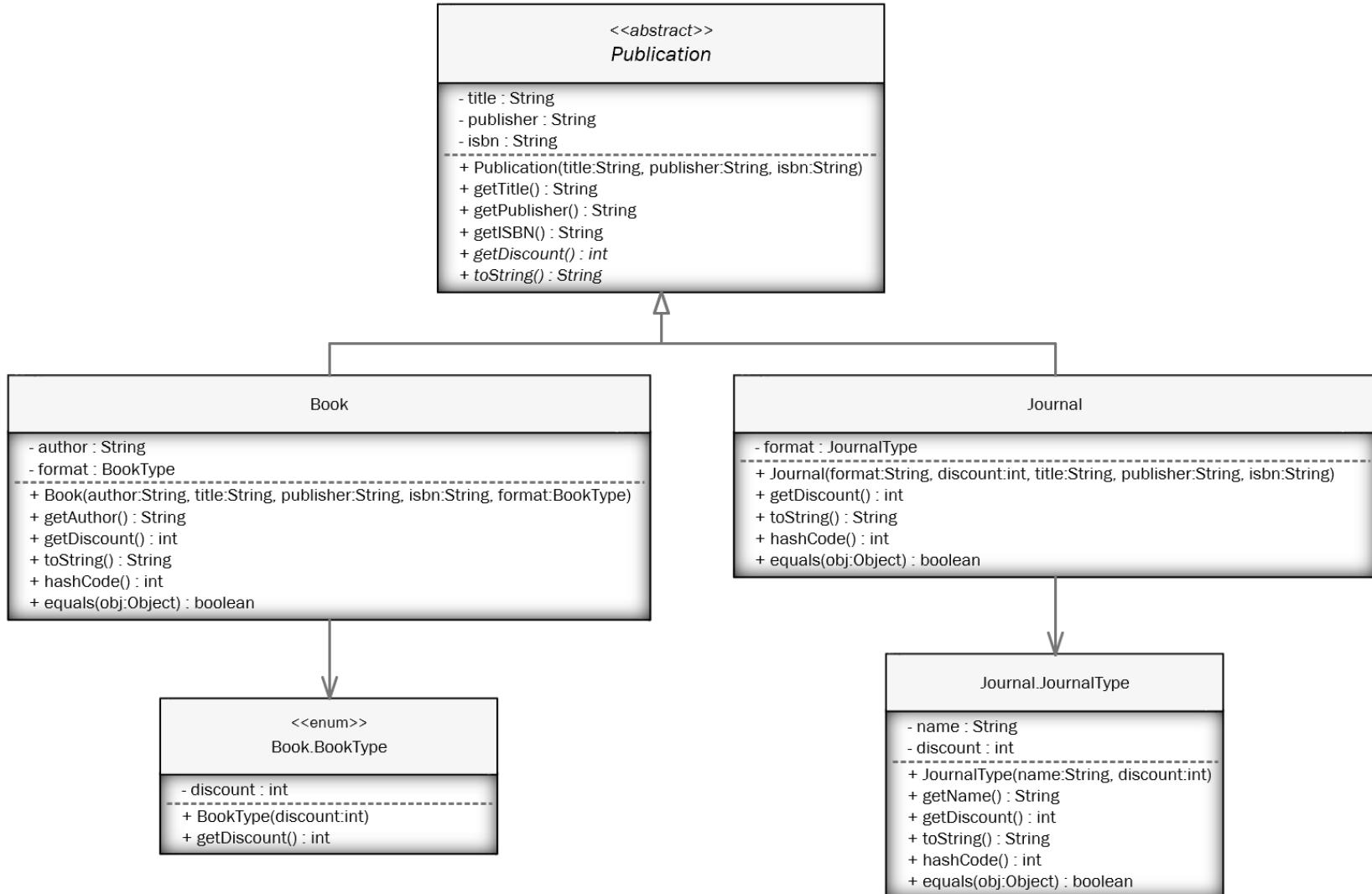
In this lecture

- You should have read the lecture notes before coming to this lecture
- We will:
 - Write a Java program that demonstrates the concepts in the lecture notes
 - Override hashCode(), equals(), and toString() methods from the Object class
 - Abstract classes and methods
 - final keyword
 - Inner classes
 - Enumerated types
- Ask questions about anything you don't understand

Problem 1

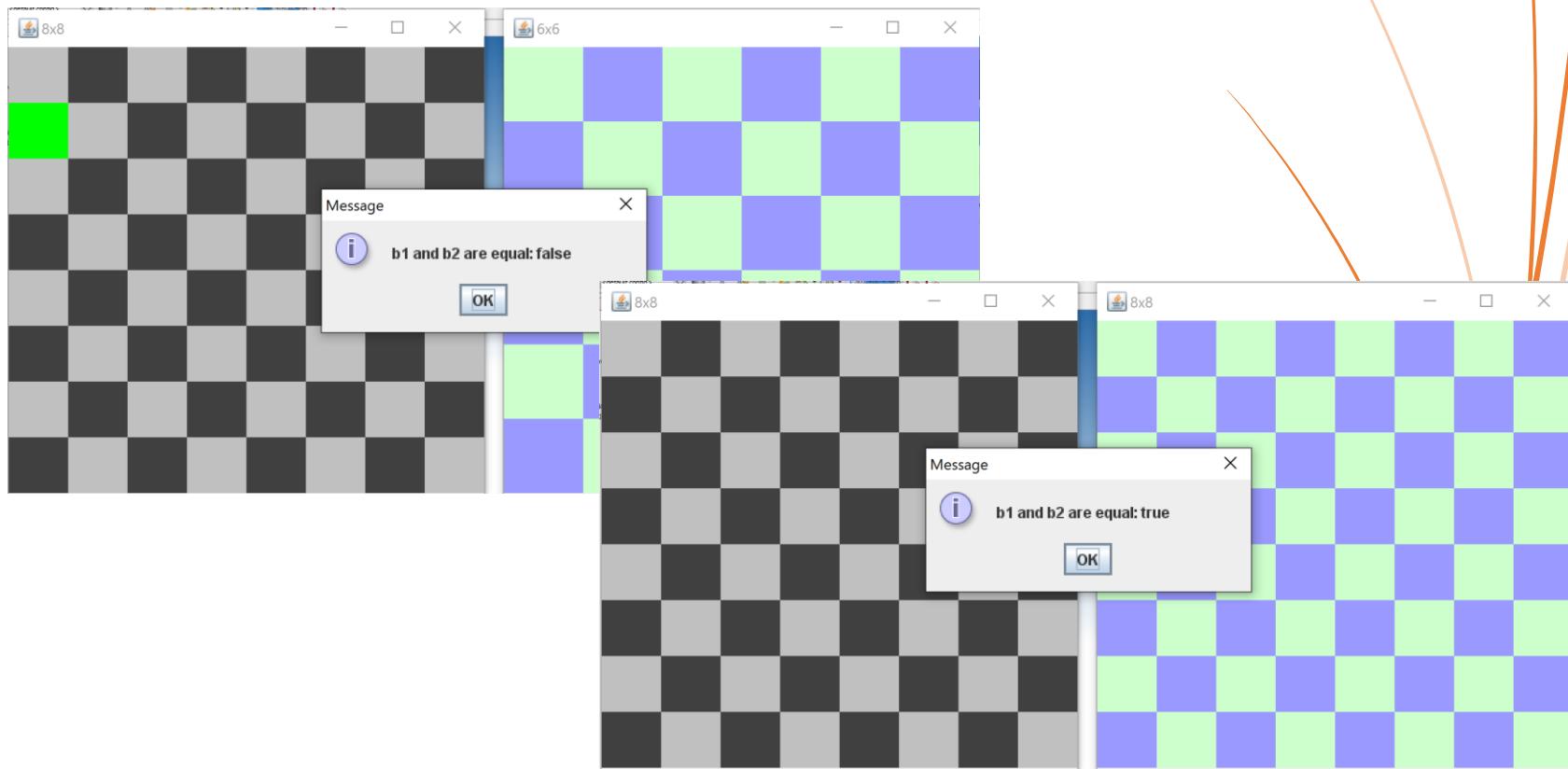
- There are two types of publication:
 - Book, printed and electronic
 - Journal, printed and online
- All publications have:
 - a title
 - a publisher
 - an ISBN
 - a discount that depends on its format
 - Printed → 0%
 - Electronic book → 10%
 - Online Journal → 5%
- Books also have an author

Problem 1 UML class diagram



Problem 2 (examine in your own time)

- Write a Swing application that displays two games boards and reports on whether they are equal or not



Overriding methods inherited from the Object class

Lecture notes

In these lecture notes

We will:

- Introduce more advance OO concepts
 - Override hashCode(), equals(), and toString() methods from the Object class
 - Abstract classes and methods
 - final keyword
 - Inner classes (static inner class, local class, nested class, and anonymous inner class)
 - Enumerated types

The purpose of the Object class

- Object is the superclass of all other Java classes
 - Defined in the package **java.lang**
 - A class that does not use extends automatically extends Object
 - An instance of any class is an instanceof Object
- Some inherited methods of Object:
 - String `toString ()`
 - boolean `equals (Object obj)`
 - int `hashCode ()`
 - Object `clone ()`
 - Class `getClass ()`
 - void `finalize ()`

<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

Overriding Object's methods

- `toString()`
 - We saw how to do this last week
- `equals()`
 - Override this when it is necessary to compare objects for equality
- `hashCode()`
 - Override this to ensure that two objects that are equal have the same hashCode

Overriding equals ()

- `equals ()` tests whether two references are to instances that can be considered equal
 - We must decide how to define “equal” for a class of objects
- The signature of the method is:
 - `public boolean equals (Object obj)`
 - Not using `Object` as the parameter type is a common mistake

Overriding equals ()

- Equal objects
 - `obj1.equals(obj2)` is true if the data values in `obj1` and `obj2` are considered equal
 - The objects referred to by `obj1` and `obj2` may be different
- Beware!
 - The `==` operator tests if two object references are to the same object
 - `obj1 == obj2` is true if and only if `obj1` and `obj2` refer to the same object

Overriding equals ()

- Many methods in the Java library assume that equals () is well-defined
 - Take care with your definition of equality
- The Java Language Specification imposes these requirements on equals ()
 - Reflexive
 - obj1.equals(obj1) must return true
 - Symmetrical
 - if obj1.equals(obj2) returns true
then obj2.equals(obj1) must return true
 - Transitive
 - if obj1.equals(obj2) and obj2.equals(obj3)
both return true
then obj1.equals(obj3) must return true

Overriding hashCode ()

- The hash code of an object is an integer that is used to store and locate an object in a hash set
 - Hash sets are part of the Java Collections Framework
 - More about this later in the module
- The signature of the method is:
 - `public int hashCode ()`
- The method returns a hash code value for the object
 - The method in the `Object` class returns the internal memory address of the object

Overriding hashCode ()

- equals () and hashCode () should be overridden together
- If two objects are equal their hash codes must be the same
 - But if two objects have the same hash code they do not have to be equal
 - Hash codes don't have to be unique
- Hash codes are often used in storing and retrieving objects
 - For example, a hash map

Overriding hashCode ()

- Most of the classes in the Java API implement the hashCode () method
 - In the Integer class, hashCode () returns its int value
 - In the Character class, hashCode () returns the Unicode of the character
 - In the String class, hashCode () returns
 - $s_0 * 31^{(n-1)} + s_1 * 31^{(n-2)} + \dots + s_{n-1}$
 - Where s_i is s.charAt(i)

Abstract classes and methods

Lecture notes



Abstract classes

- Sometimes, it is not possible to write all the functionality of a class
 - e.g. The need for a method could be obvious, but what to write for the method body might not be clear
- Such classes should be made abstract, which means that the compiler will prevent them from being instantiated
 - i.e. cannot create objects of an abstract class
- This ensures there are never any incomplete objects

To make a class abstract

- Making a class abstract is very easy

```
public abstract class MyClass  
{  
}
```

- Adding this one word means that `MyClass` objects cannot be created directly

To make a method abstract

- Making a method abstract is also very easy

```
public abstract int getWidth();
```

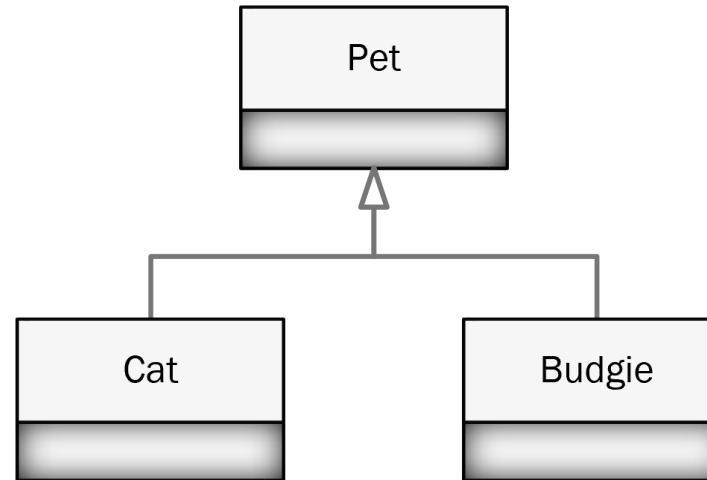
NB: There is no body for an abstract method, hence the semi-colon

- A class with at least one abstract method is incomplete, so it must be declared abstract
- Subclasses of an abstract class must:
 - Override the abstract method and provide a method body
 - Or, be declared abstract

Pet example

- Consider the Pet application

- A Cat is a pet
- A Budgie is a pet



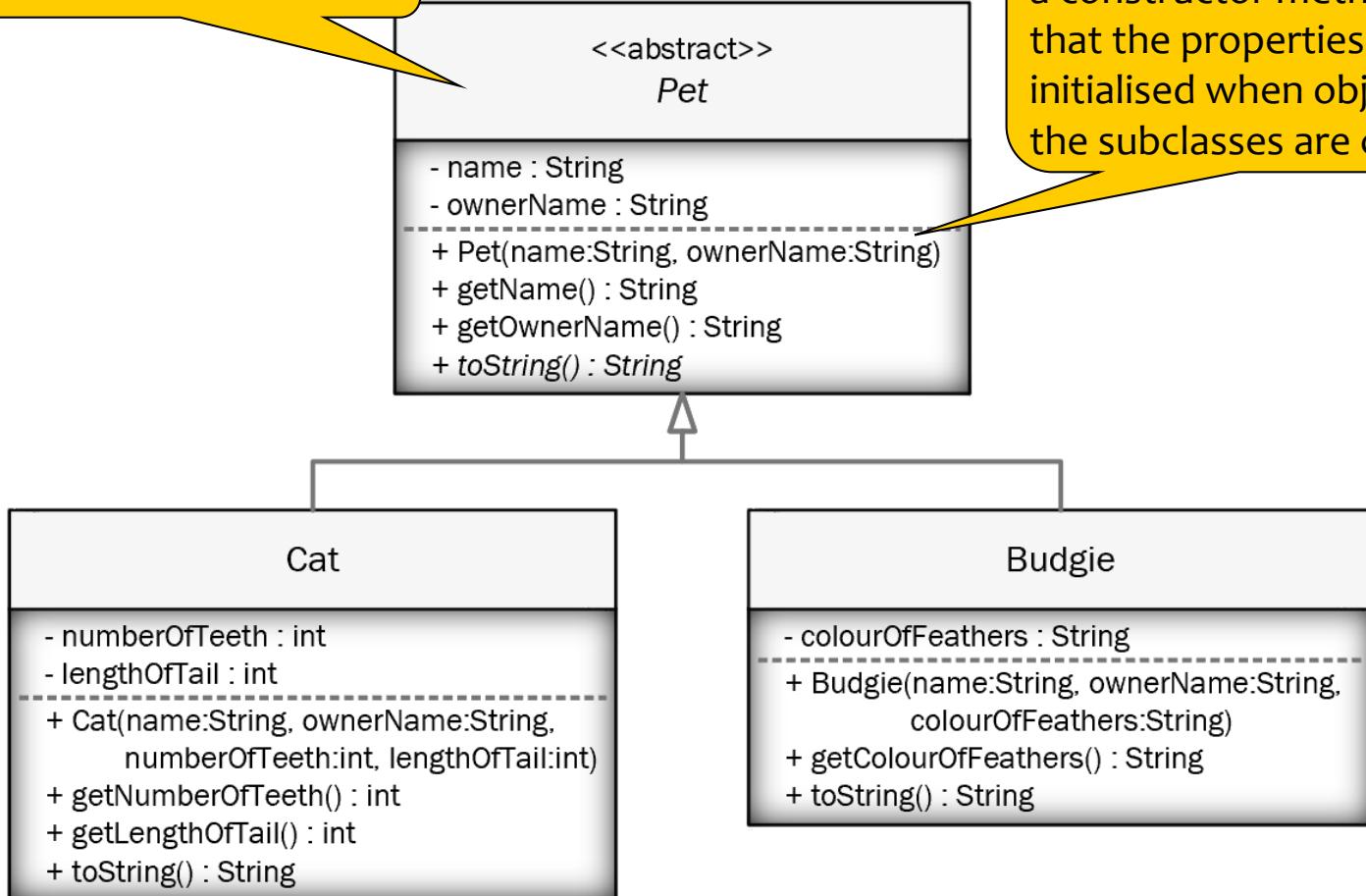
- If
 - the Pet class is incomplete,
or
 - creating a Pet object does not make sense

Then

Make the Pet class abstract

Abstract classes in UML

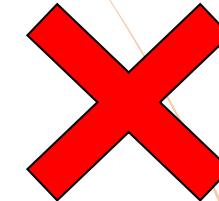
Names of abstract classes and methods are written in italics



The effects of making a class abstract

- Abstract classes cannot be instantiated directly

```
private Pet p = new Pet("Fluffy");
```



- Abstract classes can be instantiated indirectly
 - i.e. via their non-abstract subclasses

```
private Pet p = new Cat("Hisser");
```



```
private Pet p = new Budgie("Whistler");
```



Activity



Coding

- Review the example code and watch out for:
 - the effect of making `Pet` abstract
 - how polymorphism can still be used with abstract classes
- If you have any questions, ask

Project: `AbstractPet`

The final keyword

Lecture notes



The final keyword

- Used in several places to prevent changes after definition
- A final variable cannot be changed after initialisation
 - Initialise at the declaration or in the constructor method
- A final method cannot be overridden by subclasses
- A final class cannot be subclassed

Inner classes

Lecture notes

Inner (or nested) classes

- A class can contain members that are
 - Variables
 - Methods
 - Other (inner) classes
- An inner class can have public, protected, default, or private access
 - Just like any other member
- Code in the inner class has access to all members of the outer class
 - Even private members!

Why use inner classes?

- To logically group classes that are only used together
 - Very useful for composition relationships
 - The Eye class is only used in the context of a Face class
- To increase encapsulation
 - Eye and Face can access each other's member variables
 - even if they are private to all other classes
 - If the inner class is private, no other classes will even know it exists

Why use inner classes?

- Easier to read and maintain code
 - Inner classes are usually small and have a specific purpose
 - Locating them within the outer class makes them easier to find

Activity



Coding

- Review the example code and watch out for:
 - the use of an inner class in Face
- If you have any questions, ask

Project: InnerClassExample

Static vs non-static inner classes

- A static inner class can only access static members of the outer class
 - A non-static inner class can access all members of the instance of the outer class
- If not private, a static inner class is accessed via the outer class
 - `OuterClass.StaticInnerClass innerObject
= new OuterClass.StaticInnerClass();`
- If not private, a non-static inner class is accessed via an instance of the outer class
 - `OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject
= outerObject.new InnerClass();`

Local classes

- A class defined within a block of Java code rather than as a class member (inner class)
 - A class declared within a method is an example of a local class
- A local class is accessible only within the block in which it is declared

Activity



Coding

- Review the example code and watch out for:
 - the use of a local class in Crowd
- If you have any questions, ask

Project: LocalClassExample

Anonymous inner classes

- Declare and instantiate a class at the same time
- They are like local classes, except that they do not have a name
- Are useful if only one instance is needed
- Can either implement an interface or extend a class
 - Can override existing superclass methods, and implement abstract ones
 - Cannot define any new methods

Activity



Coding

- Review the example code and watch out for:
 - the use of an anonymous inner classes in ExampleFrame.setupButtonsPanel()
 - each button is given its own instance of the anonymous inner class
- If you have any questions, ask

Project: AnonymousInnerClassExample

Enumerated types

Lecture notes

Sets of values

- Sometimes sets of values are needed
 - Days of the week
 - Months of the year
- How to represent these sets?
 - Strings?
 - “Monday”, “Tuesday”, etc.
 - Does not prevent invalid values (e.g. “Middlesday”)
 - Integers?
 - 1, 2, 3, 4, 5, 6, 7
 - Which integer represents Monday? Why?

Enumerated types

- It can be useful to represent a fixed set of constants as a data type called an enumeration
 - Days of the week
 - Months of the year
- The enumeration defines a set of valid values
- Each value has an associated integer position or ordinal
 - Starts at zero
 - DayOfWeek.MONDAY.ordinal() returns zero
 - DayOfWeek.WEDNESDAY.ordinal() returns 3
 - Do not normally use the ordinal explicitly

Enumerated types

- Creating an enumerated type in Java

```
• public enum DayOfWeek  
{  
    MONDAY, TUESDAY, WEDNESDAY,  
    THURSDAY, FRIDAY, SATURDAY,  
    SUNDAY;  
}
```

- This is a public declaration, so it goes in its own file
 - e.g. DayOfWeek.java

Enumerations are data types

- An enum can be used like other data types
 - e.g. a reference to one of the enum constants:
`DayOfWeek day = DayOfWeek.MONDAY;`
- The variable `day` can refer to different values within the same enum
 - `day = DayOfWeek.FRIDAY;`
- Note the way in which the constants are accessed
 - Like public static constants in a class
 - Use the name of the enum rather than the name of a class

Enumerations in selections

- The value of a variable of type enum can be checked using standard selection

- e.g.

```
if (day == DayOfWeek.SATURDAY)
```

- e.g.

```
switch (day) {  
    case DayOfWeek.SATURDAY:  
        // do something  
        break;  
    case DayOfWeek.SUNDAY:  
        // do something  
        break;  
    default:  
        // not a handled case  
}
```

The enum class

- An enum implicitly extends `java.lang.Enum`
<http://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html>
- The constructor for an enum must not be public
 - An enum constructor should not be invoked from outside the type

The enum class

- The compiler automatically adds some useful static methods
 - `public static DayOfWeek valueOf(String name)`
returns the enum constant with the specified name
e.g. `DayOfWeek.valueOf("MONDAY")`
returns `DayOfWeek.MONDAY`
 - `public static DayOfWeek[] values()`
returns an array containing all the constants of this enum type, in the order they are declared
 - **use this to iterate through all the constants**

```
for (DayOfWeek day : DayOfWeek.values())  
{  
}
```

Advanced features of enums

- The enum body can contain methods and other fields
 - they must come after the declaration of the constants

```
public enum Month {  
    JANUARY(31),  
    FEBRUARY(28),  
    MARCH(31),  
    ...  
    DECEMBER(31);  
  
    private final int numDays;  
  
    // note private constructor  
    private Month(int nDays) {  
        numDays = nDays;  
    }  
  
    public int getNumDays() {  
        return numDays;  
    }  
}
```

This enum contains a member variable, numDays, which is initialised in the private constructor.

The constructor is called in the declaration of each constant

The method, getNumDays(), returns the value of numDays for the constant

Activity



Coding

- Review the example code and watch out for:
 - the use of the enum in the `main()` method
- If you have any questions, ask

Project: `EnumExample`

Java IO fundamental concepts

Lecture slides

In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write a Java program that demonstrates the concepts in the lecture notes
 - Read/write data from/to the console
 - Read/write text from/to files
 - PrintWriter methods
 - Exceptions
 - Use JSON when storing objects in files
 - Clone objects
 - Make immutable objects
- Ask questions about anything you don't understand

Problem

- Write an application that allows the user to
 - view a list of books
 - create a new book
 - borrow a copy of a book
 - return a copy of a book
- The application must
 - load Book and Copy objects from file when it starts
 - save Book and Copy objects to file when it closes

Fundamentals of Java I/O

Lecture notes

In these lecture notes

We will:

- Look at the fundamentals of Java I/O
 - Read data from and write data to the console
 - PrintWriter in the java.io package
- Review exceptions
- Reading and writing text with files
- Use JSON when storing objects in files
- See how to clone objects
- Introduce immutable objects

Java's standard output stream

- Java's standard output stream provides access to the console window
 - Called `System.out`
- `System.out` is an instance of `PrintWriter`, which has methods for outputting text
 - Unformatted output
 - `print()`
 - `println()`
 - Formatted output
 - `printf()`

The PrintWriter's methods

Output method	Comments
print ()	output parameter immediately following the previous output eg. System.out.print ("Hello");
println ()	output the parameter followed by new line eg. System.out.println ("Hello");
printf ()	output the data formatted as specified by the first parameter lower case L, not the digit 1

Unformatted output examples

- All examples assume name = "Bob"

```
System.out.print("Hello ");  
System.out.print(name);
```

Hello Bob

```
System.out.println("Hello ");  
System.out.println(name);
```

Hello
Bob

```
System.out.print("Hello " + name);
```

Hello Bob

```
System.out.print("Hello ");  
System.out.print("name");
```

Hello name

name in
quotes

Unformatted output examples

- Examples assume $i = 3$, $x = 6.2$

```
System.out.print("Hello " + i + x);
```

Hello 36.2

converts each variable to a string

```
System.out.print("Hello " + (i + x));
```

Hello 9.2

does calculation then concatenates

```
System.out.print("Hello " + "i + x");
```

Hello i + x

calculation in quotes is not a calculation!

