INB370 / INN370
Software Development
Lecture 2 — Object Orientation and API Documentation

Faculty of Science and Technology

Semester 1, 2010

Aims of the Week 2 lecture and practical session

- To understand how object orientation aids large-scale software development
  - Abstraction
  - Encapsulation, inheritance and polymorphism
  - How these concepts are achieved using Java
- To recognise the importance of good documentation in large-scale software development
  - Documenting Application Programming Interfaces
  - How to use the Javadoc tool to do this

Part A — Object orientation

What is ‘object orientation’?

- **Answer 1:** A programming paradigm that views programs as sets of cooperating *objects*, rather than sequences of *procedures* to follow
- **Answer 2:** A programming paradigm that focuses on the design of *abstract data types*, which define the possible *states* and *operations* on *variables* of that type
- Object-oriented programming was first introduced in the languages *Simula 67* and *SmallTalk*
  - Subsequently popularised through languages such as *C++*, *Java* and *C#*

What's it good for?

- Object orientation is of little or no value for writing small programs or scripts
  - (Which is why many educators think *C#* and *Java* are poor choices for use as introductory programming languages)
- Object orientation's usefulness becomes apparent in the *development* and *maintenance* of *large-scale* programs
  - It makes large-scale development manageable by partitioning the problem and by hiding irrelevant detail
  - It allows programs to evolve without disrupting applications that already depend on them

Abstraction as the basis for managing large-scale programming

- Abstraction is an essential divide-and-conquer concept which occurs in object-oriented programming in various ways:
  - Abstraction through parameterisation
  - Abstraction by specification
  - Procedural abstraction
  - Data abstraction
  - Iteration abstraction
  - Type hierarchies
- Object orientation extends imperative programming with a number of additional abstraction mechanisms:
  - Encapsulation
  - Inheritance
  - Polymorphism

Reference:
B. Liskov and J. Guttag.
Program Development in Java.
Addison-Wesley, 2001.

What is encapsulation?

- **Answer 1:** A design principle that involves hiding detail about the *implementation* of an object from its users
- **Answer 2:** A way of implementing *abstract data types* that hide the data structure's *representation* from its users
- Encapsulation:
  - Makes programs easier to *modify* because the implementation can be changed without affecting the *interface*
  - Makes programs easier to *understand* because unimportant detail is hidden away or abstracted
- Supported in Java by:
  - *Methods*, which can be called without considering how they work
  - *Classes*, which restrict the visibility of fields and methods
  - *Packages*, which restrict the visibility of classes

Applying encapsulation

- To obey the principles of abstraction and encapsulation we should do the following when designing a class:
  - Make fields *private*
  - Make accessors (getters) and mutators (setters) *public*
  - Make helper (utility) methods *private*
- This ensures that the published Application Programming Inerface for the class states *what* the class will do but not *how*

Visibility in Java classes and packages

	`Private`	No Modifier	`Protected`	`Public`
Same class	Yes	Yes	Yes	Yes
Same package subclass	**No**	Yes	Yes	Yes
Same package non-subclass	**No**	Yes	Yes	Yes
Different package subclass	**No**	**No**	Yes	Yes
Different package non-subclass	**No**	**No**	**No**	Yes

What is an object?

- **Answer 1:** An instance of a real-world entity, possessing its own internal state and capabilities
- **Answer 2:** A variable of a particular (abstract) type, with a (usually mutable) value and various operations that may legitimately be applied to it
- Objects are instances of classes
  - Each object has its own *state*, defined by the fields declared in its parent class
  - Each object has its own *behaviour*, defined by the methods (operations) declared in its parent class
  - Each object has its own *identity*, meaning it is distinct from other objects generated from the same class (or values of the same type)

Reference:
K. Lambert and M. Osborne. *Java: A Framework for Programming and Problem Solving*. PWS Publishing, 1999.

Objects in Java

- A new object is generated from a class definition using the **`new`** operator
  - If the class has a *constructor* method with parameters their values must be supplied when the object is created
- Fields and methods within an object are accessed using 'dot notation', i.e., *object*`.`*method*`(`*args*`)`

What is a class?

- **Answer 1:** A template from which new objects are instantiated
- **Answer 2:** A type from which new variables are declared
- Classes are typically characterised by:
  - The fields (attributes) they declare
  - The methods they declare, which may be *constructors*, *accessors*, *mutators* or *finalizers*
  - The visibility of fields and methods (usually *public* or *private*)
  - Whether or not fields and methods are duplicated within each object (*instance* variables) or shared by all objects constructed from the class (*static* variables)

Classes in Java

- Types of variables in Java classes:
  - *fields* — variables declared in a class
  - *local variables* — variables declared within a method
  - *parameters* — variables that are part of a method's signature
- Overloading of method names is allowed, provided their signatures are different
- Constructor method(s) have the same name as the class, and no return type
  - An implicit zero-argument constructor is used if none is declared (but this is not good programming practice)

Classes in Java

- Within a class the keyword **`this`** denotes the current object
  - **`this`**`()` invokes the constructor
  - **`this.`***field* accesses an instance field (variable)
  - **`this.`***method*`()` calls an instance method
- The **`static`** qualifier means only one copy exists of the field or method

Interfaces and abstract classes in Java

- An *interface* is a special kind of class that contains *abstract* methods only
  - All methods within an interface are '**`public`**' by default
  - Abstract methods are *specifications*, consisting of a signature but no body
- Interfaces cannot be instantiated as objects
  - Their abstract methods are incomplete and cannot be executed
- An *abstract class* is a class that defines some member fields and methods but leaves others abstract
  - Abstract methods are indicated by the '**`abstract`**' keyword
- Like interfaces, abstract classes cannot be instantiated
  - They are a partial implementation only

Enum types in Java

- An enum(erated) type is another special kind of class, whose fields are *named constants*
  - Implicitly introduces a '`values`' method which returns an array of values that can be used in 'for each' style **`for`** loops
  - Simple enum types can be introduced with the **`enum`** keyword
- Java allows **`enum`** classes to have other fields and methods, apart from the constants
  - Each constant can have a value (or values) associated with it when it is declared
  - This value is passed to the constructor when an **`enum`** object is created, which allows us to store 'meta' information about the particular constant in a private variable and use it in methods associated with the constant

Pass by value versus pass by reference