Formatted output

- `printf()` has two parts:
 1. Format string containing **placeholders** for data items (shown by % in format string)
 2. list of **items** to be output separated by commas
 - number of placeholders and items must be same
- Placeholder specifies:
 - Type
 - d: decimal integer
 - f: floating point
 - s: string
 - Optional number of columns
 - Justification (default is right-justified)

Formatted output examples

```
System.out.printf("%d", i);
```

3

```
System.out.printf("%f", x);
```

6.200000

defaults to 8 columns for value

```
System.out.printf("%s", "Bob");
```

Bob

```
System.out.printf("%s adds %d to %1.1f",
                  name, i, x);
```

Bob adds 3 to 6.2

three placeholders;
so three data items

Formatted output examples

```
System.out.printf("%8s", "Bob");
```

— — — — Bob

specifies minimum width of field

```
System.out.printf("%-8s", name);
```

Bob — — — —

changes alignment using -

```
System.out.printf("%7.3f", x);
```

— — 6.200

Fixes the number of decimal places

Tab and newlines

- New lines and tab characters can be included in print statements

- \t tab to next print area
- \n output new line
- \r\n output return character with new line
(needed for some applications like Notepad)

```
System.out.print("\nHello\n" + name);
```

Hello
Bob

outputs newline before and after

```
System.out.printf("\t%d\n\t%3.1f\n",
                  i, x);
```

----- 3
----- 6.2

uses tabs and newlines to format

Java's standard input stream

- Java's standard input stream provides access to data via the keyboard
 - Called `System.in`
- Objects of the `java.util.Scanner` class can be used to read `System.in`
- This class has several methods to convert stream values into different types

The Scanner's input methods

Input method	Comments
nextInt()	input integer eg. int i = kybd.nextInt();
nextDouble()	input double eg. double x = kybd.nextDouble();
next()	input string – up to next space or Enter eg. String s1 = kybd.next();
nextLine()	input string – up to next Enter eg. String s2 = kybd.nextLine();

Scanner example

Inputting numbers with Scanner

- White space delimits input
- `nextInt()` and `nextDouble()` read numbers up to, but not including, next white space
- Input data can be separated by white space

```
3 4 5<enter>
```

```
3<enter>
4<enter>
5<enter>
```

```
3 4<enter>
5<enter>
```

- If wrong type of data is input, program will throw an exception and crash

Exceptions

Lecture notes



Errors at runtime

- During execution of Java programs, errors can occur
- These runtime errors can occur for all kinds of reasons:
 - Input or output problems (e.g. file not found)
 - Network problems (e.g. connections lost)
 - Memory problems (e.g. stack overflow)
 - Calculation problems (e.g. divide by zero)
 - Data type problems (e.g. input string instead of integer)
 - Many other reasons

Generating exceptions

- Many programming languages generate exceptions when something goes wrong during execution
- When a problem occurs, an exception is **thrown** by the runtime environment
- We must decide whether to **catch** the exception, or let it be handled by the runtime environment

Some common exceptions in Java

- Dividing a number by zero
 - ArithmeticException
- Accessing an array element that doesn't exist
 - ArrayIndexOutOfBoundsException
- Reading data of the wrong type with a Scanner object
 - InputMismatchException
- A error when converting a string to an integer
 - NumberFormatException
- Opening a data file that doesn't exist
 - FileNotFoundException

Exception examples

```
import java.util.*;
public class ExceptionExamples
{
    public static void main(String[] args)
    {
        Scanner kybd = new Scanner(System.in);

        System.out.print("Enter first integer: ");
        int num1 = kybd.nextInt();

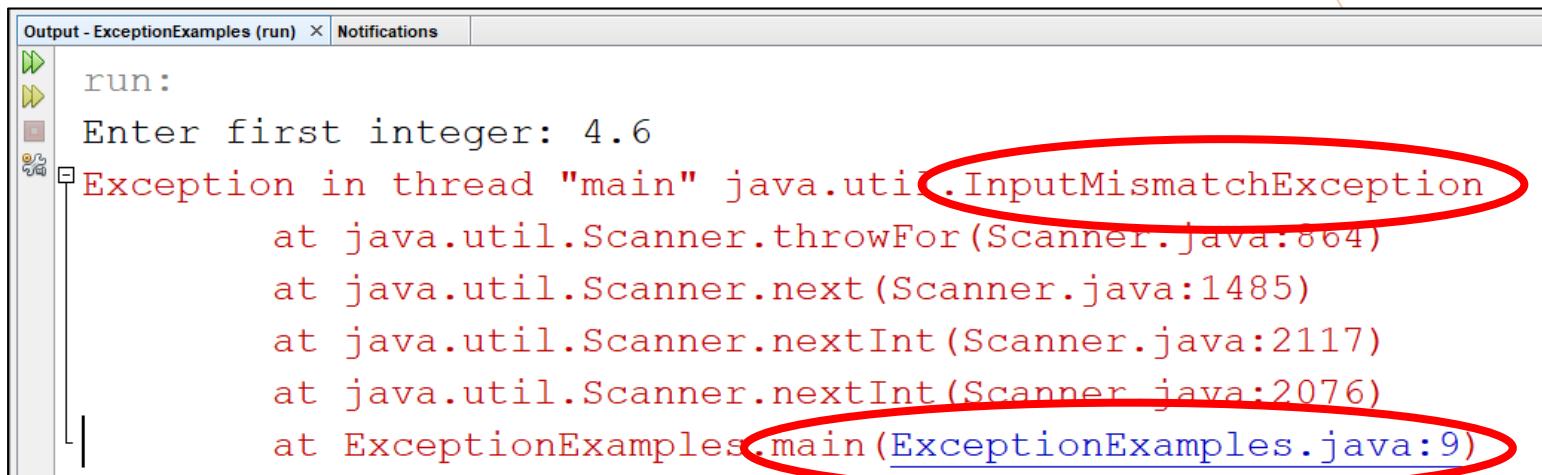
        System.out.print("Enter second integer: ");
        int num2 = kybd.nextInt();

        int result = num1 / num2;
        System.out.println(num1 + " / " + num2 +
                           " = " + result);
    }
}
```



Exceptions at run-time

- When an exception occurs, the run-time system displays details:



The screenshot shows an IDE's output window titled "Output - ExceptionExamples (run) X Notifications". It displays the following text:

```
run:  
Enter first integer: 4.6  
Exception in thread "main" java.util.InputMismatchException  
        at java.util.Scanner.throwFor(Scanner.java:864)  
        at java.util.Scanner.next(Scanner.java:1485)  
        at java.util.Scanner.nextInt(Scanner.java:2117)  
        at java.util.Scanner.nextInt(Scanner.java:2076)  
        at ExceptionExamples.main(ExceptionExamples.java:9)
```

Two specific lines in the stack trace are circled in red: "java.util.InputMismatchException" and "ExceptionExamples.main(ExceptionExamples.java:9)".

- Name of exception:
 - InputMismatchException
- Where the exception occurred:
 - line 9 in main method of ExceptionExamples class

JRE handles exceptions

- Leaving the Java Runtime Environment (JRE) to handle exceptions is not desirable
 - Users do not want to see the stack trace
- It is better to handle exceptions in the program
 - Provide more helpful messages

Program handles exceptions

- Can avoid program crash by **handling** the exception
- Code that could cause an error is put in a `try` block
- Code to handle the exception is put in a `catch` block
- The `catch` block is executed immediately an exception occurs
- Multiple `catch` blocks can deal with different exceptions that may occur
- An optional `finally` block is executed whether or not exception occurs
- The `catch` block is passed details of the exception that occurred
- The `getMessage()` method can be used to display information about the exception

Activity



Video

- Watch an explanation of how try..catch works:
 - <http://youtu.be/-ptJ8xC1AEY>

(duration – 07:16)

How `try...catch` structures execute

- The `try` block is executed
- If no exceptions occur, the `catch` blocks are not executed
- If exception occurs, the `try` block is abandoned
- First matching `catch` block is executed
- Code after the `try...catch` structure is executed
- If no `catch` blocks match, the next block out is searched

Continuing processing if possible

- Sometimes, we may wish to continue processing after an exception
 - Prompt user for integer
 - Output error if wrong type
 - Continue until valid data input
- Put input routine inside a method

Exception examples

```
public static int inputValidNum()
{
    Scanner kybd = new Scanner(System.in);
    boolean OK = false;
    int num = 0;

    while (!OK) {
        try {
            System.out.print("Number: > ");
            num = kybd.nextInt();
            OK = true;
        }
        catch (InputMismatchException e) {
            System.out.println("Invalid input, try again: ");
        }
    }
    return num;
}
```

The finally block

- The try structure can also have a finally block
 - After try block and any catch blocks
- Executed whether exception is thrown or not

```
try
{
    //...
}
catch (Exception e)
{
    //...
}
finally
{
    //...
}
```

Throwing an exception

- Use an `Exception` object to throw exceptions
 - If an error is detected, create a new `Exception` object and throw it

```
if (age < 0)
{
    throw new Exception("Age must be at least zero");
}
```

- Code can then be written to catch the exception

More to come

- We will learn more about exceptions next week

Reading and writing text with files

Lecture notes

Files

- Files are useful for storing data between executions of a program:
 - Write data to a file before closing the application
 - Read data from the file when re-opening the application
- Thus, data persists (continues to exist) between executions
 - A very important principle for almost every application in industry

Outputting data to file

- Use PrintWriter instances to write text to a file

```
// code to give values to num, salary and name

try
{
    PrintWriter out = new PrintWriter("data.txt");
    out.printf("%d %1.2f %s\n", num, salary, name);
    out.close();
}
catch (IOException ioe)
{
    System.out.println("ERROR: " + ioe.getMessage());
}
```

Inputting data from file

- Use Scanner instances to read from a file

```
try
{
    Scanner in = new Scanner(new File("data.txt"));
    int num = in.nextInt();
    double salary = in.nextDouble();
    String name = in.nextLine();
    in.close();
}
catch (IOException ioe)
{
    System.out.println("ERROR: " + ioe.getMessage());
}
```

Writing objects to a text file

- Output the values from an object

```
// code to set values in the properties of object p

try
{
    PrintWriter out = new PrintWriter("data.txt");
    out.printf("%d %1.2f %s\n",
               p.getNum(),
               p.getSalary(),
               p.getName());
    out.close();
}
catch (IOException ioe)
{
    System.out.println("ERROR: " + ioe.getMessage());
}
```

Reading objects from a text file

- Input the values for an object

```
try
{
    Scanner in = new Scanner(new File("data.txt"));
    Person p = new Person();
    p.setNum(in.nextInt());
    p.setSalary(in.nextDouble());
    p.setName(in.nextLine());
    in.close();
}
catch (IOException ioe)
{
    System.out.println("ERROR: " + ioe.getMessage());
}
```

Pitfalls of persisting objects to file

- A different order of reading and writing values
 - Write num, salary, name
 - Read salary, num, name
- Every application must use the same order
 - Always!
- The order is not always obvious
 - Which of these data items is the first name?
 - 12 13500.34 Henry James

Associating data and fields

- The ordering problem can be solved by linking the value to the field

- e.g. Write <name:value> pairs to file

num:12

salary:13500.34

lastName:Henry

firstName:James

- The order is clear

- And the order is unimportant

num:12

firstName:James

lastName:Henry

salary:13500.34

JSON

- This is the principle of JSON
 - JavaScript Object Notation
- JSON is “a lightweight data-interchange format”
 - “It is easy for humans to read and write.”
 - “It is easy for machines to parse and generate.”
(<http://www.json.org/>)
- JSON is independent of programming languages
 - e.g. Write in Java, read in C#

JSON

- The JSON syntax is simple

- See <http://www.json.org/>



- JSON constructs

- An object

```
{"num":12, "firstName":"James",  
"lastName":"Henry", "salary":13500.34}
```

Delimited by braces;
Unordered comma-separated
list of name:value pairs

- A collection of values

```
[ value1, value2, value3 ]
```

Delimited by square brackets;
Ordered comma-separated
list of values

JSON examples

```
{"id":102,  
"name":"Graham Mansfield",  
"dept": "Computing"}
```

Could be read as three separate variables, or a single object with three fields

```
[ 2, 4, 6, 8 ]
```

A collection of integers;
Could be read as an array, an ArrayList, etc.

```
[ { "id":1, "name":"Jan"},  
  { "id":2, "name":Fred} ]
```

A collection of objects

```
{"id":18,  
"name":"Julie",  
"favTVchannels": [2, 5, 12] }
```

An object containing a collection

JSON examples

```
{ "bandName" : "The Beatles",
  "members" :
  [
    {"id":1, "name" : "John Lennon"},  

    {"id":2, "name" : "Paul McCartney"},  

    {"id":3, "name" : "George Harrison"},  

    {"id":4, "name" : "Ringo Starr"}  

  ]  
}
```

An object containing other objects

Spacing used to aid readability for humans; it makes no difference to the computer

This could be read as a Band object containing a collection of references to four Person objects

Constructing JSON strings in Java

- The hard way
 - Write code to take each field and add a corresponding name : value pair to a string
- The easy way
 - Use the GSON library
- GSON: an open-source Java library from Google
 - Available from:
<https://search.maven.org/artifact/com.google.code.gson/gson/2.8.6/jar>
 - Download the latest available version

Activity



Coding

- Review the example code and watch out for:
 - how easily an object is converted to a JSON string using GSON
 - how easily a JSON string is converted to an object using GSON
- If you have any questions, ask

Project: JSONexample

Storing JSON in files

- Very simple!
- Write a JSON string to file
 - Construct a JSON string from one or more objects using GSON
 - Use a `PrintWriter` object to write the JSON string to file
- Read a JSON string from file
 - Use a `Scanner` object to read a JSON string
 - Construct one or more objects from the string using GSON

Activity



Coding

- Review the example code and watch out for:
 - how a JSON string is written to file
 - how a JSON string is read from file
- If you have any questions, ask

Project: JSONwithFile

Beware bi-directional association

```
public class MyA
{
    private MyB b;

    public MyB getB()
    {
        return b;
    }

    public void setB(MyB b)
    {
        this.b = b;
        b.setA(this);
    }
}
```

```
public class MyB
{
    private MyA a;

    public MyA getA()
    {
        return a;
    }

    public void setA(MyA a)
    {
        this.a = a;
    }
}
```

- What happens when a JSON string is created?

Beware bi-directional association

- When an object of MyA and an object of MyB hold a reference to each other...
 - GSON will produce

```
{ "b":  
  { "a":  
    { "b":  
      { "a":  
        { and so on until stack overflow }  
      }  
    }  
  }  
}
```

transient keyword

- Using `transient` will prevent the variable from being added to the JSON string
 - and stop the infinite recursion

```
public class MyA
{
    private MyB b;

    public MyB getB()
    {
        return b;
    }

    public void setB(MyB b)
    {
        this.b = b;
        b.setA(this);
    }
}
```

```
public class MyB
{
    private transient MyA a;

    public MyA getA()
    {
        return a;
    }

    public void setA(MyA a)
    {
        this.a = a;
    }
}
```

Other applications of JSON

- Web applications
 - A more concise alternative to using XML between client and server
 - e.g. JavaScript, AJAX, JQuery, Node.js
 - Passing data between client and web service
- Passing data between different applications
 - Over a network
 - Via files

Cloning objects

Lecture notes

The `clone()` method in Object

- Assigning one object to another makes a copy of the reference rather than a copy of the object

```
Person p1 = new Person("Fred");  
Person p2 = p1;
```

- `p1` and `p2` now refer to the same object
- The `Object` class has a method called `clone()`
 - It makes a copy of the object
 - It results in a second object with distinct identity but equal contents

```
p2 = (Person) p1.clone();
```

Cloning an object

- In general, a clone method should satisfy:
 - `obj1.clone() != obj1`
 - `obj1.clone().equals(obj1)`
 - `obj1.clone.getClass() == obj1.getClass()`
- The `clone()` method of `Object` is protected to prevent its direct use
- To override `clone()` in a subclass
 - Implement the `Cloneable` interface
 - Write an overriding public `clone()` method

The Cloneable interface

- The Cloneable interface does not specify any methods
 - It is a tagging interface
 - It can only be used to test whether an object implements it

```
if (obj instanceof Cloneable) ...
```
- An attempt to clone an object that is not an instance of a Cloneable class causes a CloneNotSupportedException

Overriding the `clone()` method

- The signature of the method is:

```
protected Object clone()  
throws CloneNotSupportedException
```

- Note the return type is `Object`

- Usually, the returned object will be type cast

Activity



Coding

- Review the example code and watch out for:
 - how a shallow clone works
 - The memory addresses of the objects before and after cloning
- If you have any questions, ask

Project: ShallowCloneExample

Activity



Coding

- Review the example code and watch out for:
 - how a deep clone works
 - The memory addresses of the objects before and after cloning
- If you have any questions, ask

Project: DeepCloneExample

Mutable and immutable objects

Lecture notes

Mutable vs Immutable objects

- The values of a **mutable** object can change after it has been instantiated

```
Person p1 = new Person("Fred", 21);  
p1.setAge(23);
```

- The value of age in p1 has changed
- The values of an **immutable** object cannot change after it has been instantiated
 - There are no mutator (set) methods
 - To “change” the values of an object, create a new object

```
Person p1 = new Person("Fred", 21);  
p1 = new Person("Fred", 23);
```

Why make an object immutable?

- When changes to a shared object would be harmful to other areas of an application
- To prevent changes to the state of an object
- To guarantee a “thread-safe” object when sharing an object between multiple threads
 - More about threads later in the module

How to make an object immutable

- Declare all member variables `final`
 - To prevent changes after initialisation
- Provide constructors that initialise all member variables
- Do not write mutator (set) methods
- Make the class `final`
 - If you want to prevent subclasses that could weaken (or destroy) the immutability

A class for immutable objects

```
public final class Person
{
    private final String name;

    public Person(String name)
    {
        this.name = name;
    }

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

More Java I/O

Lecture slides

In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write a Java program that demonstrates the concepts in the lecture notes
 - Subclasses of Reader and Writer for text I/O
 - Subclasses of InputStream and OutputStream for binary I/O
 - Serialization of objects
- Ask questions about anything you don't understand

Problem 1

- Modify the `JSONwithFile` example from Week 3 to demonstrate the difference between text I/O and binary I/O

Problem 2

- Modify the Lecture3Demo application so that objects are written and read in binary format

More about Java I/O

Lecture notes

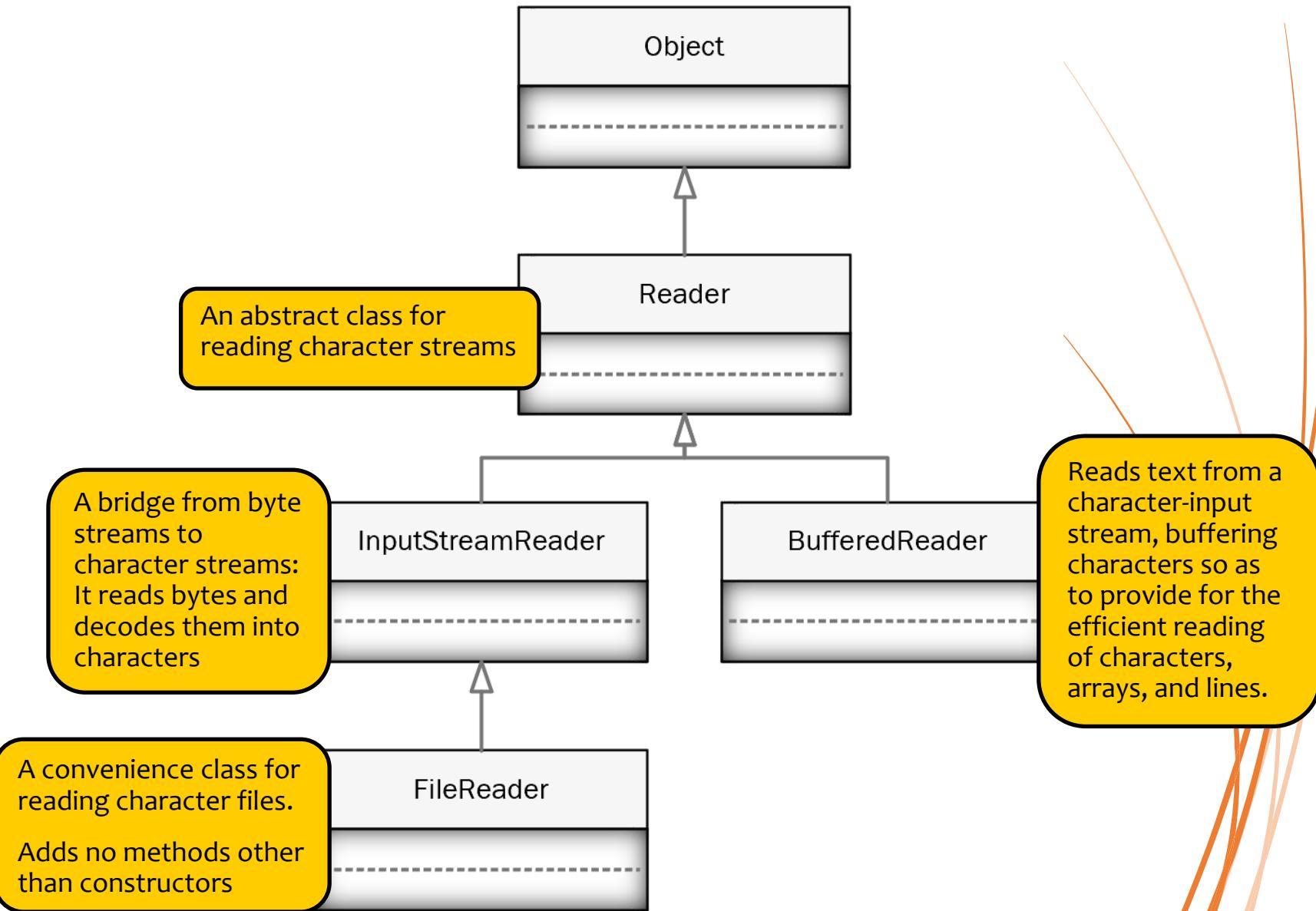


In these lecture notes

We will:

- Look at more features of Java I/O
 - Subclasses of Reader and Writer for text I/O
 - Subclasses of InputStream and OutputStream for binary I/O
 - FileInputStream and FileOutputStream
 - DataInputStream and DataOutputStream
 - BufferedInputStream and BufferedOutputStream
 - ObjectInputStream and ObjectOutputStream
- Serialization of objects

The character reader hierarchy



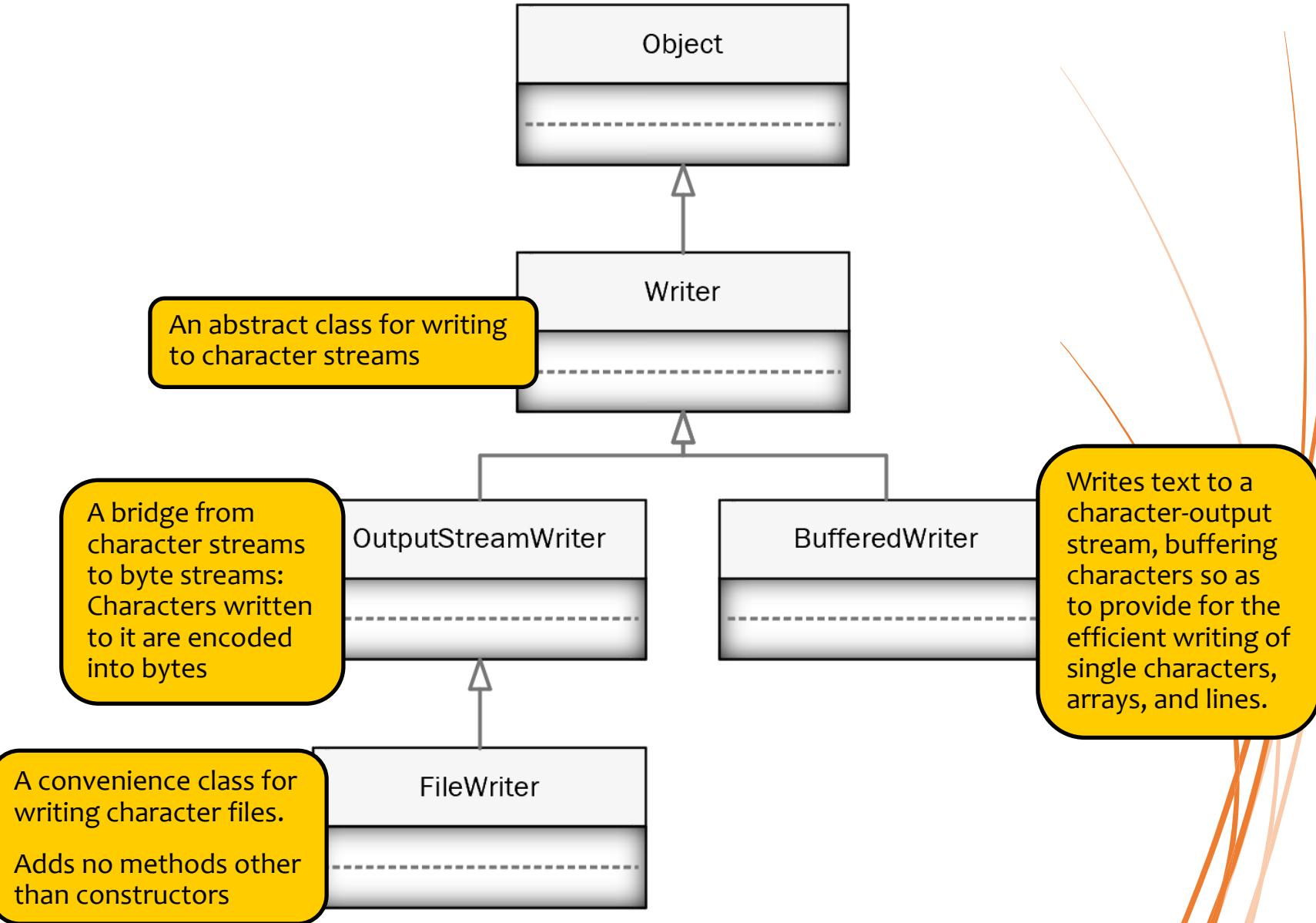
Using Reader classes

- Typically, objects of the Reader classes are wrapped inside each other to achieve the desired effect
 - For example

```
BufferedReader in  
    = new BufferedReader(  
        new FileReader("foo.in"));  
  
char c = in.read(); // read a single character  
  
String line = in.readLine(); // read a line of text  
  
Stream<String> lines = in.lines(); // read entire file  
  
in.close();
```

- More about streams later in the module

The character writer hierarchy



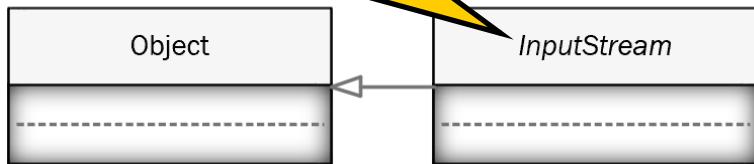
Using Writer classes

- Typically, objects of the Writer classes are wrapped inside each other to achieve the desired effect
 - For example

```
PrintWriter out
    = new PrintWriter(
        new BufferedWriter(
            new FileWriter("foo.out")));
// use the PrintWriter object as usual
out.close();
```

The binary input hierarchy

The superclass of all classes representing an input stream of bytes



Deserializes primitive data and objects previously written using an `ObjectOutputStream`

`FilterInputStream`

`ObjectInputStream`

Obtains input bytes from a file
It is intended for reading streams of raw bytes such as image data

`FileInputStream`

`DataInputStream`

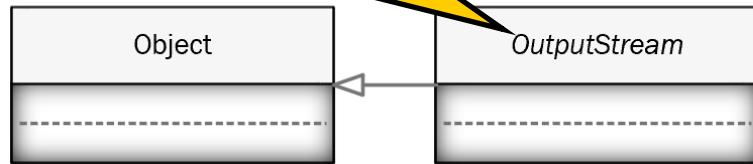
`BufferedInputStream`

Reads primitive Java data types from an underlying input stream that was previously written by a `DataOutputStream`

Adds buffering functionality to another input stream

The binary output hierarchy

The superclass of all classes representing an output stream of bytes



Writes primitive data types and graphs of Java objects to an `OutputStream`

Only objects that support the `java.io.Serializable` interface can be written to streams

FileOutputStream

FilterOutputStream

ObjectOutputStream

DataOutputStream

BufferedOutputStream

Writes bytes to a file
It is intended for writing streams of raw bytes such as image data

Writes primitive Java data types output stream in a portable way (i.e. independent of the underlying operating system)

Adds buffering functionality to an underlying output stream

Using binary I/O classes

- Typically, objects of the various binary I/O classes are wrapped inside each other to achieve the desired effect
 - The object closest to the raw bytes is wrapped inside an object that adds a higher layer of processing
 - For example

```
DataInputStream inStream =  
    new DataInputStream(  
        new BufferedInputStream(  
            new FileInputStream("temp.dat") ) );  
  
DataOutputStream outStream =  
    new DataOutputStream(  
        new BufferedOutputStream(  
            new FileOutputStream("out.dat") ) );
```

Writing objects to binary streams

- Use an ObjectOutputStream
 - Call the `writeObject()` method
 - For example

```
Person p = new Person();  
...  
ObjectOutputStream output =  
    new ObjectOutputStream(  
        new FileOutputStream("person.dat"));  
output.writeObject(p)  
output.close();
```

- The data in the object `p` is written in binary format (bytes) not text

Reading objects from binary streams

- Use an `ObjectInputStream` to read binary data previously written by an `ObjectOutputStream`
 - Call the `readObject()` method
 - For example

```
ObjectInputStream input =
    new ObjectInputStream(
        new FileInputStream("person.dat"));
Person p = (Person)input.readObject();
input.close();
```

- The bytes representing `p` are read from the stream and converted to a `Person` object

Objects in binary streams

- To read and write objects in binary streams, the objects must be serializable
 - See notes 4b

Serializing objects

Lecture notes

Definition

- Serializing an object means to convert it into a stream so it can be persisted in a storage medium (e.g physical file, database)
- Deserializing an object means to convert it from a stream back into a copy of the original object
- To make a class of serializable objects, implement the `java.io.Serializable` interface

The Serializable interface

- The Serializable interface does not specify any methods
 - It is a tagging interface
 - It can only be used to test whether an object implements it

```
if (obj instanceof Serializable) ...
```
- An attempt to serialize an object that is not an instance of a Serializable class causes a NotSerializableException

Making a class serializable

```
public final class Person implements Serializable
{
    private final String name;

    public Person(String name)
    {
        this.name = name;
    }

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

Serialization process

- Information about the object is encoded in a byte stream
 - Class name and signature
 - Current value of all object attributes
 - Static attributes are not included because they belong to the class as a whole
 - If an attribute is an object reference, that object is also serialized and assigned a serial number
 - If there is later a reference to the same object, only its serial number is stored, the object is not duplicated

De-serialization process

- De-serializing an object reverses the process that serialized it
 - The object is reconstituted from binary data

Serializing object references

- If an object contains a reference to another object, that object must also be serializable
- Many Java classes implement Serializable
 - e.g. Date, Collection classes, and many more
 - Some classes such as threads and IO streams are not suitable for serialization
- Arrays and collections are serializable only if all their elements are serializable
 - Instead of saving each object separately, save the entire array or collection
 - Each element is automatically deserialized as the correct type

Transient data

- If some object attributes shouldn't be serialized, mark them with the keyword transient

```
private transient int currentIndex;
```

- the value of `currentIndex` won't be saved with the rest of the object
- When the object is deserialized, transient attributes are set to their default values
 - zero for int and other numeric types
 - null for objects
 - Useful for not serializing objects that do not implement `Serializable`
 - Need to store and restore their state separately

Generic classes

Lecture notes

In these lecture notes

We will:

- Look at how to create and use generic classes
- Collections
 - `ArrayList` and `HashMap`
 - `TreeSet` and `TreeMap`
 - `ArrayDeque`
- Defining ways to compare objects
 - `java.util.Comparator` and
`java.util.Comparable`

The need for a generic class



Coding

- Review the example code and watch out for:
 - how similar the two queue classes are
 - what is different about the two queue classes
 - run the application to see what happens
- If you have any questions, ask

Project: QueueExample

The need for a generic class

- In the QueueExample project, the only difference between the two queue classes is the data type of the items they accept
 - The code is conceptually identical
 - Any variation is simply because of the different data types
- To create another queue to handle Person objects, Vehicle objects, or any other kind of objects, would be just as similar

The need for a generic class

- This is wasteful coding!
- It would be better to have just one class that accepts all data types, but also enforces the correct type of data
 - i.e. a String queue does not accept integers
- This is the concept of generic classes

Declaring a generic class

- To declare a generic class, include the generic type after the class name

```
public class GenericQueueWithArray<T> { ... }
```

- <T> indicates a placeholder for a data type
 - Any letter can be used

Using a generic data type

- In the generic class, the generic type can be used instead of a named data type

```
private final T[] queue;  
public void enqueue(T item) ...  
public T dequeue() ...
```

Setting a generic data type

- When the generic class is used, the generic type is set

```
GenericQueueWithArray<String> strQ;  
GenericQueueWithArray<Integer> intQ;
```

- The compiler will...
 - set T to String for the strQ variable
 - set T to Integer for the intQ variable

Different types for different objects

- For `GenericQueueWithArray<String> strQ;`
- T becomes

```
private final String[] queue;  
public void enqueue(String item) ...  
public String dequeue() ...
```

- For `GenericQueueWithArray<Integer> intQ;`
- T becomes

```
private final Integer[] queue;  
public void enqueue(Integer item) ...  
public Integer dequeue() ...
```

Example of a generic class



Coding

- Review the example code and watch out for:
 - how generic queue class removes the need for multiple queue classes
 - how a generic array is created 😕
 - run the application to see what happens
- If you have any questions, ask

Project: GenericQueueExampleWithArray

Generic Collections

Lecture notes



Generic collections

- Java has a lot of collection classes
 - Generics were added to Java in version 5
- We will look at a few
 - **ArrayList and HashMap**
 - **TreeSet and TreeMap**
 - **ArrayDeque**
- They are all generic classes

ArrayList

- Uses an array as its underlying data structure
- An ArrayList ...
 - is dynamic in size (unlike an array)
 - holds only objects (not primitive types)
 - it implements the List interface
 - has methods to add and retrieve items
 - has many other methods

An ArrayList example



Coding

- Review the example code and watch out for:
 - how using an `ArrayList` instead of an array improved functionality
 - how an `ArrayList` is easier to create than a generic array 😊
 - run the application to see what happens
- If you have any questions, ask

Project: GenericQueueExampleWithArrayList

Another ArrayList example

- The Lecture4Demo2 project is another example of using an ArrayList

Iterating through an ArrayList



Coding

- Review the example code and watch out for:
 - how to use an `Iterator` to move through an `ArrayList` instead of using a `for` loop
 - see `viewListOfBooks()` and `findBook()`
 - run the application to see what happens
- If you have any questions, ask

Project: `ArrayListIterationExample`

Java Map and Set interfaces

- Map objects map keys to values
 - A map cannot contain duplicate keys
 - Each key can map to at most one value
 - A value can be retrieved using its key
 - Examples
 - HashMap, TreeMap and many others
- A Set object holds a collection of values
 - It cannot contain duplicate values
 - No order is imposed of the values in the set
 - Examples
 - HashSet, TreeSet and many others

HashMap

- Stores a key-value pair in a hash table
 - The hash code of the key indicates the location for storing the value
- To declare a HashMap object

```
HashMap<String, Book> books;
```

- The key is of type String (use book ISBN)
- The value is of type Book (use Book object)

HashMap

- To declare a HashMap object

```
HashMap<Integer, Copy> copies;
```

- The key is of type Integer (use copy id)
- The value is of type Copy (use Copy object)

A HashMap example



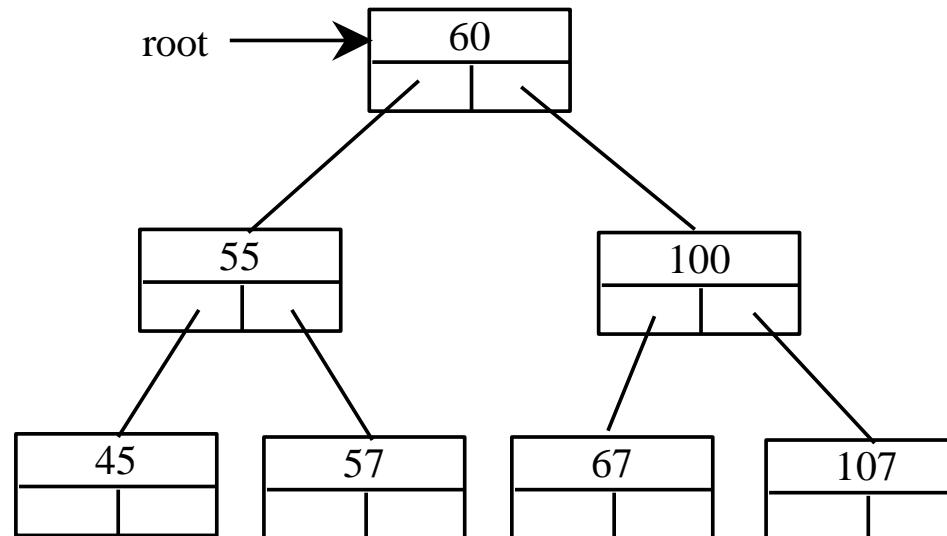
Coding

- Review the example code and watch out for:
 - how to use a `HashMap` instead of an `ArrayList`
 - run the application to see what happens
- If you have any questions, ask

Project: `HashMapExample`

TreeMap and TreeSet

- These use tree structures to hold their data
 - Implement either the Map or the Set interface (as their names suggest)
 - Useful for sorted collections
 - New elements are added in sorted order
 - Collection is traversed in sorted order



Implementations of Map, Set and others

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
Hashtable	x			No	No
TreeMap	x			Sorted	By natural order or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By natural order or custom comparison rules
LinkedHashSet		x		By Insertion order	No
ArrayList			x	By index	No
LinkedList			x	By index	No

More about custom comparison in the next notes



ArrayDeque

- Deque means “double-ended queue”
 - Pronounced “deck”
 - Values can be added to the front and rear
 - Values can be removed from the front and rear
- ArrayDeque is a resizable collection
 - Like ArrayList, it grows as required
 - It implements the Deque interface

More information

- Much more information about these classes (and others) is available at:
<https://docs.oracle.com/javase/8/docs/api/index.html>

Comparing objects

Lecture notes



Checking for equality

- One form of comparison is checking for equality
- Can use `equals()` to check objects for equality
 - Remember to override `hashCode()` whenever overriding `equals()`
- See Week 2 for more information on this

Comparing values

- When comparing values, A and B, there are three possibilities
 - $A < B$
 - $A = B$
 - $A > B$
- In Java, comparison produces three possible values
 - $A < B \rightarrow$ any negative integer
 - $A = B \rightarrow$ zero
 - $A > B \rightarrow$ any positive integer

Comparing strings

- The String class has a `compareTo()` method that makes a lexicographical comparison of two strings
 - `"A".compareTo("B")` → any negative integer
 - `"dd".compareTo("dd")` → zero
 - `"Q".compareTo("P")` → any positive integer
- The String class implements the `java.lang.Comparable` interface
 - Strings are objects
 - Objects can be compared if their classes implement the Comparable interface

The Comparable interface

- The `java.lang.Comparable` interface is a generic class and has only one method
 - `int compareTo(T obj)`
- `compareTo()` should return
 - zero if the result of `this.equals(obj)` is true
 - any negative integer if this object is less than (comes before) `obj`
 - any positive integer if this object is greater than (comes after) `obj`

Comparing objects

- How to determine if one object is less than or greater than another?
- With strings, it is easy
 - Which comes first in a dictionary?
- With Integer, it is easy
 - Numerical order
- With Person?
 - It depends on how the application would order the Person objects

Comparing objects

- The `compareTo()` method must examine the same attributes as `equals()`
- If a `Person` class has two attributes, name and age
 - The `equals()` method returns `true` if the two `Person` objects have equal name and age
 - The `hashCode()` method returns the same hashcode for two equal `Person` objects

Comparing objects

- The `compareTo()` method imposes a sort order
- If a Person class has two attributes, name and age
 - Which attribute is most important?
 - What is the order?
- The `compareTo()` method could order by
 - age, then name
 - or
 - name, then age

The Comparator interface

- The `java.util.Comparator` interface is a generic class and has several methods, one of which is
 - `int compare(T obj1, T obj2)`
- `compare()` should return
 - zero only if `obj1.equals(obj2)` is true
 - any negative integer if `obj1` is less than (comes before) `obj2`
 - any positive integer if `obj1` is greater than (comes after) `obj2`

The Comparator interface

- By writing classes that implement the Comparator interface, we can impose different comparison rules at different times
- For example,
 - A Comparator to compare two Person objects by name then age
 - Another Comparator to compare two Person objects by age then name

Reminder...

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
Hashtable	x			No	No
TreeMap	x			Sorted	By natural order or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By natural order or custom comparison rules
LinkedHashSet		x		By Insertion order	No
ArrayList			x	By index	No
LinkedList			x	By index	No



Ordered collections

- The table above shows that some collections are ordered by
 - Natural order
 - Numeric classes in ascending numeric order
 - Strings in ascending lexicographical order
 - Other classes as defined by the `compareTo()` method
 - Custom comparison rules
 - Defined by objects implementing the `Comparator` interface

Ordered collections of Person objects

- Natural order is defined by the Person class implementing `java.lang.Comparable`

- The `compareTo()` method defines the natural order

e.g. name, then age

```
new TreeSet<Person>()
```

- Customised order is defined by providing Comparator objects that implement the `java.util.Comparator` interface

- The `compare()` method defines the custom order

e.g. age, then name

```
new TreeSet<Person>()
```

```
new PersonAgeComparator())
```

Some methods in TreeSet and TreeMap

Method	Description
TreeSet.ceiling(e)	Returns the lowest element $\geq e$
TreeMap.ceilingKey(key)	Returns the lowest key $\geq key$
TreeSet.higher(e)	Returns the lowest element $> e$
TreeMap.higherKey(key)	Returns the lowest key $> key$
TreeSet.floor(e)	Returns the highest element $\leq e$
TreeMap.floorKey(key)	Returns the highest key $\leq key$
TreeSet.lower(e)	Returns the highest element $< e$
TreeMap.lowerKey(key)	Returns the highest key $< key$
TreeSet.pollFirst()	Returns and removes the first entry
TreeMap.pollFirstEntry()	Returns and removes the first key-value pair
TreeSet.pollLast()	Returns and removes the last entry
TreeMap.pollLastEntry()	Returns and removes the last key-value pair
TreeSet.descendingSet()	Returns a NavigableSet in reverse order
TreeMap.descendingMap()	Returns a NavigableMap in reverse order

The Collections class

Lecture notes

Some methods in Collections

- `sort(List list): void`
 - Sorts list into natural order
 - The objects in the list must implement Comparable
- `sort(List list, Comparator c): void`
 - Sorts list using the ordering specified in c
- Example

```
ArrayList<Person> thePeople = new ArrayList<>();  
...  
Collections.sort(thePeople);  
...  
Collections.sort(thePeople,  
                 new PersonAgeComparator());
```



Some methods in Collections

- `binarySearch(List list, Object key) : int`
- `binarySearch(List list,
Object key,
Comparator c) : int`
- These methods search for the key in list
- list must be sorted prior to searching
 - the Comparator c specifies the ordering
- If the key is found, its index is returned
otherwise, a negative value is returned
 - (insertion point) - 1
 - Where insertion point is the position the key would have if it were in the list

Some methods in Collections

- `shuffle(List list) : void`
 - Randomly reorder the list elements
- `max(Collection c) : Object`
- `max(Collection c, Comparator cp) : Object`
 - Returns the largest element in the list
 - Three are also `min()` methods with the same parameter lists

Generic classes and Collections

Lecture slides

In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write a Java program that demonstrates some of the concepts in the lecture notes
 - Look at how to create and use generic classes
 - Collections
 - ArrayList, HashMap, TreeMap and TreeSet
 - Comparing objects
 - java.util.Comparator and
java.util.Comparable
- Ask questions about anything you don't understand

Problem

- Write a Person class and create a variety of collections of Person objects to explore their differences
- Also, sort the ArrayList and TreeSet collections using natural order and custom comparison rules

Auto-closeable resources and Custom exceptions

Lecture slides

In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write a Java program that demonstrates the concepts in the lecture notes
 - The `AutoCloseable` interface
 - The `try-with-resources` statement
 - Create our own exceptions
- Ask questions about anything you don't understand

Problem

- Modify the Lecture4Demo2 application so that objects are written and read using try-with-resources statements
- When problems are discovered in creating a new book, throw and catch custom exceptions

The need for auto-closeable resources

Lecture notes

In these lecture notes

We will:

- Look at why we might need auto-closable resources
- Introduce
 - AutoCloseable interface
 - The try-with-resources statement
- Create our own exceptions

What is a resource?

- An object that must be closed after the program has finished using it
 - For example:
 - Scanner
 - input streams, output streams (weeks 3 & 4)
 - readers and writers (weeks 3 & 4)
 - database connections (later in the module)
 - many other classes

Using a resource

- Until now, we have been responsible for closing any resource we have opened

```
try
{
    SomeClass sc = new SomeClass(); // create the resource

    // use the resource in multiple statements

    sc.close(); // close the resource
}

catch (Exception e) { }
```

Using a resource – the traditional way



Coding

- Review the example code and watch out for:
 - how the `try` block opens and closes the resource (`SomeClass`)
 - the order in which the methods are called
- If you have any questions, ask

Project: ResourceExample

The problem with the traditional way

- The programmer must remember to close a resource
 - It is easy to forget
- Wouldn't it be good if the resource was automatically closed?

Auto-closeable resources

Lecture notes

To make a resource auto-closeable

- Implement the `java.lang.AutoCloseable` interface
 - No need for an import statement because the `java.lang` package is available to all classes

```
public class SomeClass implements AutoCloseable
{
    ...
    @Override
    public void close() throws Exception
    {
        // close the resource
    }
}
```

Using auto-closeable resources

Lecture notes

The try-with-resources statement

- A try-with-resources statement is a try statement that...
 - declares one or more resource
 - automatically closes all declared resources

```
try (SomeClass sc = new SomeClass()) // create the resource
{
    // use the resource in multiple statements

    // the resource is automatically closed
}

catch (Exception e) { }
```

The `try-with-resources` statement



Coding

- Review the example code and watch out for:
 - how the `try-with-resources` statement opens and closes the resource (`SomeClass`)
 - the order in which the methods are called (different from the previous example)
- If you have any questions, ask

Project: TryWithResourceExample

The `try-with-multiple resources`



Coding

- Review the example code and watch out for:
 - how the `try-with-resources` statement opens and closes multiple resources (`SomeClass` and `SomeOtherClass`)
 - the order in which the methods are called
- If you have any questions, ask

Project: TryWithMultipleResourcesExample

Custom exceptions

Lecture notes



Checked and unchecked exceptions

- Decide whether to write a checked or unchecked exception
- A checked exception is checked by the compiler to see that
 - a method using it declares that it throws the exception or has a try-catch statement to catch it
 - a calling method has a try-catch statement to catch the exception if the called method is declared to throw it
- An unchecked exception is not checked by the compiler in the way described above

Checked exception example



Coding

- Review the example code and watch out for:
 - how methods declare that they could throw an exception instead of handling it
 - how methods that call “throwing methods” must also declare that they could throw an exception instead of handling it
- If you have any questions, ask

Project: CheckedExceptionExample

Unchecked exception example



Coding

- Review the example code and watch out for:
 - how methods do not declare that they could throw an unchecked exception instead of handling it
- If you have any questions, ask

Project: `UncheckedExceptionExample`

Writing our own exceptions

- Although there are many standard exceptions in Java, we might want to write our own
- For example, we might want an exception that
 - has a name meaningful to our application
 - holds data relevant to the error

Writing a custom exception

- To write a checked exception
 - Write a subclass of `Exception`
 - Write appropriate constructor methods
 - Write any methods that are needed
- To write an unchecked exception
 - Write a subclass of `RuntimeException`
 - Write appropriate constructor methods
 - Write any methods that are needed

Custom exception examples

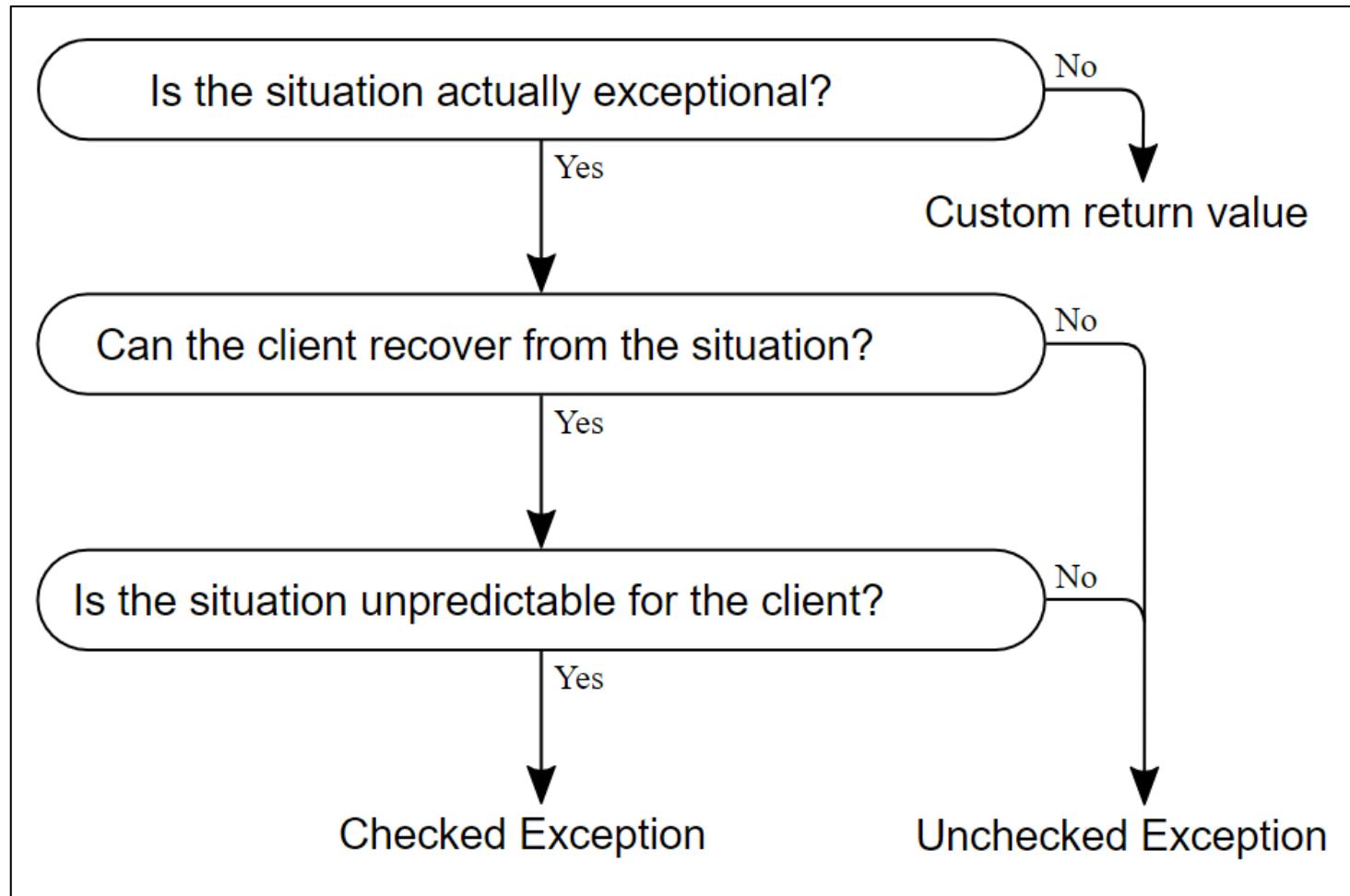


Coding

- Review the example code and watch out for:
 - how custom checked exceptions are written
 - how custom unchecked exceptions are written
- If you have any questions, ask

Project: CustomExceptionExamples

How to decide which to use?



<https://programming.guide/java/choosing-between-checked-and-unchecked-exceptions.html>

How to decide which to use?

- Checked Exceptions
 - used for predictable, but unpreventable errors that are reasonable to recover from
- Unchecked Exceptions
 - used for everything else
<https://stackoverflow.com/questions/27578/when-to-choose-checked-and-unchecked-exceptions>
- Oracle – the definitive statement?
 - “If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception”
<https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>

Lambda expressions and Functional interfaces

Lecture slides

In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write Java programs that demonstrate the concepts in the lecture notes
 - Introduce lambda expressions
 - Introduce some functional interfaces
- Ask questions about anything you don't understand

Problem 1

- Write a Java program that creates an ArrayList of integers, and populates it with values 10, 9, ... 1
- Use lambda expressions to
 - Print the list
 - Sort the list into ascending order
 - Print the list again

Problem 2

- Modify the solution to Problem 1 so that it uses...
 - a Supplier to create and populate the ArrayList of integers
 - a Consumer to print the ArrayList of integers
 - a Consumer to sort the ArrayList of integers
 - a Function and a Predicate to make all odd numbers in the ArrayList negative
 - a Consumer and a Predicate to triple all positive numbers in the ArrayList
 - a Predicate to remove all numbers in the range -1 to 6 (inclusive) from the ArrayList

Introduction to Lambda expressions

Lecture notes



In these lecture notes

We will:

- Introduce lambda expressions
- Introduce some functional interfaces

What is a lambda expression?

- A lambda expression can be thought of as an anonymous method
 - A method without a name
 - (Similar concept to anonymous classes in Week 2)
- Lambda expressions express instances of single-method classes more compactly
 - As shown in the following examples

Lambda expression example 1

- Print the contents of an ArrayList
 - Assume the ArrayList already contains data
 - A for-each loop

```
for (int num : nums) {  
    System.out.print(num + " ");  
}
```

- Can be replaced with

```
nums.forEach(  
    num -> { System.out.print(num + " "); }  
) ;
```

Lambda
expression

Syntax of a Lambda expression

- A lambda expression consists of...

```
nums.forEach(  
    (num) ->  
        { System.out.print(num + " "); }  
) ;
```

1. A comma-separated list of formal parameters enclosed in parentheses

2. The arrow token, ->

3. A body, which consists of a single expression or a statement block

Lambda expression example 2

- In AnonymousInnerClassExample, Week 2
 - An anonymous single-method inner class

```
b1.addActionListener(  
    new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent ae) {  
            getContentPane().setBackground(Color.BLUE);  
        } } );
```

- Can be replaced with a lambda expression

```
b1.addActionListener(  
    (ActionEvent ae) ->  
    { getContentPane().setBackground(Color.BLUE); } );
```

Lambda expression example 3

- A lambda expression can have many parameters
 - An anonymous single-method inner class

```
nums.sort(  
    new Comparator<Integer>() {  
        @Override  
        public int compare(Integer t1, Integer t2) {  
            return Integer.compare(t1, t2);  
        }  
    } );
```

- Can be replaced with a lambda expression

```
nums.sort(  
    (Integer t1, Integer t2) -> Integer.compare(t1, t2));
```

Lambda expression example 4

- A lambda expression can omit parameter types
 - An anonymous single-method inner class

```
nums.sort (  
    new Comparator<Integer> () {  
        @Override  
        public int compare(Integer t1, Integer t2) {  
            return Integer.compare(t1, t2);  
        }  
    } );
```

- Can be replaced with a lambda expression

```
nums.sort((t1, t2) -> Integer.compare(t1, t2));
```

Functional Interfaces in Java

Lecture notes



Functional interfaces

- Functional interfaces provide target types for lambda expressions and method references
- Each functional interface has a single abstract method
 - called the functional method
 - the lambda expression's parameter and return type are matched or adapted to the functional method
- Define “function shapes”

Basic function shapes

- Function
 - unary function from T to R
- Predicate
 - unary function from T to Boolean
- Supplier
 - nilary function to R
- Consumer
 - unary function from T to void



Function interface

- Represents a function that accepts one argument and produces a result

```
@FunctionalInterface  
public interface Function<T, R>
```

- T – the type of the input to the function
- R – the type of the result of the function

Function interface

- Methods

- `apply(T t)`
the functional method that returns the value resulting from applying the function to `t`
- `andThen(Function after)`
returns a composed function of `this` followed by `after`
- `compose(Function before)`
returns a composed function of `before` followed by `this`
- `identity()`
returns a function that always returns its input argument

4 Function interface examples



Coding

- Review the example code and watch out for:
 - how to define a function with an anonymous class
 - how to define a function with a lambda expression
 - how to use the various methods in the Function interface
- If you have any questions, ask

Project: FunctionalInterfaceExamples

Predicate interface

- Represents a function that accepts one argument and produces a Boolean value

```
@FunctionalInterface  
public interface Predicate<T>
```

- T – the type of the input to the function

Predicate interface

- Methods

- `test (T t)`
the functional method that evaluates this predicate on `t`
- `and (Predicate other)`
returns a composed predicate of this **AND** `other`
- `or (Predicate other)`
returns a composed predicate of this **OR** `other`
- `negate ()`
returns a predicate that returns the negation of this
- `isEqual (Object obj)`
returns a predicate that tests for equality between this and `obj`

4 Predicate interface examples



Coding

- Review the example code and watch out for:
 - how to use the various methods in the Predicate interface
 - If you have any questions, ask

Project: FunctionalInterfaceExamples

Supplier interface

- Represents a function that has no arguments and produces a value

```
@FunctionalInterface  
public interface Supplier<R>
```

- R – the type of results supplied by this supplier
- There is no requirement for a new or distinct result to be returned each time the supplier is invoked
- Method
 - get ()
the functional method that supplies a value of type R

2 Supplier interface examples



Coding

- Review the example code and watch out for:
 - how to use a `Supplier` to generate an integer
 - how to use a `Supplier` to generate an array of integers
- If you have any questions, ask

Project: `FunctionalInterfaceExamples`

Consumer interface

- Represents a function that accepts one argument and returns no result

```
@FunctionalInterface  
public interface Consumer<T>
```

- T – the type of the input to the function
- Unlike most other functional interfaces, Consumer is expected to operate via side-effects

Consumer interface

- Methods

- `accept (T t)`
the functional method that performs the operation on `t`
- `andThen (Consumer after)`
returns a composed consumer that performs this operation followed by `after`

2 Consumer interface examples



Coding

- Review the example code and watch out for:
 - how to use a Consumer to print an integer
 - how to use a Consumer to print an array of integers
 - how to use the various methods in the Consumer interface
- If you have any questions, ask

Project: FunctionalInterfaceExamples

Arity of functional interfaces

- Arity means “the number of arguments taken by a function” (<https://en.wikipedia.org/wiki/Arity>)
- Function shapes have a natural arity based on how they are most commonly used
- The basic shapes can be modified by an arity prefix to indicate a different arity
 - e.g. BiFunction, BiConsumer, etc.

BiFunction interface

- Represents a function that accepts two arguments and produces a result

```
@FunctionalInterface  
public interface Function<T, U, R>
```

- T – the type of the first input to the function
- U – the type of the second input to the function
- R – the type of the result of the function

BiFunction interface

- Methods

- `apply(T t, U u)`
the functional method that returns a value of type R resulting from applying the function to t and u
- `andThen(Function after)`
returns a composed function of this followed by after

Java Streams

1

Lecture slides

In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write Java programs that demonstrate the concepts in the lecture notes
 - Introduce Java streams
 - Use some streams pipelines
- Ask questions about anything you don't understand

Problem 1

- Modify the solution to Problem 2 from last week so that it uses an IntStream and a pipeline of operations
 - Use a Supplier that uses a stream builder to create an IntStream with numbers 10, 9, ... 1
 - Use a Function and a Predicate to make all odd numbers in the stream negative
 - Use a Function and a Predicate to triple all positive numbers in the stream
 - Filter out all numbers in the range -1 to 6 (inclusive)
 - Sort the stream
 - Print the stream

Problem 2

- Modify the solution to Problem 1 so that it uses `IntStream.iterate()` instead of a `Supplier` to create an `IntStream` with numbers 10, 9, ... 1

Introduction to Java streams

Lecture notes

In these lecture notes

We will:

- Introduce Java streams
- Look at some non-terminal operations for streams
- Look at some terminal operations for streams

What is a Java stream?

- A stream is a sequence (possibly infinite) of values (elements)
 - It is not the same as an input or output stream, which operates on binary values
- A stream conveys elements from a source
 - It is not a data collection, which stores elements
- A stream is capable of internal iteration of its elements
 - A data collection requires external iteration (i.e. written by the programmer)

Stream operations

- A stream supports sequential and parallel aggregate operations
 - An operation is applied to all the elements in the stream
- An operation on a stream produces a result
 - but does not change the values of the source
- Operations try to be lazy
 - Do the least possible work to accomplish the result

Stream operations

- Operations are either...
 - *Intermediate*
Produce another stream
- or
- *Terminal*
Produce a value or a side-effect
- Example operations
 - Filtering
 - Mapping
 - Removal of duplicates

Several ways to obtain a stream

- From a Collection
 - Call its `stream()` or `parallelStream()` method
- From an array
 - Call `Arrays.stream(Object[])`
- From static methods, e.g.:
 - `Stream.of(Object[])`
 - `IntStream.range(int, int)`
- From files
 - `BufferedReader.lines()`

Several ways to obtain a stream

- To get a stream of random integers
 - `Random.ints()`
- Many other classes have methods to produce streams

IntStream example

- Get an IntStream

```
IntStream stream = IntStream.range(1, 10);
```

- Produces a sequential ordered stream of int values starting with 1, and ending with 9 (i.e. the second parameter is not included)

```
stream.forEach(  
    num -> { System.out.print(num + " "); }  
) ;
```

- The terminal operation `forEach()` applies the action to each element of the stream
- This code is in the StreamExamples project

Stream pipelines

- A stream can be processed only once
 - e.g. Calling `forEach()` twice for the same stream gives an exception
- Operations can be chained together to form a stream pipeline
 - Zero or more intermediate operations followed by one terminal operation

```
IntStream.range(1, 10)
    .map(i -> i * 6)    // multiply each value by 6
    .map(i -> i - 2)    // subtract 2 from each value
    .forEach(i -> System.out.print(num + " "));
```

Stream pipelines

- The operations on a stream are not applied to the stream's elements until the terminal operation method is called

```
IntStream.range(1, 10)
    .map(i -> i * 6)    // multiply each value by 6
    .map(i -> i - 2)    // subtract 2 from each value
    .forEach(i -> System.out.print(num + " "));
```

- When `forEach()` is called, the preceding intermediate operations are performed
- This code is in the `StreamExamples` project

The peek () operation

- The peek () operation...
 - is an intermediate operation,
 - applies an action to each element, and
 - returns a stream with the same values as the stream to which it is applied
- Example
 - Print the stream elements to see the effect of an operation

Order of processing a stream

- The `peek()` operation can be used to see the order in which elements of a stream are processed

```
IntStream.range(1, 10)
    .peek(i -> System.out.print(num + " "))
    .map(i -> i * 6)
    .peek(i -> System.out.print(num + " "))
    .map(i -> i - 2)
    .forEach(i -> System.out.print(num + " "));
```

- This code is in the StreamExamples project

Some Java stream operations

Lecture notes

Parameter of stream operations

- Most stream operations accept parameters that describe user-specified behaviour
 - They are always instances of a functional interface such as `Function`, and are often lambda expressions or method references
- To preserve correct behaviour, these behavioural parameters:
 - must be non-interfering
 - they do not modify the stream source
 - in most cases must be stateless
 - their result should not depend on state that might change while executing the stream's pipeline

map()

- Stream<R> map (Function<T, R> f)
 - An intermediate operation that applies function, f, to the elements of this stream and returns a stream of the results
- A previous example
 - Applies two mapping operations

```
IntStream.range(1, 10)
    .map(i -> i * 6) // multiply each value by 6
    .map(i -> i - 2) // subtract 2 from each value
    .forEach(i -> System.out.print(num + " "));
```

mapToInt()

- A mapping operation can change the type of the stream

```
Arrays.asList("One", "Two", "Three", "Four", "Five")
```

```
.stream()
```

Returns a Stream<String>

```
.mapToInt(String::length)
```

Returns an IntStream

```
.forEach(e -> System.out.print(e + " "));
```

- The stream starts with a sequence of strings, which is mapped to a sequence of integers, which is printed

IntStream.sorted()

- IntStream sorted()
 - An intermediate operation that returns a stream containing the integers of this stream in sorted order

```
Random r = new Random();  
  
int[] ints  
    = r.ints(10, 50, 60)  
        .peek(i -> System.out.print(i + " "))  
        .toArray();  
  
System.out.println();  
  
Arrays.stream(ints)  
    .sorted()  
    .forEach(i -> System.out.print(i + " "));
```

- This code is in the StreamExamples project

Stream.sorted()

- Stream<T>.sorted()
 - An intermediate operation that returns a stream containing the elements of this stream sorted in natural order
- Stream<T>.sorted(Comparator<T> c)
 - An intermediate operation that returns a stream containing the elements of this stream sorted according to Comparator c

count() and distinct()

- long count()
 - A terminal operation that returns the number of elements in this stream
- Stream<T> distinct()
 - An intermediate operation that returns a stream consisting of the distinct elements in this stream
 - The new stream has no duplicate objects as defined by Object.equals(Object))

Stream.filter()

- Stream<T> filter(Predicate<T> p)
 - An intermediate operation that returns a stream containing the elements of this stream that match Predicate p

Arrays

```
.asList("One", "Two", "Three", "Four", "Five")
.stream()
.filter(e ->
    e.startsWith("F") || e.startsWith("T"))
.forEach(e -> System.out.print(e + " "));
```

- This code is in the StreamExamples project

Efficiency of stream pipelines

- The order of intermediate operations in a stream pipeline can make the pipeline more or less efficient

```
Arrays.stream(ints)  
    .sorted()  
    .distinct();
```

Sort, including duplicates
then
Remove duplicates

is less efficient than

```
Arrays.stream(ints)  
    .distinct()  
    .sorted();
```

Remove duplicates first
then
Sort without duplicates

- This code is in the StreamExamples project

Building Java streams

Lecture notes



Building streams

- A Stream.Builder object can be used to make a stream
- The builder has three methods
 - void accept (T t)
Adds an element to the stream being built
 - Stream.Builder<T> add (T t)
Adds an element to the stream being built and returns a reference to the builder (i.e. itself)
 - Stream<T> build ()
builds the stream from the elements previously added

Building streams

- To build a stream of strings

```
Stream<String> animals
    = Stream.<String>builder()
        .add("Dog")
        .add("Cat")
        .add("Bird")
        .add("Fish")
        .add("Elephant")
        .build();
```

- The built stream can be processed as described previously
- This code is in the StreamExamples project

Java Streams

2

Lecture slides

In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write Java programs that demonstrate the concepts in the lecture notes
 - Perform reductions of Java streams
 - `reduce()` and `collect()`
 - Search for data in a stream
 - Use the `flatMap()` method
- Ask questions about anything you don't understand

Problem

- Modify the lecture demo from Week 6 to use streams, reductions, data searching and flat mapping as appropriate
 - Existing methods
 - In `displayBook()` use a stream collection when printing
 - In `viewListOfBooks()` replace the loop with a functional operation
 - In `createBook()`, `findBook()` and `Book.findCopy()` replace the loop with a stream
 - Add a new menu option
 - View the total number of book copies in the library, including how many are on loan

Java stream reduction

Lecture notes



In these lecture notes

We will:

- See how to perform reductions of Java streams
- See how to search for data in a stream
- Look at the `flatMap ()` method

What is a Java stream reduction?

- A terminal operation that returns:
 - A single value by combining the values in the stream (called “reduction”)
 - A collection containing values in the stream (called “collection”)
- For custom reductions use
 - Stream.`reduce()` and
 - Stream.`collect()`

IntStream reduction examples



Coding

- Review the example code to see how the following IntStream reductions are used:
 - average ()
 - count ()
 - max ()
 - min ()
 - sum ()
- If you have any questions, ask

Project: ReductionExamples

Custom IntStream reduction

- Use `IntStream.reduce()`

- Method signature

```
int reduce(int identity, IntBinaryOperator op)
```

- `identity` is the initial value of the reduction
(also the result of the reduction if the stream is empty)
 - `op` is an accumulator function that takes two integer parameters and returns an integer result

Custom IntStream reduction

- To sum an IntStream using `reduce()`
 - The accumulation function adds two integers

```
int sum = integers.reduce(0, (a, b) -> a+b);
```

- Can also be written as

```
int sum = integers.reduce(0, Integer::sum);
```

Accumulator
function

- This is equivalent to the following pseudo-code

```
int result = identity
foreach element in this IntStream loop
    result = accumulator.applyAsInt(result, element)
end loop
return result
```

Stream reduction

- Any kind of stream can be reduced
- For example
 - Reduce a stream of Person objects to a string containing all the names not beginning with ‘A’

```
String names  
      = personStream  
          .map(p -> p.getName())  
          .filter(name -> !name.startsWith("A"))  
          .reduce("", (str, name) -> str += name + " ")  
          .trim()  
          .replaceAll(" ", ", ");
```

- This code is in the ReductionExamples project

Stream reduction

- This example attempts to do the same thing as the previous example without maps to a stream of strings

```
String names  
      = personStream  
          .filter(p -> !p.getName().startsWith("A"))  
          .reduce("", (str, p) -> str += p.getName() + " ")  
          .trim()  
          .replaceAll(" ", ", ");
```

This line does
not compile

- This code is in the ReductionExamples project

Stream reduction problem

- The problem is to do with data types
 - According to the method signature

```
T reduce(T identity,  
        BinaryOperator<T, T> accumulator)
```

- The accumulator should receive and return objects of the same type, but ...

```
.reduce("", (str, p) -> str += p.getName() + " ")
```

- The accumulator actually has Person and String objects i.e. *the data types do not match*
- This can be fixed by using the version of reduce () that has a combiner...

Stream reduction

- The combiner
 - is a function for combining two values
 - must be compatible with the accumulator
 - “Compatible” is made clear by the signature of `reduce()`

```
U reduce(U identity,  
         BiFunction<U, T, U> accumulator,  
         BinaryOperator<U> combiner)
```

- The combiner operates on data of the same type as that returned by the accumulator
i.e. data type U

Stream reduction

```
String names
    = personStream
        .filter(p -> !p.getName().startsWith("A"))
        .reduce("", _____)
        (str, p) -> str += p.getName() + " ", _____
        String::concat)
        .trim()
        .replaceAll(" ", ", ");
```

The identity

The accumulator

The combiner

- This example solves the problem because
 - The identity is a string
 - The accumulator's first parameter and return type are strings
 - The combiner operates on strings
 - This code is in the ReductionExamples project

Java stream collection

Lecture notes



IntStream collection

- Use IntStream.collect()
 - Method signature

```
R collect(Supplier<R> supplier,  
          ObjIntConsumer<R> accumulator,  
          BiConsumer<R, R> combiner)
```

- supplier, accumulator and combiner all work with the same data type
- The data type must provide mutable objects

IntStream collection example

- Collect the stream into an ArrayList

```
ArrayList<Integer> list  
    = intStream.collect(  
        ArrayList::new,  
        ArrayList::add,  
        ArrayList::addAll);
```

Supplies a new ArrayList

Accumulates values using add()

Combines partial collections into
one ArrayList

- This code is in the ReductionExamples project

Stream collection

- Use Stream.collect() in the same way as for IntStream
- The Collectors class can also be used to provide ready-made collectors

```
List<Integer> list  
    = intStream.collect(Collectors.toList());
```

- This code is in the ReductionExamples project

Search for data in Java streams

Lecture notes

Searching IntStreams

- There are several methods for searching an IntStream

```
OptionalInt findFirst()
```

Returns an OptionalInt describing the first element of this stream, or an empty OptionalInt if the stream is empty

```
OptionalInt findAny()
```

Returns an OptionalInt describing any element of this stream, or an empty OptionalInt if the stream is empty

```
boolean allMatch(IntPredicate predicate)
```

Returns whether all elements of this stream match the provided predicate

```
boolean anyMatch(IntPredicate predicate)
```

Returns whether any element of this stream matches the provided predicate

```
boolean noneMatch(IntPredicate predicate)
```

Returns whether no elements of this stream match the provided predicate

What is OptionalInt?

- A container object which may or may not contain an int value
- If a value is present
 - `isPresent ()` will return `true`
 - `getAsInt ()` will return the value
- The class also has methods that depend on the presence or absence of a contained value
 - Familiarise yourself with the [documentation](#)

Searching Streams

- There are corresponding methods for searching a Stream

```
Optional<T> findFirst()
```

Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty

```
Optional<T> findAny()
```

Returns an Optional describing any element of this stream, or an empty Optional if the stream is empty

```
boolean allMatch(Predicate<T> predicate)
```

Returns whether all elements of this stream match the provided predicate

```
boolean anyMatch(Predicate<T> predicate)
```

Returns whether any element of this stream matches the provided predicate

```
boolean noneMatch(Predicate<T> predicate)
```

Returns whether no elements of this stream match the provided predicate

What is Optional?

- A container object which may or may not contain a non-null value
- If a value is present
 - `isPresent ()` will return `true`
 - `get ()` will return the value
- The class also has methods that depend on the presence or absence of a contained value
 - Familiarise yourself with the [documentation](#)

Examples of searching streams



Coding

- Review the example code to see how to use some of the stream searching methods
- If you have any questions, ask

Project: SearchingExamples

Java stream flatMap () method

Lecture notes

What does flatMap() do?

- The flatMap() operation
 - applies a one-to-many transformation to the elements of the stream
 - and then flattens the resulting elements into a new stream
- Examples
 - Flatten a stream of sentences into a stream of words
 - Convert a 2d array to a 1d array

flatMap () a Stream of sentences

```
sentenceStream.flatMap(  
    sentence -> Stream.of(sentence.split(" ")))
```

"The cat sat on the mat"
"The dog ate the sausage"



"The"
"cat"
"sat"
"on"
"the"
"mat"
"The"
"dog"
"ate"
"the"
"sausage"

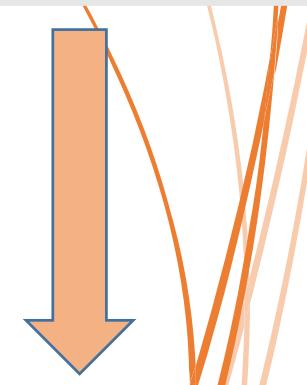
- The stream of sentences has two elements
- This flatMap () operation converts each sentence into a stream of individual words and flattens the result into a single stream with 11 elements

flatMap() a 2-d array

```
Arrays.stream(_2dArray)
    .flatMapToInt(a -> IntStream.of(a))
    .toArray()
```

- The 2-d array stream has two elements, each an array
- This flatMap() operation converts each array into a stream of integers and flattens the result into a single stream with 9 elements

```
{  
    { 2, 4, 6, 8 },  
    { 1, 3, 5, 7, 9 }  
}
```



```
{ 2, 4, 6, 8, 1, 3, 5, 7, 9 }
```

flatMap() examples



Coding

- Review the example code and watch out for:
 - uses of flatMap()
 - using Collectors.joining() to produce a single string containing stream data
- If you have any questions, ask

Project: FlatMapExamples

Java applications with JDBC

Lecture slides

In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write Java programs that demonstrate the concepts in the lecture notes
 - Use a relational database
 - Use the JDBC from a Java application
- Ask questions about anything you don't understand

Problem

- Modify the lecture demo from Week 9 to use a database instead of a file to persist data

Relational databases and SQL

Lecture notes



In these lecture notes

We will:

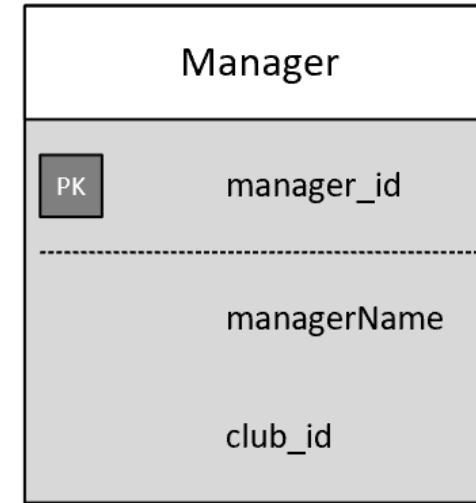
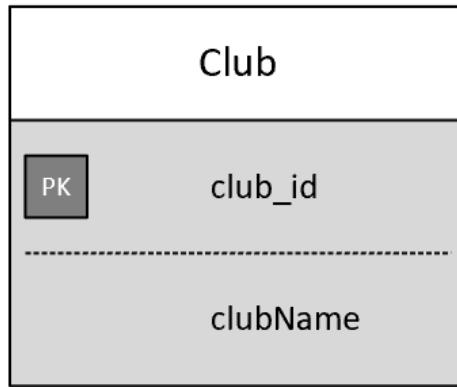
- Review relational databases and SQL
- Introduce the core interfaces of the JDBC API
- See how to use a relational database from a Java program

Relationships in data

- Data in an application often represents relationships
 - A club has a manager
 - A player plays for a club
 - A club has many players
- These relationships can be represented in the data written to a file
 - Using the id values
- It can be difficult to retrieve efficiently two or more items of related data from a file

Relational databases

- Relational databases hold data in tables



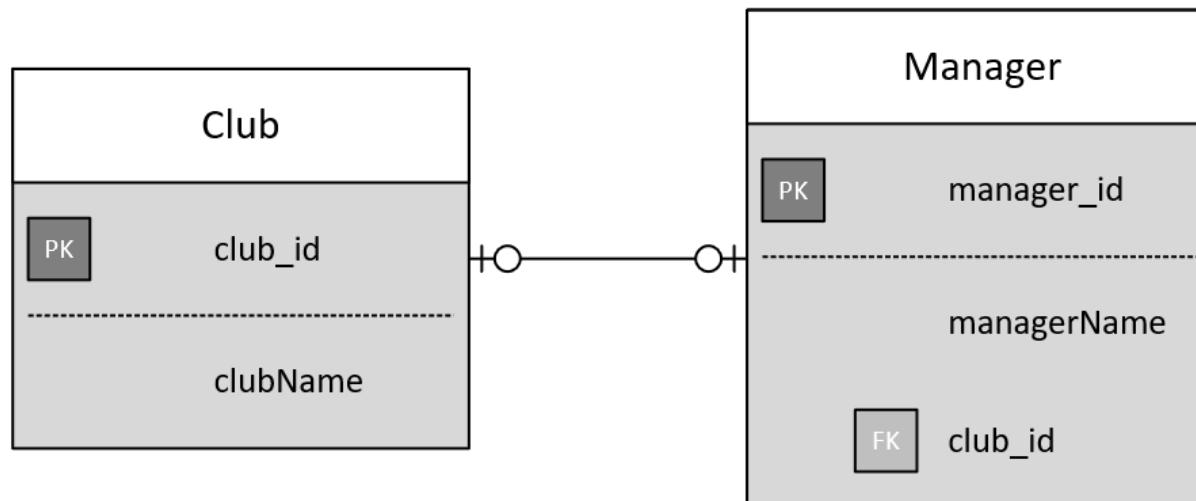
- Each row represents one entity in the table

club_id	clubName
1	A-Team
2	R-Team
3	B-Team

manager_id	managerName	club_id
1	Jan	2
2	Graham	
3	Phil	1

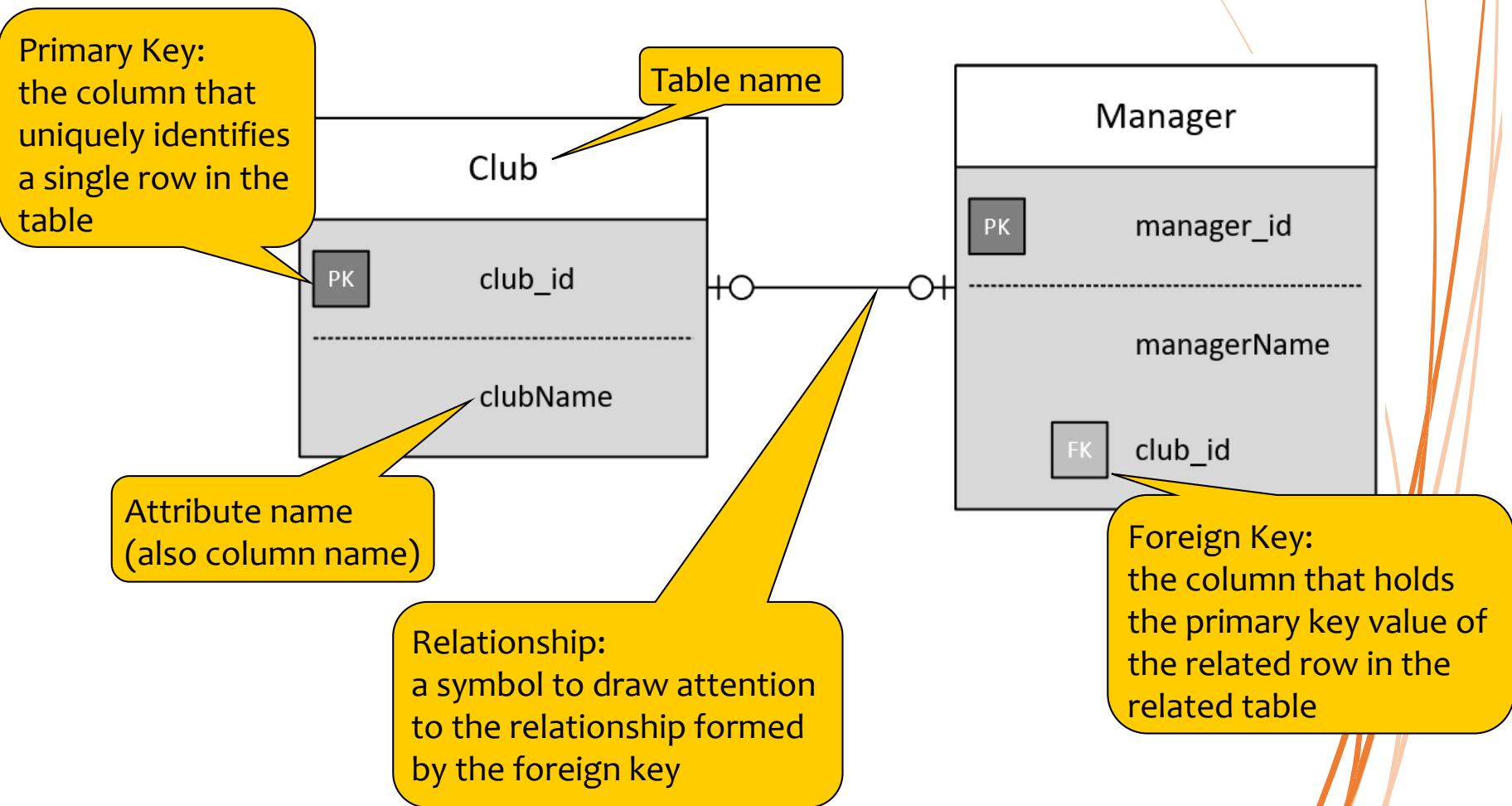
Relational databases

- Relational database tables are linked by relationships
- Tables and their relationships can be represented in a Entity-Relationship Diagram (ERD)
 - A club has a manager



ERD notation

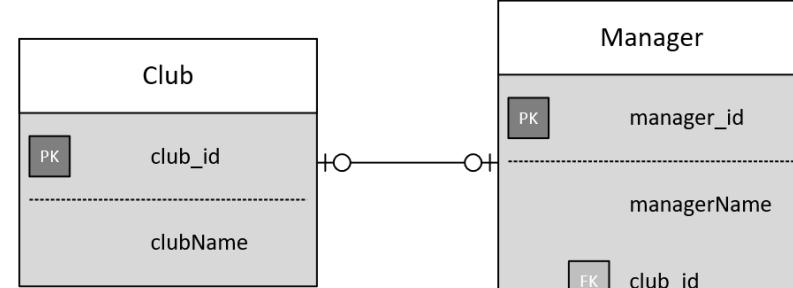
- Crow's Foot notation



Example data

Club table

club_id	clubName
1	A-Team
2	R-Team
3	B-Team



Manager table

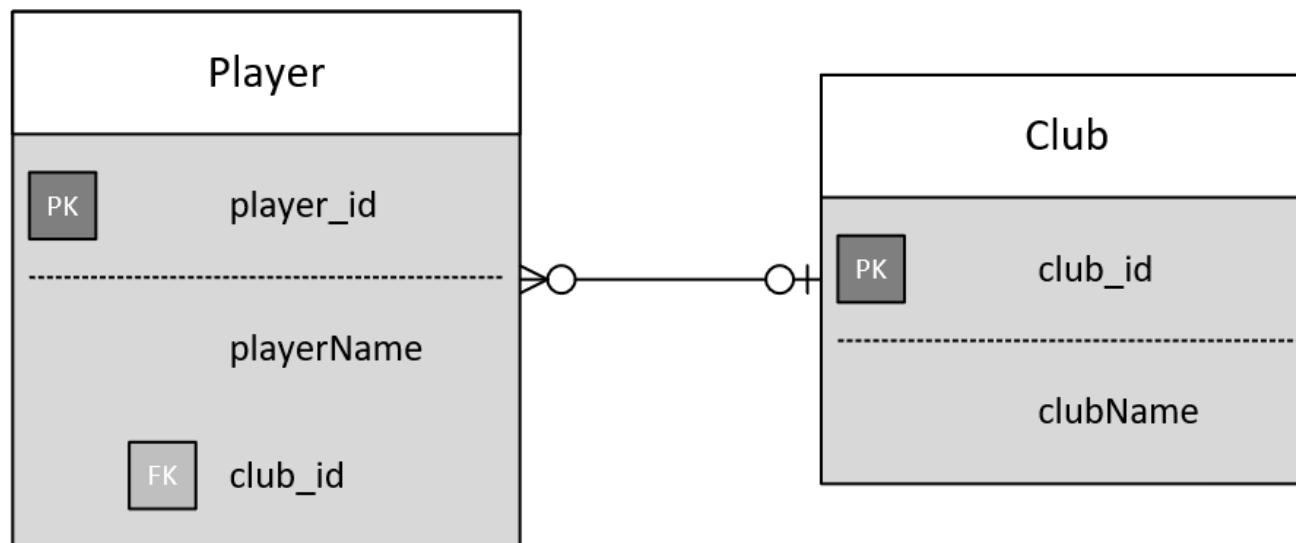
manager_id	managerName	club_id
1	Jan	2
2	Graham	
3	Phil	1

Each row represents
one entity in the table

The primary key value
of the related row in
the related table

ERD notation

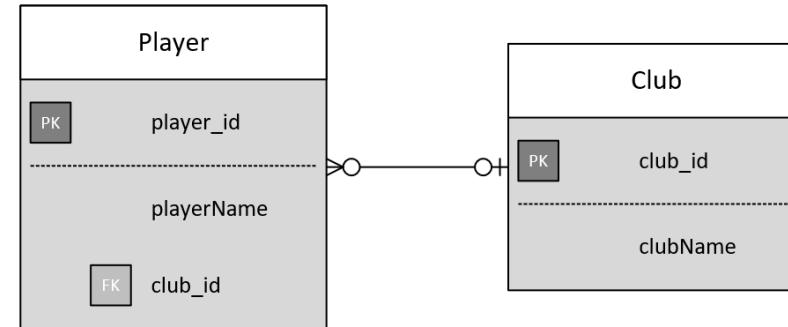
- Crow's Foot notation
 - 1:m relationship
 - A club has many players
 - i.e. many Player rows may have the same club_id



Example of 1:m data

Club table

club_id	clubName
1	A-Team
2	R-Team
3	B-Team

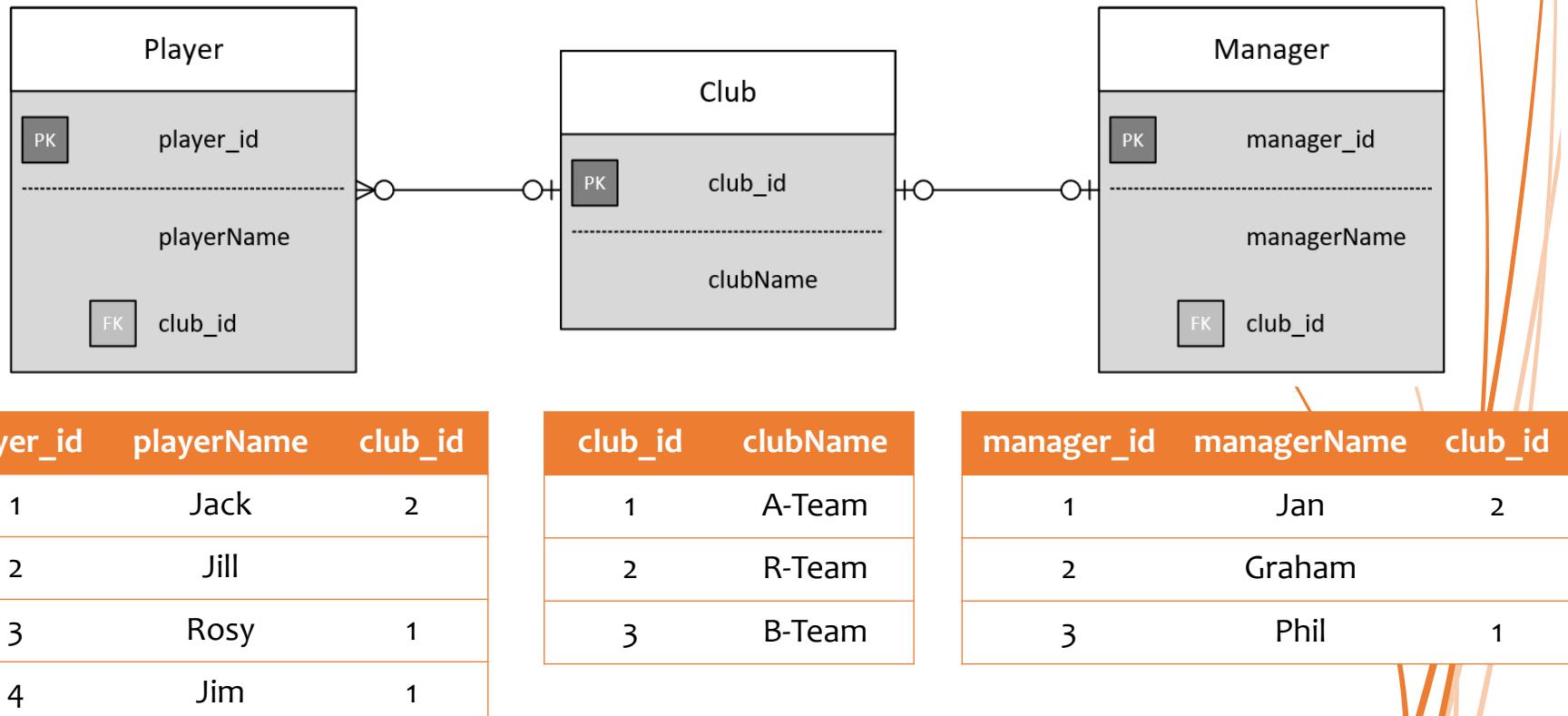


Player table

player_id	playerName	club_id
1	Jack	2
2	Jill	
3	Rosy	1
4	Jim	1

Making relationships work

- Who is Rosy's manager?



SQL

- Structured Query Language
 - A language for manipulating relational databases
- Can be used to...
 - Create and delete tables
 - Insert new rows into a table
 - Update existing rows within a table
 - Delete rows from a table
 - Retrieve data from tables
 - aka querying tables

SQL: create table

- Format

- ```
CREATE TABLE table_name
(
 column_name data_type constraints
 ...
)
```

- Example

- ```
CREATE TABLE Club
(
    club_id number primary key,
    clubName varchar(20) not null unique
)
```

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.10.1.2/ref/index.html>



SQL: delete table

- Format

- `DROP TABLE table_name;`

- Example

- `DROP TABLE Club;`

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.10.1.2/ref/index.html>



SQL: insert row

- Format

- ```
INSERT INTO table_name (col1, col2)
VALUES (value1, value2, ...);
```

- Examples

- ```
INSERT INTO Club
VALUES (1, 'A-Team');
```
 - ```
INSERT INTO Player (player_id, playerName)
VALUES (1, 'Jack');
```

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.1.2/ref/index.html>



# SQL: update row

- Format

- UPDATE table\_name  
SET column\_name = value  
WHERE condition;

- Examples

- UPDATE Club  
SET clubName = 'Top Team'  
WHERE club\_id = 1;
  - UPDATE Player  
SET playerName = 'James'  
WHERE playerName = 'Jim';

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.10.1.2/ref/index.html>



# SQL: delete row

- Format

- ```
DELETE FROM table_name  
[WHERE condition];
```

- Examples

- ```
DELETE FROM Club
WHERE clubName = 'Top Team';
```
- ```
DELETE FROM Player  
WHERE club_id = 1;
```
- ```
DELETE FROM Manager;
```

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.10.1.2/ref/index.html>



# SQL: query a table

- Format

- ```
SELECT column_names FROM table_name
[WHERE condition]
[ORDER BY column];
```

- Examples

- ```
SELECT * FROM Club;
```
  - ```
SELECT player_id, playerName
FROM Player
WHERE club_id = 1
ORDER BY playerName;
```

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.10.1.2/ref/index.html>



SQL: query two tables

- Format

- ```
SELECT column_names
 FROM table_name1 JOIN table_name2
 ON condition
[WHERE condition]
[ORDER BY column];
```

- Example

- ```
SELECT Club.club_id, clubName, playerName
      FROM Club JOIN Player
                ON Club.club_id = Player.club_id;
```

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.1.2/ref/index.html>



SQL: query three tables

- Format

- ```
SELECT column_names
 FROM table_name1 JOIN table_name2 ON condition
 JOIN table_name3 ON condition
[WHERE condition]
[ORDER BY column];
```

- Example

- ```
SELECT Club.club_id, clubName,
          playerName, managerName
      FROM Club
      JOIN Player ON Club.club_id = Player.club_id
      JOIN Manager ON Club.club_id = Manager.club_id;
```

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.1.2/ref/index.html>



SQL: auto-generate primary key

- Example

- CREATE TABLE Club
 - (
 - club_id number primary key
**generated always as identity
(start with 1, increment by 1),**
 - clubName varchar(20) not null unique
 -) ;

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.1.2/ref/index.html>



SQL: foreign key

- Example

```
• CREATE TABLE Player
  (
    player_id integer primary key
      generated always as identity
      (start with 1, increment by 1),
    playerName varchar(20) not null,
    club_id integer,
    FOREIGN KEY (club_id)
      REFERENCES Club(club_id)
  );
```

Full Java DB syntax at: <http://docs.oracle.com/javadb/10.1.2/ref/index.html>

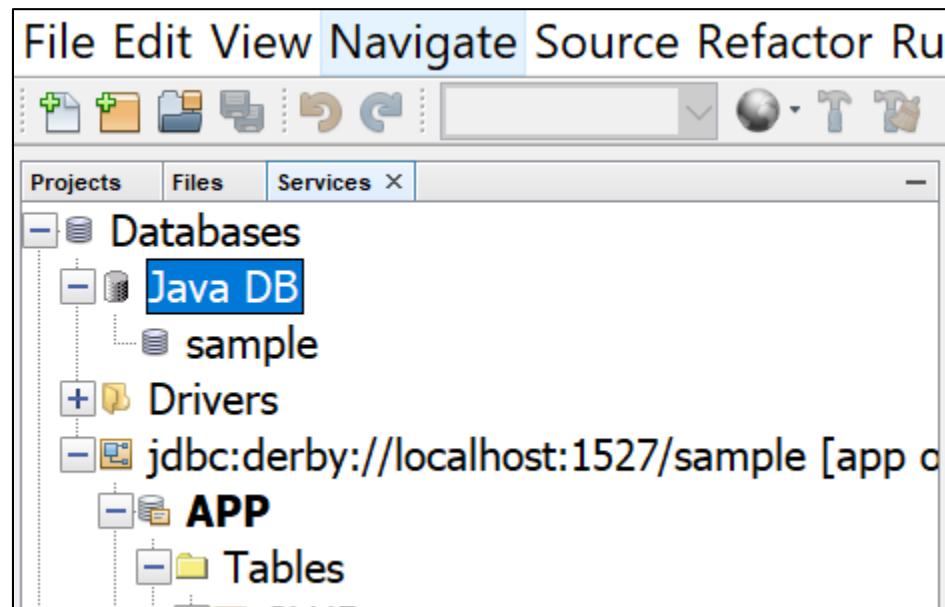


Java DB

Lecture notes

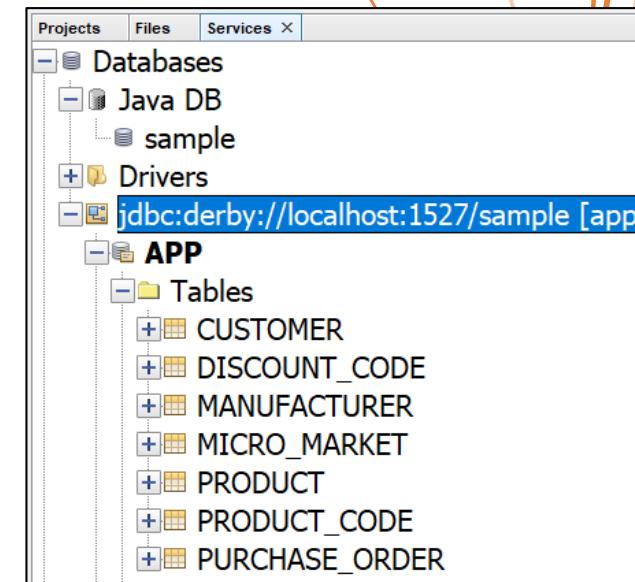
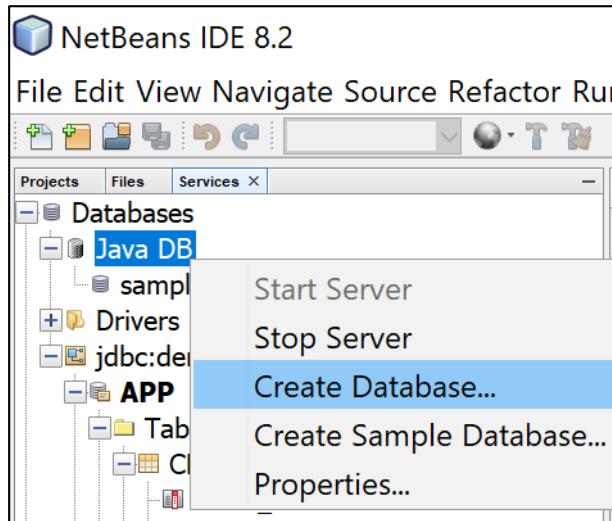
Java DB

- A relational database that ships with the JDK
- Can be accessed via the Services panel in NetBeans



Java DB

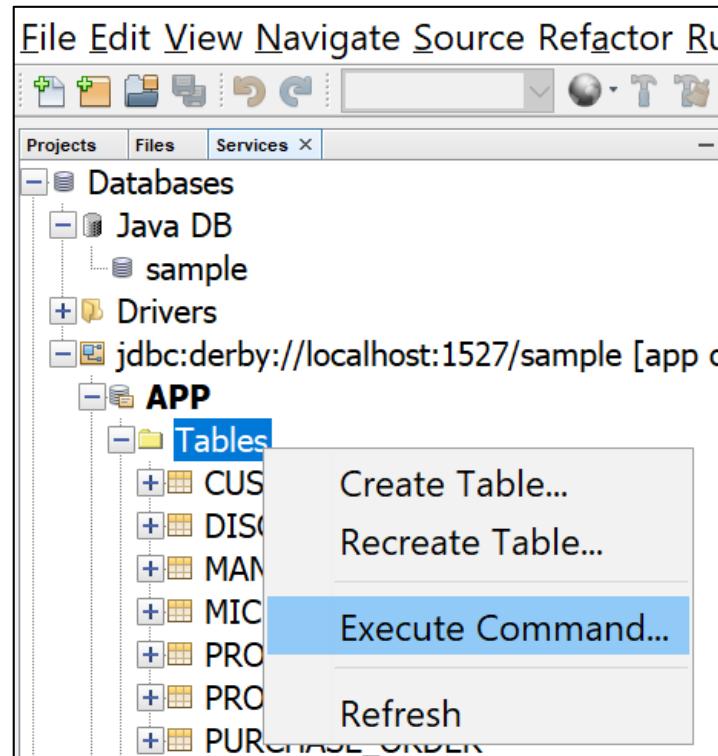
- Right-click Java DB to create a local database



- Java DB has a sample database with example tables

Java DB

- Right-click **Tables** to execute SQL commands



SQL example

```
drop table Player;  
drop table Club;  
  
create table Club (  
    club_id integer primary key  
        generated always as identity  
        (start with 1, increment by 1),  
    clubName varchar(20) not null unique  
);  
create table Player (  
    player_id integer primary key  
        generated always as identity  
        (start with 1, increment by 1),  
    playerName varchar(20) not null,  
    club_id integer,  
    foreign key(club_id) references Club(club_id)  
);
```

SQL example

```
INSERT INTO Club (clubName) VALUES ('A-Team');  
INSERT INTO Club (clubName) VALUES ('R-Team');  
INSERT INTO Club (clubName) VALUES ('B-Team');  
  
INSERT INTO Player (playerName, club_id) VALUES ('Jack', 2);  
INSERT INTO Player (playerName) VALUES ('Jill');  
INSERT INTO Player (playerName, club_id) VALUES ('Rosy', 1);  
INSERT INTO Player (playerName, club_id) VALUES ('Jim', 1);
```

Accessing databases in Java

Lecture notes

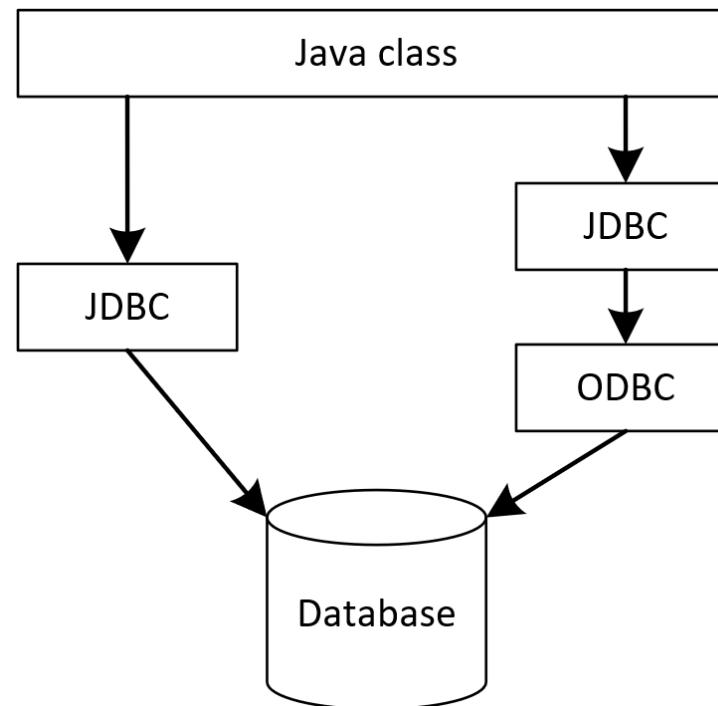


Accessing databases in Java

- Accessing relational databases from Java is easy
 - Setting up a connection varies according to the database
 - After connecting, the same Java code is used to work with the data, irrespective of the database manufacturer
- If it is decided to change the database that a Java program uses...
 - Only two lines of Java code have to change
 - The rest of the code remains the same!

Component architecture

- Two possibilities
 - Manufacturer JDBC classes
 - JDBC-ODBC bridge



The software we need

- JDBC classes for our database
 - Java classes that interact with the database management system (DBMS)
 - Sit between our Java program and the DBMS
 - Available from the database manufacturer
- If not available, use the JDBC:ODBC bridge
 - Ships with Java

The software we need

- JDBC classes for our database
 - **Java DB**
<glassfish installation folder>\javadb\lib\derbyclient.jar
 - **Oracle**
Download jdbc classes from:
<https://www.oracle.com/uk/database/technologies/appdev/jdbc-downloads.html>
 - **MySQL**
Download jdbc classes from:
<https://dev.mysql.com/downloads/connector/j/>
 - **Other database**
Google search and download

Prepare to connect to the database

- Import the SQL package

```
import java.sql.*;
```

- Determine the URL of the database
 - JDBC-ODBC bridge

```
String url = "jdbc:odbc:<driver_name>";
```

- JDBC classes

```
String url = "jdbc:derby://localhost:1527/sample";
```

```
String url =
"jdbc:oracle:thin:@crusstuoral.staffs.ac.uk:1521:stora";
```

```
String url =
"jdbc:mysql://localhost/<database_name>?user=root";
```

Prepare to connect to the database

- Load the database driver manager
 - JDBC-ODBC bridge

```
try
{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch (Exception e)
{
    e.printStackTrace(System.out);
}
```

Prepare to connect to the database

- Load the database driver manager
 - JDBC classes
 - Consult the manufacturer's documentation to discover the correct class name
 - e.g. For a Java DB database ...

```
try
{
    DriverManager.registerDriver(
        new org.apache.derby.jdbc.ClientDriver());
}
catch (Exception e)
{
    e.printStackTrace(System.out);
}
```

Prepare to connect to the database

- Load the database driver manager
 - JDBC classes
 - Consult the manufacturer's documentation to discover the correct class name
 - e.g. For the students' Oracle database ...

```
try
{
    DriverManager.registerDriver(
        new oracle.jdbc.driver.OracleDriver());
}
catch (Exception e)
{
    e.printStackTrace(System.out);
}
```

Prepare to connect to the database

- Load the database driver manager

- JDBC classes

- Consult the manufacturer's documentation to discover the correct class name
 - e.g. For a MySQL database ...

```
try
{
    Class.forName("com.mysql.cj.jdbc.Driver")
        .newInstance();
}
catch (Exception e)
{
    e.printStackTrace(System.out);
}
```

No more differences!

- All other database manipulation code in Java is the same
 - Whichever relational database you use

Connect to the database

- Declare and create a Connection object

```
Connection con = null;  
try  
{  
    con = DriverManager.getConnection(url,  
                                    username,  
                                    password);  
}  
catch (Exception e)  
{  
    e.printStackTrace(System.out);  
}
```

- The database is ready for use

Query the database

```
PreparedStatement stmt;  
ResultSet rs;  
String sqlStr = "SELECT * FROM Club";  
  
try  
{  
    stmt = con.prepareStatement(sqlStr);  
    rs = stmt.executeQuery();  
}  
catch (Exception e)  
{  
    e.printStackTrace(System.out);  
}
```

The SQL statement
to execute
NB: no semi-colon

Process the result set

```
try
{
    while (rs.next())
    {
        System.out.printf("%4d\t%s\n",
                           rs.getInt("club_id"),
                           rs.getString("clubName")) ;
    }
    rs.close();
    stmt.close();
}
catch (Exception e)
{
    e.printStackTrace(System.out);
}
```

Check to see if a row is available.
If it is, load it and return true,
Otherwise, return false

The column name
in the database

Use the method that returns the
column data in the correct type

Insert data

```
PreparedStatement stmt;  
String sqlStr = "INSERT INTO Club(clubName) "  
    + "VALUES ('Staffs United');  
int rowCount;  
  
try  
{  
    stmt = con.prepareStatement(sqlStr);  
    rowCount = stmt.executeUpdate();  
    System.out.println("Rows inserted: " + rowCount);  
    stmt.close();  
}  
catch (Exception e)  
{  
    e.printStackTrace(System.out);  
}
```

The SQL statement to execute
NB: no semi-colon

Used for every statement except queries

Update data

```
PreparedStatement stmt;
String sqlStr = "UPDATE Club "
                + "SET clubName = 'Staffs Rovers' "
                + "WHERE clubName = 'Staffs United';";
int rowCount;

try
{
    stmt = con.prepareStatement(sqlStr);
    rowCount = stmt.executeUpdate();
    System.out.println("Rows updated: " + rowCount);
    stmt.close();
}
catch (Exception e)
{
    e.printStackTrace(System.out);
}
```

Delete data

```
PreparedStatement stmt;
String sqlStr = "DELETE FROM Club "
               + "WHERE clubName = 'Staffs Rovers'";
int rowCount;

try
{
    stmt = con.prepareStatement(sqlStr);
    rowCount = stmt.executeUpdate();
    System.out.println("Rows deleted: " + rowCount);
    stmt.close();
}

catch (Exception e)
{
    e.printStackTrace(System.out);
}
```

Parameters in prepared statements

- Instead of hard-coding values to query, insert, update or delete, use parameters in the prepared statements
- A question mark is a placeholder for a parameter

```
String sqlStr = "INSERT INTO Club(clubName) "  
    + "VALUES (?)";
```

- Set the value of the placeholder

Parameter index;
starts at 1 (not zero)

Use the method
that sets the
column data in the
correct type

```
stmt = con.prepareStatement(sqlStr);
```

```
stmt.setString(1, teamName);
```

```
rowCount = stmt.executeUpdate();
```

Parameter value;
can be supplied by
the user

Activity



Coding

- Review the example code and watch out for:
 - how the database connection is set up
 - how insert, update, delete and query operations are performed on the database
 - how multiple placeholders are used in prepared statements
- If you have any questions, ask

Project: DatabaseExample

Introduction to design patterns

Lecture notes

In these lecture notes

We will:

- Introduce the concept of design patterns
- Introduce some design patterns
 - Singleton
 - Factory
 - Builder
 - Null Object
 - Table Data Gateway
 - Command

The need for design patterns

- Designing reusable software is not easy
- Some design problems and solutions occur repeatedly
- Understanding solutions to recurring problems is beneficial
 - Uses tried-and-tested solutions to problems
 - Provides abstractions to help simplify complexity
 - Gives well-understood structure to the application
 - The solutions can be re-used in different contexts

Design patterns: Essentials

- In general, a design pattern has four essential elements:
 - The pattern name
 - A description of the problem the pattern addresses
 - A general description of the solution to the problem
 - Possible alternative solutions
 - The consequences or results of applying the pattern

Design patterns: Classification

- Design patterns are classified by two criteria
 - Purpose (also called Intent)
 - Scope
- Purpose
 - Reflects what a pattern does
 - Can be:
 - **Creational:** how objects are created
 - **Structural:** the composition of classes or objects
 - **Behavioural:** how classes or objects interact and distribute responsibility

Design patterns: Classification

- Scope
 - Specifies whether the pattern applies primarily to classes or to objects
 - Can be:
 - **Class:** relationships between classes and their subclasses
 - Established through inheritance
 - Static because they are fixed at compile-time
 - **Object:** relationships between objects
 - Can be changed at run-time and are more dynamic
 - Most patterns are in the Object scope
 - Almost all patterns use inheritance to some extent
 - The only patterns labelled "class patterns" are those that focus on class relationships.

Design patterns: Resources

- Gamma , E. et al (1995) *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley
 - This is the seminal work.
 - It describes the concepts of patterns, and catalogues 23 design patterns with their full documentation
- There are many other resources, printed and on-line

Singleton design pattern

Lecture notes

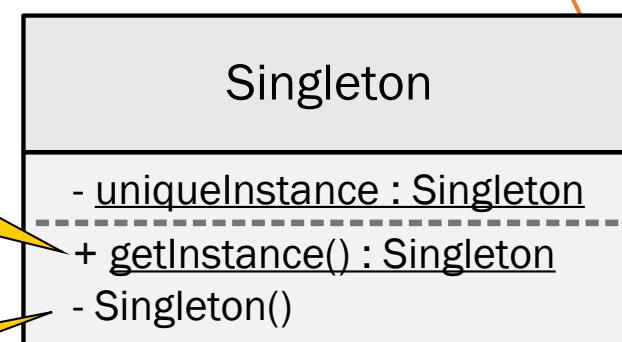
Singleton (Object Creational)

- Intent
 - Ensure a class has only one instance

- Motivation
 - Ensure there is only one instance of a class
 - Class is responsible for enforcing singularity

```
if (uniqueInstance == null)
    uniqueInstance
        = new Singleton();
return uniqueInstance;
```

Private constructor prevents uncontrolled instantiation



Singleton (Consequences)

- Benefits
 - Controlled access to sole instance
 - Because the Singleton class encapsulates its sole instance, it has strict control over how and when clients access it
 - Reduced name space
 - The Singleton pattern is an improvement over global variables
 - It avoids polluting the name space with global variables that store sole instances

Singleton (Consequences)

- Benefits
 - Permits refinement of operations and representation
 - The Singleton class may be sub-classed
 - Permits a variable number of instances
 - If requirements change, the pattern can easily be modified to allow more than one instance of the Singleton class
 - Use the same approach to control the number of instances that the application uses
 - Only the method that gives access to the Singleton instance needs to change

Singleton (Example)

- An example of factory method is in the design patterns example

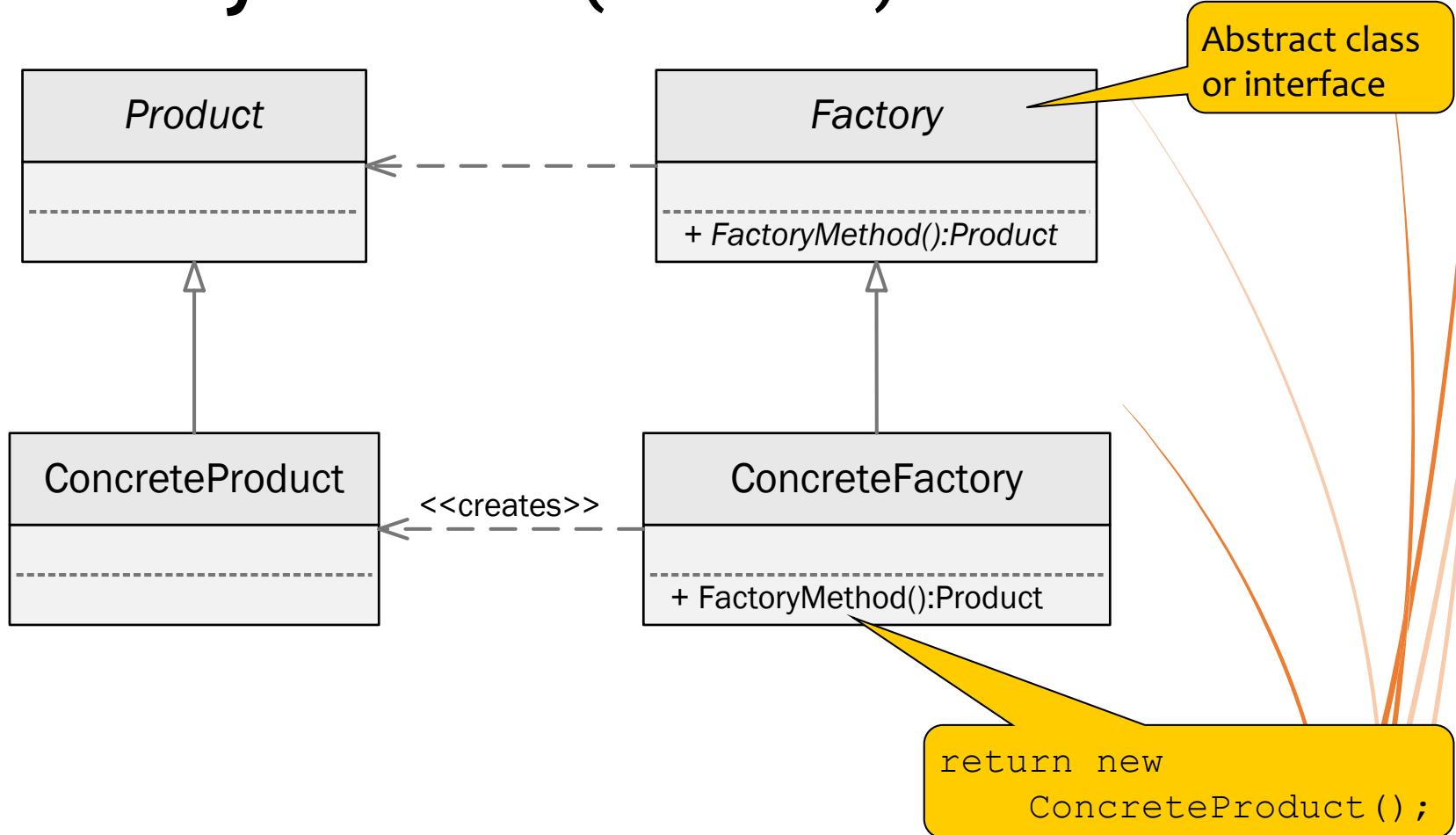
Factory method design pattern

Lecture notes

Factory method (Class Creational)

- Intent
 - Define an interface for creating an object
 - The factory method defers instantiation to subclasses
- Motivation
 - Need to create an object, but don't know which class to instantiate

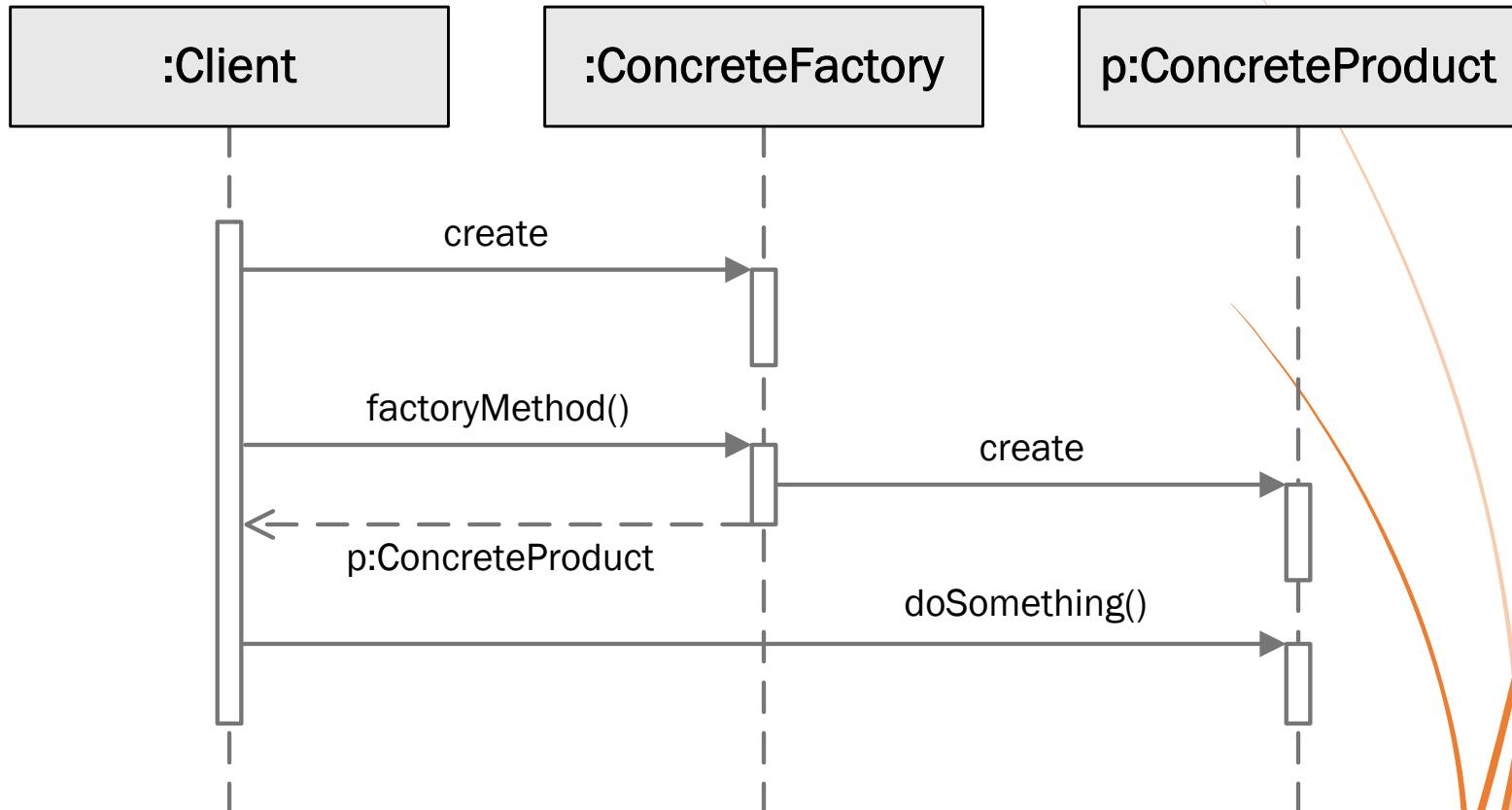
Factory method (Structure)



Factory method (Participants)

- Factory
 - Abstract class or interface
 - Declares the factory method
 - This might have a default implementation that returns a default ConcreteProduct object
- ConcreteFactory
 - Overrides the factory method to return a ConcreteProduct object
- Product
 - Defines the interface for a family of concrete product objects
- ConcreteProduct
 - Implements the Product interface

Factory method (Sequence diagram example)



Factory method (Variation)

- Design patterns are general solutions to general problems
 - They often need to be adjusted to provide a specific solution to a specific problem
- For example...
Parameterised factory methods
 - The factory method takes a parameter that identifies the kind of object to create

Factory method (Consequences)

- Advantages
 - Eliminate the need to bind the client to application-specific classes
 - client code only deals with the Product interface
 - i.e. it works with any user-defined ConcreteProduct object
 - Hide potentially complex creation logic
 - Can use performance-enhancing memory management strategies
 - e.g. object caching or recycling
- Possible disadvantages
 - Might need to create a subclass of the Factory class to create a particular ConcreteProduct object
 - This might introduce another point of evolution
 - i.e. choose which type of Create to use

Factory method (Example)

- An example of factory method is in the design patterns example

Design patterns

Lecture slides



In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write Java programs that demonstrate the concepts in the lecture notes
 - Singleton
 - Factory
 - Builder
 - Null Object
 - Table Data Gateway
 - Command
- Ask questions about anything you don't understand

Problem

- Modify the lecture demo from Week 10 to use the design patterns introduced in the lecture notes

Builder design pattern

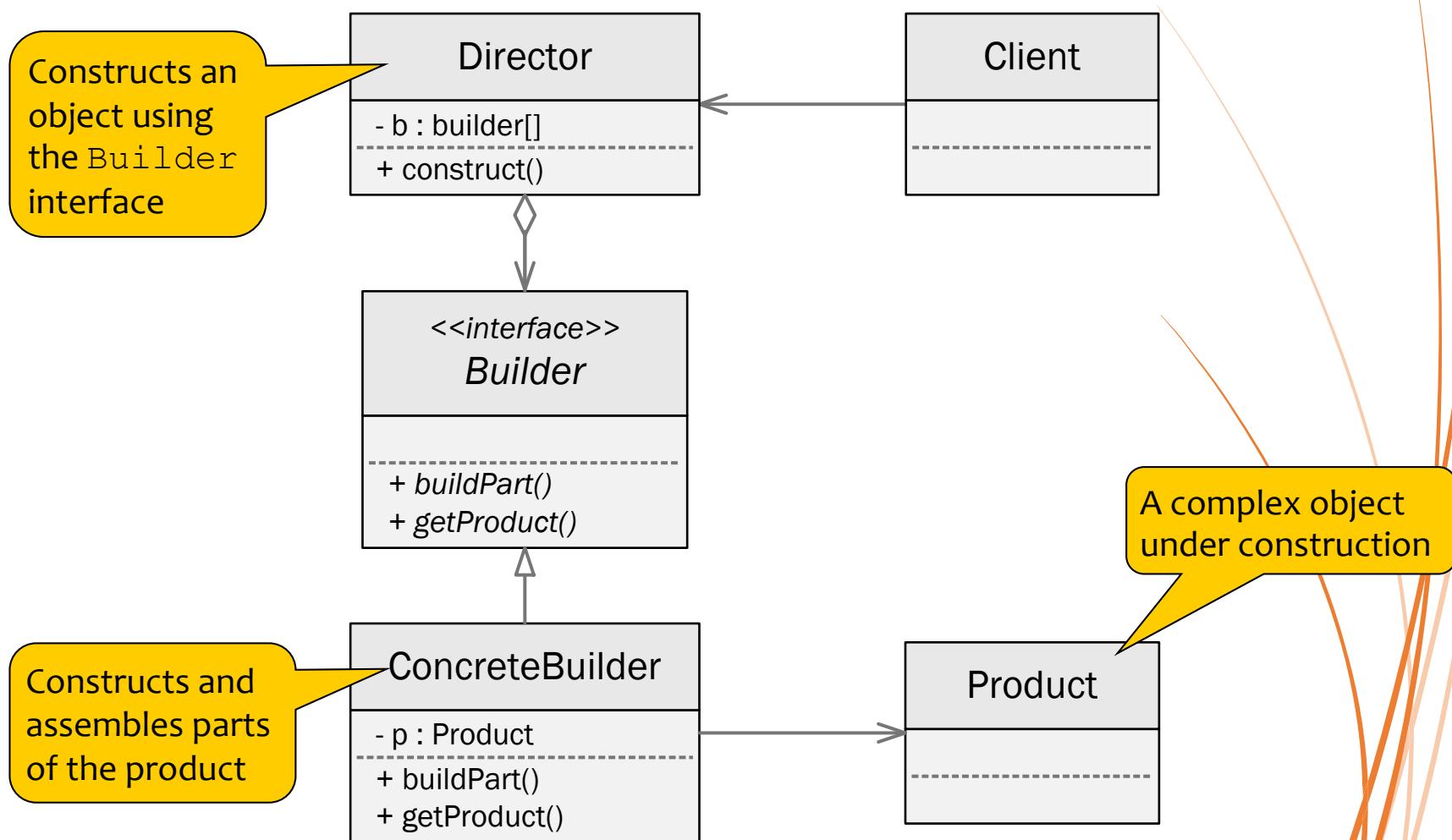
Lecture notes



Builder (Object Creational)

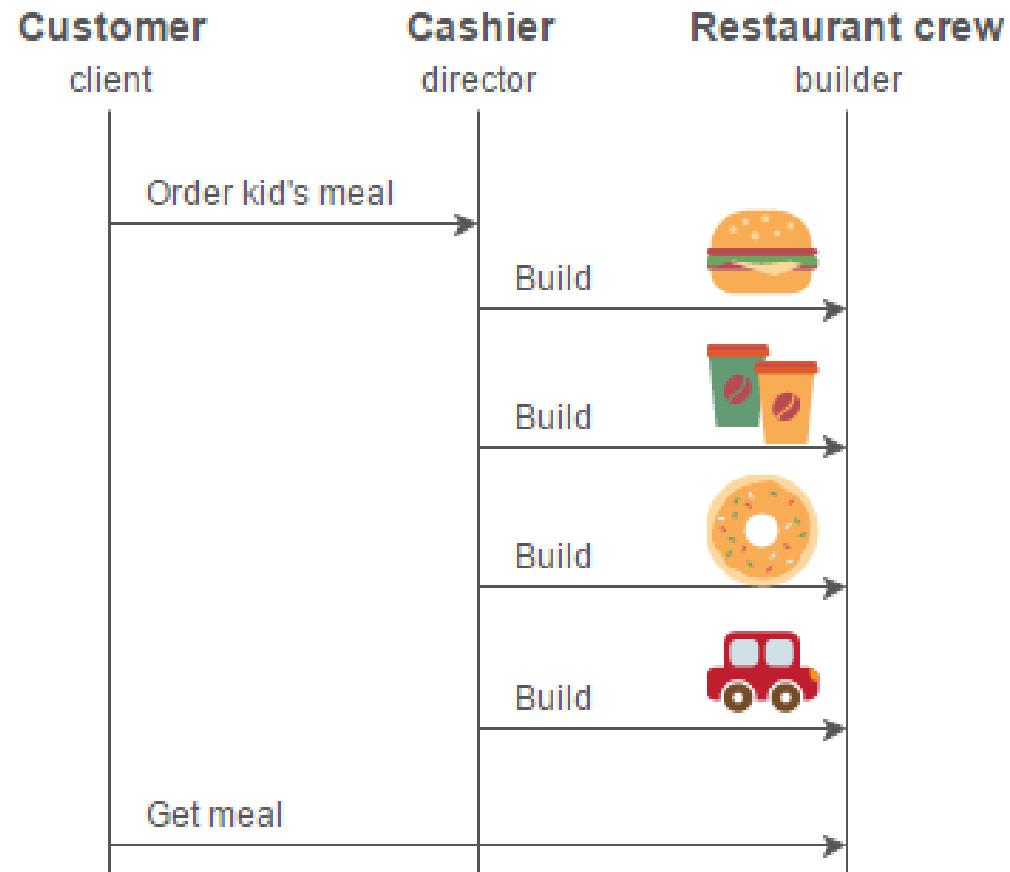
- Intent
 - Separate the construction of a complex object from its representation
 - the same construction process can then create different representations
- Motivation
 - An application needs to create a complex aggregate object
 - Not all parts of the aggregate object are available at the same time
 - i.e. the aggregate object must be built a bit at a time
 - The aggregate object might have different configurations

Builder (Structure)



Builder (Example)

- Client wants a kid's meal



- https://sourcemaking.com/design_patterns/builder

Builder (Consequences)

- Builder decouples construction of an object from its representation
 - It encapsulates the way a complex object is constructed
- Builder hides the internal representation of the complex product from the director
 - e.g. calling `withEmployee()` tells the director nothing about how the `Employee` object is held in the `Department` object being constructed
- It's easy to vary the elements required for construction
 - The director is given the final product only after all the required parts have been built

Builder method (Example)

- An example of builder is in the design patterns example

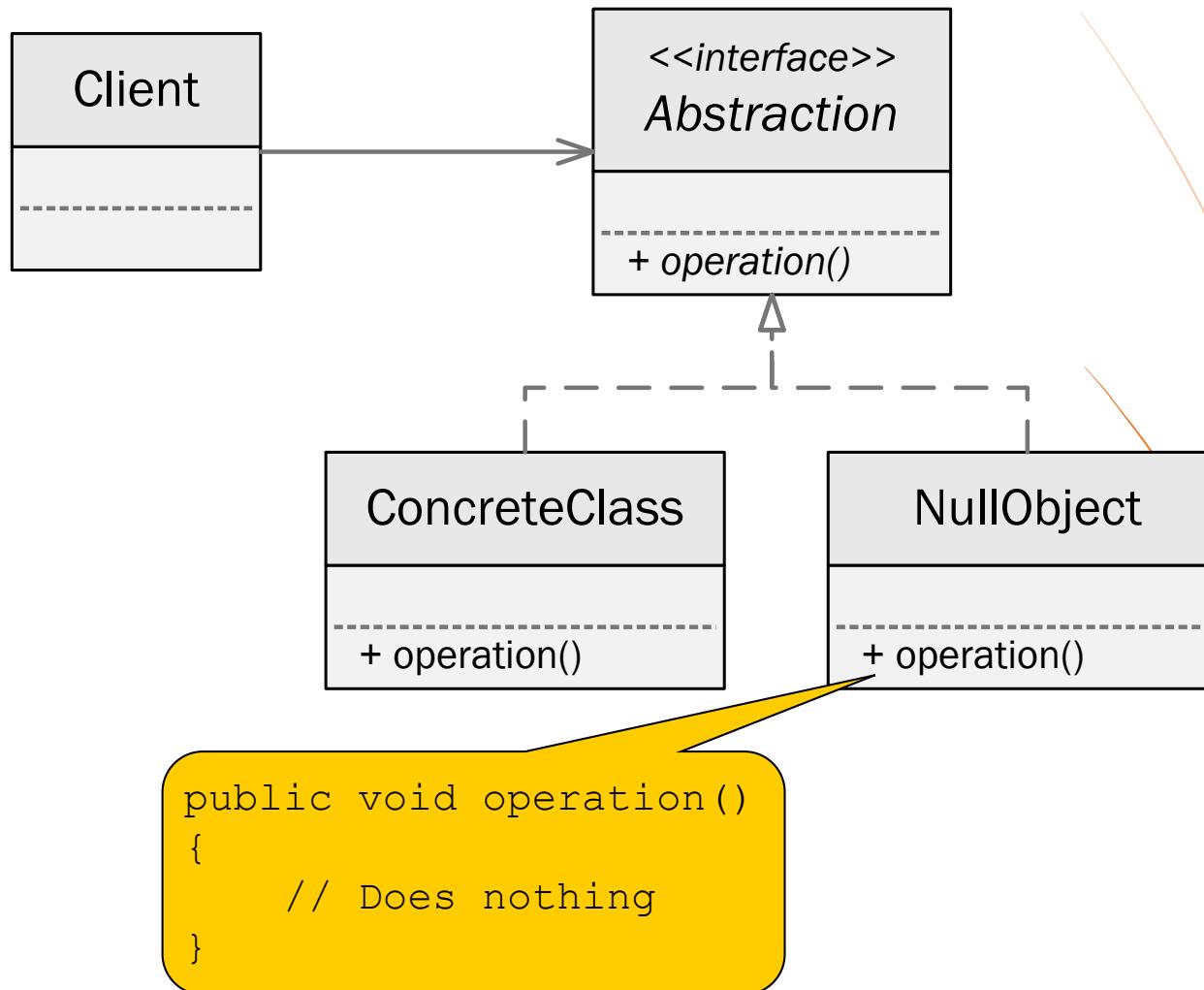
Null Object design pattern

Lecture notes

Null Object (Object Behavioural)

- Intent
 - Represents a null value with an object
 - thereby avoiding a null pointer exception
- Motivation
 - When using a null reference, a test must be performed to avoid generating a null pointer exception at run-time
 - The test...
 - takes up processing time
 - complicates the design

Null Object (Structure)



Null Object (Consequences)

- Null object helps to avoid exceptions at run-time
- Eliminating checks for null can make code easier to read
- Need to test that no code ever assigns null instead of Null Object
- Sometimes checks for Null Object are still necessary
 - Re-introduces an overhead

Null Object (Example)

- An example of null object is in the design patterns example

Table Data Gateway design pattern

Lecture notes

Table data gateway (Object Behavioural)

- Intent
 - To separate database access from use of the data by providing access to a table in a relational database
- Motivation
 - Database access code can be duplicated in many parts of the application when several classes all use the same database table
 - It is desirable to centralise database access

Table data gateway (Description)

- A simple interface with several standard methods
 - `update()`
 - `insert()`
 - `delete()`
 - A series of `find()` methods for querying the database
- Holds all the SQL for accessing a single table
 - Essentially a wrapper class for SQL statements
- Typically, one table data gateway class per table or view
 - Other classes call these methods for all interaction with the database table

Table Data Gateway (Example)

- An example of the table data gateway is in the design patterns example

Command design pattern

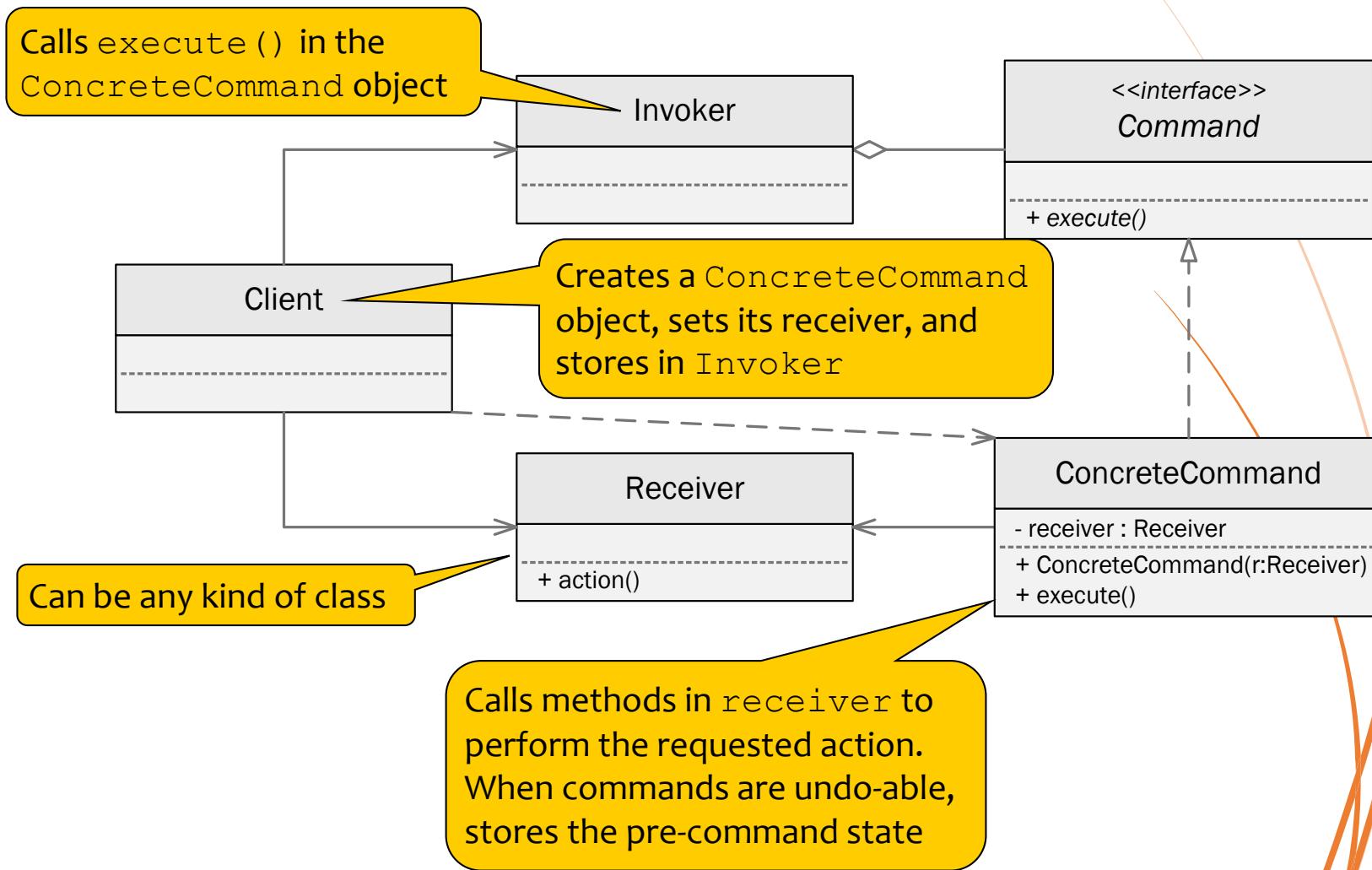
Lecture notes



Command (Object Behavioural)

- Intent
 - Encapsulate a request as an object, thereby enabling:
 - Clients to be parameterised with different requests
 - Queue or log requests
 - Support undo-able operations
- Motivation
 - An application that receives many types of request has a complicated method that processes the requests
 - e.g. A large selection
 - It is desirable to simplify the method

Command (Structure)



Command (Consequences)

- Command decouples the object that invokes the operation from the one that knows how to perform it
- Commands can be assembled into macro commands
 - i.e. a composite command
- It's easy to add new ConcreteCommand classes
 - existing classes do not change
- Can use a Factory Method to create commands
 - Decouple client even more

Activity



Coding

- Review the example code and watch out for:
 - how the design patterns described this week are used in the code
 - Singleton
 - Factory
 - Builder
 - Null Object
 - Table Data Gateway
 - Command
- If you have any questions, ask

Project: DesignPatternsExample_BeforePatterns
and DesignPatternsExample_AfterPatterns

Introduction to blackbox testing

Lecture notes

In these lecture notes

We will:

- Introduce the concepts of blackbox testing
- Introduce JUnit testing

Testing

- Testing software is a very important activity because it provides information about the quality of the code being tested
- Testing can show whether a piece of code...
 - Satisfies the software requirements
 - Functions as expected

Complete testing is impossible

- A software application can **never** be tested to exhaustion
- Consider a program that sums two integers
 - There is an infinite number of integers that could be input to the program
 - Therefore, there is an infinite number of tests that should be performed to prove that the program is correct
 - This is not possible to do in practice

Equivalence partitioning

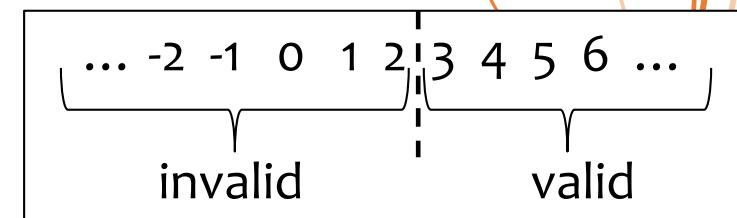
- Equivalence partitioning divides the set of all possible input values into groups of equivalent values, and then uses at least one value to represent each group
 - It is assumed that if a test passes with the representative values from the group, then all values from that group would pass the test
- Consider:
 - Add two integers
 - One group: all integers
 - Add two positive integers
 - Two groups: all positive integers; all negative integers

Black-box testing

- For each functional requirement, use equivalence partitioning to devise **test cases** that cover each of the following:
 - **Normal values** — Prefix the case number with N (e.g. N12)
 - Values that are drawn from the set of valid values for the application
 - **Invalid values** — Prefix the case number with I (e.g. I12)
 - Values that are drawn from the set of invalid values, the application should detect these as invalid
 - **Boundary values** — Prefix the case number with B (e.g. B12)
 - Values that test the boundaries between partitions
 - **Special cases** — Prefix the case number with S (e.g. S12)
 - Values that have not been considered by the other tests e.g. remove last item from a stack

Boundary values test cases

- A boundary exists between two adjacent equivalence groups
 - Use two test cases for a boundary, one each side of the boundary
- For the requirement num must be greater than two, we can identify the following
 - There are two partitions
 - Invalid values: **two** and below
 - Valid values: **three** and above
 - The values either side of the boundary are **two** and **three**
 - There are two boundary test cases:
 - B1: num = 2
 - B2: num = 3



The test plan

- A test plan consists of one or more tests
- A test consists of:
 - A unique identifier
 - for ease of reference
 - A set of instructions
 - The sequence of operations to perform in the test
 - A list of the test cases addressed by the test
 - To show the reason for choosing the test data
 - A description of the expected outcome of the test
 - So the tester can compare actual results against the expected results

Test coverage

- A single test covers:
 - A single invalid test case
or
 - A single special test case
or
 - One or more normal or boundary test cases
- Invalid and special cases are tested in isolation to avoid the possibility that one error masks another

Best practice in testing

- Use “easy” data values
 - e.g. it is easier to calculate

7 * 2

than

35463 * 453

- Do not assume an order of the tests in the plan
 - Each test must be independent of all other tests
 - This avoids side effects – i.e. one test does not affect another, possibly masking problems
- Keep the test plan as small as possible while ensuring all cases are covered
 - This avoids unnecessary tests

Test plan example

Problem

A sports team has a name, and consists of a number of players. When the team has the required number of players, it can compete.

A player has a name, a team and a position. The position may be “Attack” or “Defence”, and is set only when the player is allocated to a team. If the player is not allocated to a team, the position must be blank (i.e. “”).

A player can only be in one team at a time. A player cannot be added to a team if he or she is already in the team.

A team cannot have more than the required number of players, although it can have fewer.

A team’s required number of players must be one or more.

Write and test Java classes to implement these requirements.

Activity



Coding

- Review the example test plan and watch out for:
 - how the requirements are derived from the problem description
 - how the test cases are derived from the requirements
 - how the test cases address all requirements
 - how the test plan is derived from the test cases
 - how the test plan covers all test cases
- If you have any questions, ask

File: 12a_LectureNotes_TestPlanExample.pdf

Lecture Notes 12a: Test plan example

Problem

A sports team has a name and consists of a number of players. When the team has the required number of players, it can compete.

A player has a name, a team and a position. The position may be “Attack” or “Defence” and is set only when the player is allocated to a team. If the player is not allocated to a team, the position must be blank (i.e. “”).

A player can only be in one team at a time. A player cannot be added to a team if he or she is already in the team.

A team cannot have more than the required number of players, although it can have fewer.

A team’s required number of players must be one or more.

Write and test Java classes to implement these requirements.

Requirements

- R1 A team has a name
- R2 A team has a required number of players, which must be one or more
- R3 A player has a name
- R4 A player’s position is either “Attack” or “Defence” only when in a team
- R5 A player’s position must be blank (i.e. “”) when the player is not in a team
- R6 A team has zero or more players
- R7 A player can be added to a team
- R8 A player is in no team
- R9 A player is in one team
- R10 A player cannot be in more than one team at a time
- R11 A player cannot be added to a team if already in the team
- R12 A player can be removed from a team
- R13 When the number of players in a team is equal to the team’s required number of players, the team can compete
- R14 A team cannot have more than the required number of players

Test cases

Normal values

- N1 Team name = "A-Team" (R1)
- N2 Required number of players = 5 (R2)
- N3 Player name = "Jimmy" (R3)
- N4 Player position = "Attack" (R4)

Invalid values

- I1 Team name = "" (R1)
- I2 Required number of players = -4 (R2)
- I3 Player name = "" (R3)
- I4 Player position = "Wing" (R4)

Boundary values

- B1 Required number of players = 0 (R2)
- B2 Required number of players = 1 (R2)
- B3 Number of players in a team is one less than the team's required number of players (R13)
- B4 Number of players in a team is equal to the team's required number of players (R13)

Special cases

- S1 Add player (Jimmy) to team (A-Team) in position "Defence" (R4, R7, R9)
- S2 Player (Jimmy) is removed from team (A-Team) (R5, R8, R12)
- S3 Remove a player from a team that has no players (R6)
- S4 Add a player to a team that already has the required number of players (R14)
- S5 Add a player to a team (B-Team) when the player is already in team (A-Team) (R10)
- S6 Add a player to a team (A-Team) when the player is already in team (A-Team) (R11)

Test plan

Test plans should not be cumulative; i.e. a test must not rely on the prior successful execution of another test. Each test should be independent of all other tests.

Test	Instruction (Satisfies test cases)	Expected result	Actual result
T1	Create a team called "A-Team" requiring 5 players (N1, N2)	There is a team whose name is "A-Team" requiring 5 players	Pass
T2	Create a player named "Jimmy" (N3)	There is a player whose name is "Jimmy", and whose position is blank	Pass
T3	Create a team with a blank name requiring 5 players (I1)	Exception: team not created	Fail: team created with blank name. Pass after correction
T4	Create a team called "Z-Team" requiring -4 players (I2)	Exception: team not created	Pass
T5	Create a player with a blank name (I3)	Exception: player not created	Pass
T6	Create a team called "Y-Team" requiring zero players (B1)	Exception: team not created	Pass
T7	Create a team called "B-Team" requiring 1 player (B2, B3)	There is a team whose name is "B-Team" requiring 1 player The team cannot compete	Pass
T8	Create a player named "Jimmy" Create a team called "B-Team" requiring 1 player Add player "Jimmy" to team "B-Team" in position "Defence" (B4, S1)	Team "B-Team" has only one player, and that player's name is "Jimmy" The team can compete Player "Jimmy" has a team with name = "B-Team", and a position = "Defence"	Pass

Lecture Notes 12a: Test plan example

Test	Instruction (Satisfies test cases)	Expected result	Actual result
T9	Create a player named "Jimmy" Create a team called "B-Team" requiring 1 player Add player "Jimmy" to team "B-Team" in position "Defence" Create player ("John"), and add this new player to team "B-Team" in position "Attack" (S4)	Player "John" is not added to team B-Team Player "John" does not have a team, and a position that is blank The team "B-Team" has one player whose name is "Jimmy"	Pass
T10	Create a player named "Jimmy" Create a team called "B-Team" requiring 1 player Add player "Jimmy" to team "B-Team" in position "Defence" Remove player "Jimmy" from team "B-Team" (S2)	Team "B-Team" has no players The team cannot compete Player "Jimmy" has no team, and a position that is blank	Pass
T11	Create a player named "Jimmy" Create a team called "B-Team" requiring 1 player Add player "Jimmy" to team "B-Team" in position "Defence" Remove player "Jimmy" from team "B-Team" Remove player "Jimmy" from team "B-Team" (S3)	Player "Jimmy" is not removed The team "B-Team" has no players	Pass
T12	Create a team called "A-Team" requiring 5 players Create a team called "B-Team" requiring 1 player Create a player named "Jimmy" Add player "Jimmy" to team "A-Team" in position "Attack" Add player "Jimmy" to team "B-Team" in position "Defence" (N4, S5)	Team "A-Team" has only one player, and that player's name is "Jimmy" Player "Jimmy" has a team with name = "A-Team", and a position = "Attack" Team "B-Team" has no players, and cannot compete	Pass

Lecture Notes 12a: Test plan example

Test	Instruction (Satisfies test cases)	Expected result	Actual result
T13	Create a team called "A-Team" requiring 5 players Create a player named "Jimmy" Add player "Jimmy" to team "A-Team" in position "Attack" Add player "Jimmy" to team "A-Team" in position "Defence" (S6)	Player "Jimmy" is not added Team "A-Team" has only one player, and that player's name is "Jimmy" The team cannot compete Player "Jimmy" has a team with name = "A-Team", and a position = "Attack"	Pass
T14	Create a team called "A-Team" requiring 5 players Create a player named "John" Add player "John" to team "A-Team" in position "Wing" (I4)	Player "John" is not added, has no team, and a position that is blank Team "A-Team" has no players	Pass

Introduction to JUnit testing

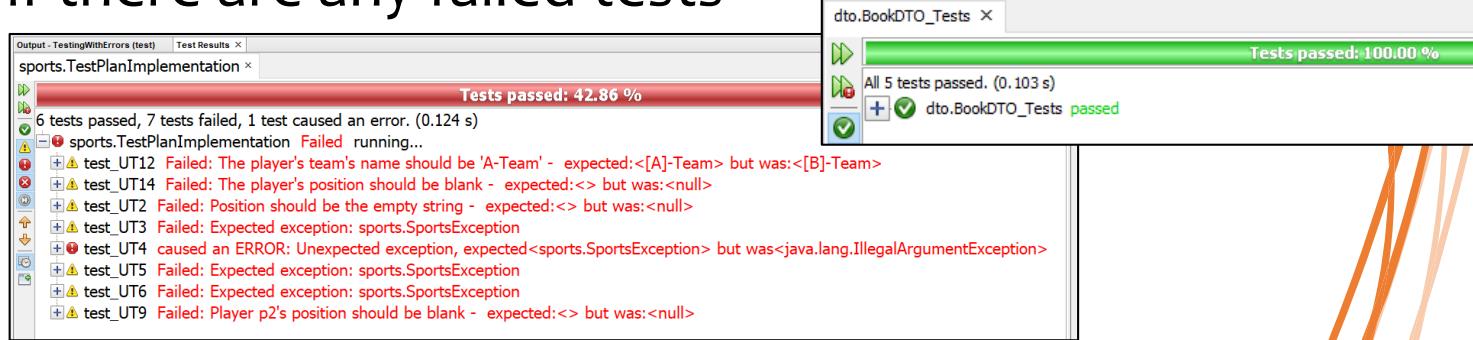
Lecture notes

Implementing the test plan

- One way of implementing a test plan is to write an application class
 - Each test is written in its own method, outputting the results to the console window
 - The `main()` method calls each test in turn
- Advantage
 - Conceptually very easy
- Disadvantage
 - Success or failure of each test can only be detected by detailed visual inspection of the output
 - What if there are 10,000 tests in the plan?

Implementing the test plan – JUnit

- Another way of implementing a test plan is to use the JUnit framework
 - A popular test environment for Java programs
 - Automates the execution and reporting of tests
 - Ships with the JDK
- Success or failure of each test is detected and displayed automatically
 - Only a quick visual inspection is required to know if there are any failed tests



JUnit improves the testing process

- Each test that fails can be considered in turn and corrected
- After each correction, the test class can be executed again
 - Testing has become easier
 - Testing has become consistent

JUnit test class

- A JUnit test class has this basic structure

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*;  
  
public class TeamAndPlayerTesting  
{  
}
```

- Each test is written as a method in this class
 - The test is manually written
 - When the class is run, the tests are automatically executed and their outcomes reported

JUnit test method

- A JUnit test method has this basic structure

```
@Test  
public void test_T1()  
{  
}
```

An annotation that identifies this as a test method

- The test class can have many test methods
 - The JUnit framework will automatically execute all test methods when the class is run
- There is no guarantee of the order in which the tests will be performed
 - Hence the Best Practice advice given earlier

Using JUnit in NetBeans

- Create a Java class in the Test Packages section of the NetBeans project

```
package com.gdm1.teamandplayerwithtesting;  
  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*;  
  
public class TeamAndPlayerTesting  
{  
    @Test  
    public void test_T1()  
    {  
    }  
}
```

Putting this class in this package means there is no need to import the Team and Player classes

Using assertions for verification

- A test method can use assertions to verify the truth of conditions
 - If an assertion is incorrect, the test immediately fails and the assertion's error message is output
 - The test only passes if all its assertions are correct
- Some assertion methods

```
assertEquals(expectedObject, actualObject);  
  
assertTrue(booleanExpression);  
assertFalse(booleanExpression);  
  
assertNull(objectReference);  
assertNotNull(objectReference);  
  
assertSame(expectedObject, actualObject);  
assertNotSame(expectedObject, actualObject);
```

Testing for exceptions

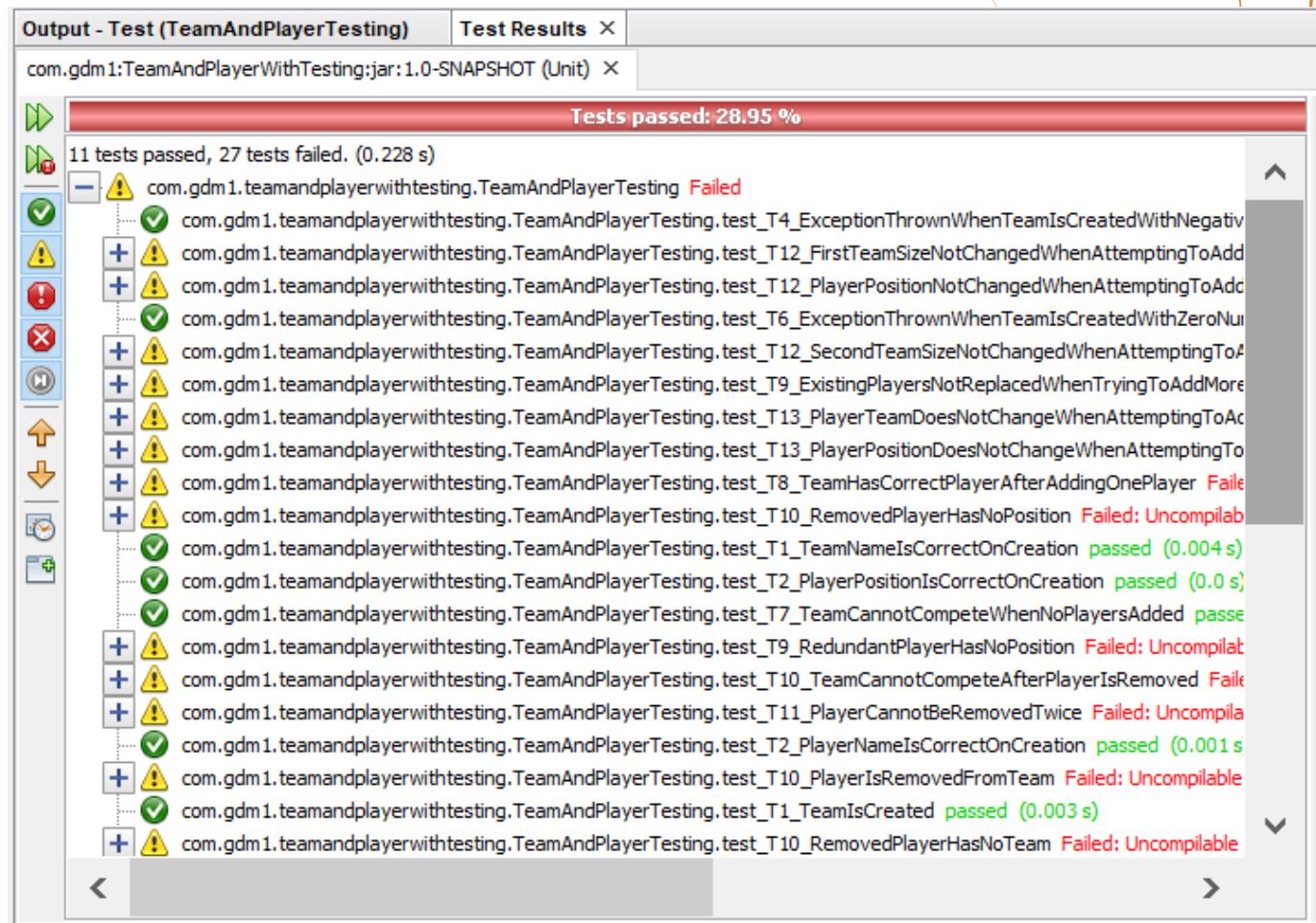
- In some tests, an exception is the desired outcome
 - e.g. Creating a team with a blank name
 - To assert that an exception is the correct result of the test...

```
@Test  
public void test_T3()  
{  
    assertThrows(  
        IllegalArgumentException.class,  
        () -> new Team("", 5));  
}
```

The test passes only if this exception is generated

Running the JUnit test class

- Running the JUnit test class produces test results such as:



Running the JUnit test class

- Each test that fails can be considered in turn and corrected
- After each correction, the test class can be executed again
 - Testing has become easier
 - Testing has become consistent

Activity



Coding

- Review the example code and watch out for:
 - how the JUnit tests are implemented
 - how assertions are used to verify correct values
 - how to assert that an exception is the correct outcome
 - only having one assertion per test
- If you have any questions, ask

Project: TeamAndPlayerWithTesting

Threading

Lecture slides



In this lecture

- You should have read the lecture notes before coming to this lecture
- We will write a Java class that demonstrates synchronisation as described in the lecture notes
- Ask questions about anything you don't understand

Problem

- First, in Lecture13Demo, review the Buffer class and then modify it to use synchronized blocks that are as small as possible

Introduction to threading

Lecture notes



In these lecture notes

We will:

- Introduce threads
 - the Thread class
 - the Runnable interface
- Consider Thread states and the transitions between them
- Discuss thread synchronisation

Multi-processing vs Multi-threading

- Operating systems (OS) often allow multiple processes to co-exist
 - The processes share the processor and are isolated from each other by the OS
 - They should not be able to overwrite each other's memory
- Languages like Java, C++ and C# allow an application to have more than one thread of execution
 - A thread is a path of execution through a program
 - It is the smallest unit of execution in the OS
 - Multiple threads in an application share the same process space and have access to the same memory

Why use threads?

- A lot of the time a computer is doing nothing!
 - Often waiting for user input or a slow peripheral device
- A computer can use any spare time by:
 - Multi-processing – more than one process using the CPU
 - Multi-threading - more than one thread using the CPU
- Threads can be used to:
 - Allow the application to do multiple tasks concurrently
 - e.g. start processing data before all the data has loaded
 - e.g. maintain an active user interface so the interface does not freeze while processing takes place

Threads in Java

- There are two ways of creating threads in Java:
 - Write a subclass of Thread
 - This is appropriate when the work of the thread is mostly independent of other objects
 - Write a class that implements Runnable
 - This is appropriate when the work of the thread is linked to the context of a specific object
 - An instance of the Thread class is constructed using an instance of the Runnable class as an argument
- In either case the key method is `run()`
 - The life of the thread exists through the execution of `run()`

Example: Extending Thread

```
public class MyThread extends Thread
{
    private String msg;
    private int num;

    public MyThread(String m, int n)
    {
        msg = m;
        num = n;
    }

    public void run()
    {
        // the work of the thread is done here
        // the thread terminates when this method ends
    }
}
```

Example: Implementing Runnable

```
public class SimpleRun implements Runnable
{
    private String msg;
    private int num;

    public SimpleRun(String m, int n)
    {
        msg = m;
        num = n;
    }

    public void run()
    {
        // the work of the thread is done here
        // the thread terminates when this method ends
    }
}
```

Starting a thread

- Although the thread lives through the `run()` method, it is started by a call to the `start()` method

```
MyThread t1 = new MyThread("Hello", 5);  
t1.start();  
  
Thread t2 = new Thread(new SimpleRun("Pete", 20));  
t2.start();
```

- Calling `start()` will cause the JVM to set up the thread and invoke the `run()` method

Multiple threads sharing a processor

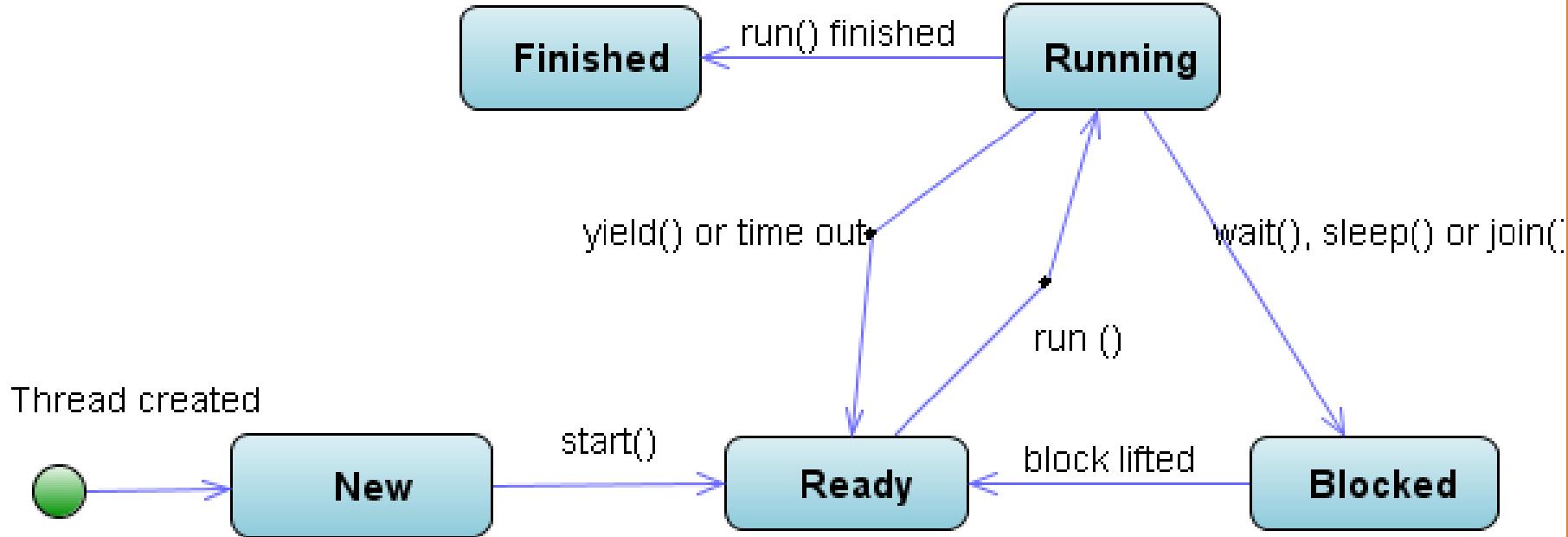


- The JVM schedules the threads
 - Tells them when to start
 - Tells them when their time slice ends
- Many computers now have multiple cores
 - JVM still schedules the threads, which might execute concurrently

Thread states and priorities

Lecture notes

Thread states



- `isAlive()` returns true if the thread is ready, blocked or running
 - i.e. the thread has started and has not yet finished

Thread states

- When a thread is created it is in the New state
- Calling `start()` moves it to the Ready state
- The thread might not yet be running
 - It is waiting for its allocated time slice
- The JRE invokes the `run()` method – Running state
- When `run()` finishes, the thread moves to the Finished state
- If the thread uses up its allocated time before `run()` finishes, it goes back to the Ready state
- A Running thread can be asked to yield
- The static method `Thread.yield()` moves the currently running thread back to the Ready state

Thread priorities

- The Thread class has `setPriority(int p)`
 - Priority range is 1 to 10
 - Thread priority constants
 - `Thread.MIN_PRIORITY` 1
 - `Thread.NORM_PRIORITY` 5
 - `Thread.MAX_PRIORITY` 10
- A thread has the same priority as the thread that spawned it
- The thread executing `main()` has `Thread.NORM_PRIORITY`

Allocating time on the processor

- The JVM selects the runnable thread with the highest priority
 - if two or more have the same priority, they are allocated time slices in turn
- If a high priority thread is running, lower priority threads will never get a chance to run until it finishes
 - As soon as the high priority thread times out, the JVM will select it to run again
 - This is called thread starvation
 - Avoid it by periodically asking the high priority thread to sleep to give other threads a time slice

Threading example



Coding

- Review the example code and watch out for:
 - how the array is (inefficiently) sorted
 - first without threading
 - then with threading
 - the timing of the two sorts (when you run the program)
 - the way the threads are created and run
- If you have any questions, ask

Project: ThreadingExample

Thread sleep() and join()

Lecture notes

Blocking a thread

- A thread in the Running state can move to the Blocked state
 - `run()` method stops executing
- The thread will not get time on the processor while in the Blocked state

Reason for entering Blocked state	Move to Ready state when
<code>sleep()</code> method called	<code>sleep()</code> times out
<code>join()</code> method called	target thread finished
<code>wait()</code> method called	notification received via <code>notify()</code> or <code>notifyAll()</code>

Thread.sleep (long millis)

- A static method in the Thread class
 - Puts the currently running thread in the Blocked state
 - millis is the number of milliseconds to sleep before moving to the Ready state
 - Can throw an InterruptedException,
 - It is a checked exception, so it must be handled
 - Usually do nothing with it
 - To pause a running thread for 1 second:

```
public void run() {  
    // some code to execute  
    try { Thread.sleep(1000); }  
    catch (InterruptedException e) { }  
    // more code to execute  
}
```

join()

- join() is a Thread object method
 - Forces one thread to wait for another one to finish
- For two thread objects, t1 and t2
 - In the run() method of t1 ...

```
t2.join();
```

will force t1 to be blocked until t2 reaches
the Finished state
i.e. its run() method has completed

Potential problems with threads

Lecture notes



Potential problems with threads

- Access to shared resources by multiple threads may lead to:
 - Deadlock
 - Two or more threads are each waiting for the other to relinquish a resource
 - Livelock
 - Two or more threads are responding to each other
 - Race conditions
 - Two or more threads are trying to use the same data
 - The outcome depends on the order the threads access the data
 - Thread starvation
 - A thread never gets a chance to use a shared resource because higher priority threads have locked access to it

synchronized keyword

Lecture notes

Concurrent data access

- Problems can occur if multiple threads can modify the same data
- Consider this class
 - What can go wrong when several threads execute in these methods?

```
public class Resource
{
    private int credit = 5;

    public void up( )
    {
        credit++;
    }

    public void down( )
    {
        credit--;
    }

    public int getCredits( )
    {
        return credit;
    }
}
```



Concurrent data access

- `up()` and `down()` are not atomic operations
 - First, the value of `credit` is retrieved from memory
 - Then, 1 is added or subtracted from it
 - Finally, the new value is stored back in `credit`
- Suppose thread A moves from the Running state in the middle of the `up()` method
 - just after the value of `credit` is retrieved – say it is 3
- Then thread B enters the Running state and starts executing `down()`
- When thread B finishes, `credit` has been decrease to 2
- When thread A starts again, it will increase the 3 value it retrieved to 4
- Thread B's execution of `down()` has had no effect
- What if thread B left the running state before finishing `down()`?
- What if thread B had executed `up()` instead?

What's the solution?

- Use synchronization
- When a thread enters a synchronized block, it holds a lock on the entire object
- No other thread can enter **any** synchronized blocks on the same object until the first exits
- Use the keyword `synchronized` to make a method synchronized
 - Locks the entire object
 - No other synchronized method can be called on the same object while the lock is held

Defining synchronized methods

```
public class Resource
{
    private int credit = 5;
    public synchronized void up( )
    {
        credit++;
    }
    public synchronized void down( )
    {
        credit--;
    }
    public synchronized int getCredit( )
    {
        return credit;
    }
}
```



Use of synchronized methods

- Any method can be synchronized, except the constructor
- To prevent problems in objects that are accessed by multiple threads, decide which methods should be synchronized
 - But this can cause decrease in performance, deadlock or livelock
- Not necessary for attributes marked final
 - Because they can't be modified

Potential deadlocks

- A thread might be suspended in a synchronized block until some condition is fulfilled
- Example: cannot decrease credit if it is already zero
 - `down()` needs to suspend until another thread calls `up()` to increase the credit
- Calling `Thread.sleep()` will put it in the Blocked state, but won't release the lock
 - Deadlock!
 - No other thread can access the object to increase credit

Avoiding deadlocks

- Instead, the thread in `down()` should call `wait()` while in the synchronized method
 - It will move to the Blocked state and release the lock
- A thread can release the block on all waiting threads by calling `notifyAll()`
 - Any waiting threads move to the Ready state and can resume operation

wait() and notifyAll()

Lecture notes

wait() and notifyAll()

```
public class Resource {  
    private int credit = 5;  
    public synchronized void up() {  
        credit++;  
        notifyAll();  
    }  
    public synchronized void down() {  
        while (credit <= 0) {  
            try { wait(); }  
            catch (InterruptedException e) {}  
        }  
        credit--;  
    }  
    public synchronized int getCredit() {  
        return credit;  
    }  
}
```

wait() and notifyAll()

- Both methods are inherited from the Object class
 - so can be called on any object
- wait()
 - Should only be called from inside synchronized code
 - Moves the currently executing thread to the Blocked state, allowing another thread to enter the Running state
- notifyAll()
 - When a thread has the lock on an object and calls notifyAll(), all threads waiting for this lock will move into the Ready state
 - When each of these threads move to the Running state, they attempt to re-acquire the lock and resume where they left off

wait() and notifyAll()

- In our example, `wait()` is inside a `while` loop
- When the thread resumes, the condition for waiting might still be true
 - So need to wait again

```
public synchronized void down()  
{  
    while (credit <= 0) {  
        try { wait(); }  
        catch (InterruptedException e) {}  
    }  
    credit--;  
}
```

Synchronized blocks

Lecture notes



Synchronized blocks

- Synchronized methods lock and unlock the object they are operating on
- It is also possible to synchronize a block of code
 - The lock is based on a specific object

```
public void down()  
{  
    // some code  
    synchronized (someObject)  
    {  
        // only one thread at a  
        // time in the block  
    }  
    // some more code  
}
```

The object that holds the lock

Synchronized blocks or methods?

- Entering a synchronized section of code locks out other threads
 - This is true of both synchronized blocks and synchronized methods
- Therefore, the guiding principle should be...

Make the synchronized section of code as short as possible

Synchronized blocks

Lecture notes



Synchronized blocks

- Synchronized methods lock and unlock the object they are operating on
- It is also possible to synchronize a block of code
 - The lock is based on a specific object

```
public void down()  
{  
    // some code  
    synchronized (someObject)  
    {  
        // only one thread at a  
        // time in the block  
    }  
    // some more code  
}
```

The object that holds the lock

Synchronized blocks or methods?

- Entering a synchronized section of code locks out other threads
 - This is true of both synchronized blocks and synchronized methods
- Therefore, the guiding principle should be...

Make the synchronized section of code as short as possible

More concurrent programming

Lecture slides



In this lecture

- You should have read the lecture notes before coming to this lecture
- There is no lecture demonstration
 - Just a double-length practical session
- Ask questions about anything you don't understand

More on concurrent programming

Lecture notes



In these lecture notes

We will:

- Introduce
 - the Callable interface
 - the ExecutorService interface
 - the java.util.concurrent.atomic package
- Give an overview of the Fork/Join framework
- Give an overview of using parallel streams

The Callable interface

- The Callable interface is similar to the Runnable interface...

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

```
public interface Runnable
{
    void run();
}
```

- Both represent tasks that can be executed by a thread
 - but Callable
 - returns a result
 - can throw a checked exception

The Callable interface

- Example

```
import java.util.concurrent.Callable;

public class MyCallableClass
    implements Callable<String>
{
    @Override
    public String call()
    {
        // the code for the task
    }
}
```

Calling a Callable

- To execute the Callable task asynchronously
 - Instantiate the class
 - Pass the instance to an ExecutorService object, which will
 - Set up an asynchronous thread and call the `call()` method
 - Receive the return value from the `call()` method and put it into a Future object
 - Get the Future object and retrieve the result

The Future interface

- The Future interface

```
public interface Future<V>
{
    boolean cancel(boolean mayInterruptIfRunning);
    V get();
    V get(long timeout, TimeUnit unit);
    boolean isCancelled();
    boolean isDone();
}
```

Cancels the asynchronous task that will give the result

Returns the result provided by the task

Returns true if the task was cancelled

Returns true if the task has completed

- Calling `get()` before the task completes will block until the result is ready
- Calling `get(timeout, unit)` before the task completes will block until the result is ready or the timeout period expires, whichever is shorter

The ExecutorService interface

- A service that provides methods to
 - Execute Runnable objects, including Thread objects, returning a Future object
 - Execute Callable objects , returning a Future object
 - Shutdown the service, which means it will not accept any more tasks
- NB:

The service must be shutdown, otherwise your application will continue to run after the main () method has completed

Some ExecutorService methods

- To execute asynchronous tasks

- `void execute(Runnable task)`
 - Cannot obtain the result
- `Future<?> submit(Runnable task)`
 - The Future's get () method returns null
- `Future<T> submit(Callable<T> task)`
 - The Future's get () method returns the task's result
- `List<Future<T>> invokeAll(`
`Collection<Callable<T>> tasks)`
 - Results of the tasks are in the List of Futures

Some ExecutorService methods

- To terminate the executor service

- `void shutdown()`

- Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted
 - Does not wait for previously submitted tasks to complete execution

- `boolean awaitTermination(long timeout, TimeUnit unit)`

- Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first

Get an ExecutorService object

- The Executors class provides factory methods to create ExecutorService objects, including

- ```
public static ExecutorService
 newCachedThreadPool()
```

- Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available

- ```
public static ExecutorService  
    newFixedThreadPool(int nThreads)
```

- Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue

Activity



Coding

- Review the example code and watch out for:
 - how the `ArraySortCallable` class implements `Callable` interface
 - how the `ExecutorService` is created and used to `execute` `Callables` in the `ArraySorterWithCallables` class
 - how the `ExecutorService` is used to execute `Threads` in the `ArraySorterWithThreads` class
- If you have any questions, ask

Project: CallableExample

The atomic package

Lecture notes



The need for the `atomic` package

- Multi-threaded applications can have problems resulting from using shared resources
 - See Week 13
- For example, if several threads are using a variable, each might
 - Check the current value and then set the value
 - This is a *check-then-act* operation
- This can lead to race conditions
 - Multiple threads are trying to use the same data
 - The outcome depends on the order in which the threads access the data

The need for the `atomic` package

- A *check-then-act* operation needs to be `atomic`
 - Both parts of the operation must be executed as an indivisible block of code
 - Any thread that starts the *check-then-act* operation will finish executing it without interference from other threads
 - i.e. no other thread can execute the atomic operation at the same time
- In Week 13, the solution to this problem was to use synchronized methods or blocks

The atomic package

- The `java.util.concurrent.atomic` package
 - Provides a small toolkit of classes that support lock-free thread-safe programming on single variables
- The classes in this package provide atomic conditional update operations for fields and array elements

The AtomicInteger class

- An int value that may be updated atomically
- It has two constructors

```
import java.util.concurrent.AtomicInteger;

public class MyClass
{
    private AtomicInteger num1
        = new AtomicInteger();

    private AtomicInteger num2
        = new AtomicInteger(45);

    ...
}
```

The contained
int value is
initialised to
zero

The contained
int value is
initialised to
45

Some AtomicInteger methods

- All are atomic operations

- ```
public final int addAndGet(int delta)
```

- Atomically adds the given value to the current value

- ```
public final int getAndIncrement()
```

- Atomically increments by one the current value

- ```
public final int incrementAndGet()
```

- Atomically increments by one the current value

Returns the previous value

Returns the updated value

# Some AtomicInteger methods

- ```
public final int getAndAccumulate(  
    int x, IntBinaryOperator accumulatorFunction)
```

 - Atomically updates the current value with the result of applying the accumulator to the current and given values
 - Returns the previous value
- ```
public final int accumulateAndGet(
 int x, IntBinaryOperator accumulatorFunction)
```

  - Atomically updates the current value with the result of applying the accumulator to the current and given values
  - Returns the updated value

# Some AtomicInteger methods

- ```
public final boolean compareAndSet(int expect,
                                    int update)
```

- Atomically sets the value to update value if the current value == expect
- Returns true if successful
- Returns false if the current value != expect

- There are more methods
 - Check the [documentation](#) for details
- There are more atomic classes
 - Check the [package documentation](#) for details

Activity



Coding

- Review the example code and watch out for:
 - how classes `ArraySortThread` and `ArraySortCallable` each use an `AtomicInteger` object
- If you have any questions, ask

Project: CallableExample

Overview of the Fork-Join framework

Lecture notes



The fork and join principle

- This consists of two steps that are performed recursively
 - Fork
 - A (large) task will *fork* (split) itself into smaller tasks that can be executed concurrently
 - Smaller tasks could also fork
(this is the recursive nature of fork)
 - Join
 - The (large) task will wait for the smaller forked tasks to complete and then will *join* (merge) the results (if any) into a single result
 - Smaller tasks will join the results from the sub-tasks they forked before completing
(this is the recursive nature of join)

The ForkJoinPool class

- This is an ExecutorService for running ForkJoinTasks
 - If you know how to use an ExecutorService, you can work out how to use a ForkJoinPool
 - See the [documentation](#)
- A ForkJoinTask is an abstract base class for tasks that run within a ForkJoinPool
 - ForkJoinTask objects can be derived by
 - Writing and instantiating sub-classes
 - Calling static methods to adapt Runnables and Callables
 - See the [documentation](#)

Some additional reading

- Tutorial by Jakob Jenkov
<http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html>
- “A Java Fork-Join Calamity”
<http://coopsoft.com/ar/CalamityArticle.html>
- “When to use ForkJoinPool vs ExecutorService”
<https://www.infoworld.com/article/2078440/java-tip-when-to-use-forkjoinpool-vs-executorservice.html>
- Google search
[java fork join review](#)

Overview of parallel streams

Lecture notes



Java Streams API

- In Weeks 8 & 9, we looked at the Java streams API
 - A stream is a sequence (possibly infinite) of values (elements)
 - A stream supports sequential and parallel aggregate operations
 - An operation is applied to all the elements in the stream
 - Operations are one of...
 - *Intermediate* – produce another stream
 - *Terminal* – produce a value or a side-effect

Sequential vs parallel streams

- A sequential stream uses a single thread to process the stream's pipeline
 - Its operations are performed one by one
 - It does not take advantage of any underlying multi-core system that could support parallel execution
- A parallel stream uses multiple threads to process the stream's pipeline
 - The threads can be executed concurrently on separate cores of the system, combining the separate results into one
 - The order of execution is not predictable and can give unordered results

Parallel streams

- Parallel streams have higher overheads than sequential streams
 - Coordinating the threads takes significant time
- Suggestion: use sequential streams by default and only consider parallel ones if:
 - There is a lot of data to process
 - The processing of data takes a long time
 - The process can be performed concurrently
 - There is already a performance problem

(<https://stackoverflow.com/questions/20375176/should-i-always-use-a-parallel-stream-when-possible>)

Getting a parallel stream

- Very easy:
 - Call `parallelStream()` instead of `stream()` when creating the stream
- Call `parallel()` on an existing sequential stream
- All subsequent operations in the pipeline could be processed in parallel

```
Collection<String> strings = ...  
Stream stream = strings.parallelStream();
```

```
Stream parallelStream  
= Arrays.stream(ints).parallel();
```

Reverting to a sequential stream

- Very easy:
 - Call `sequential()` on a parallel stream
 - All subsequent operations in the pipeline will be processed sequentially

Activity



Coding

- Review the example code and watch out for:
 - how the parallel stream is created
 - the time difference between
 - sorting a sequential stream
 - sorting a parallel stream
- If you have any questions, ask

Project: ParallelStreamExample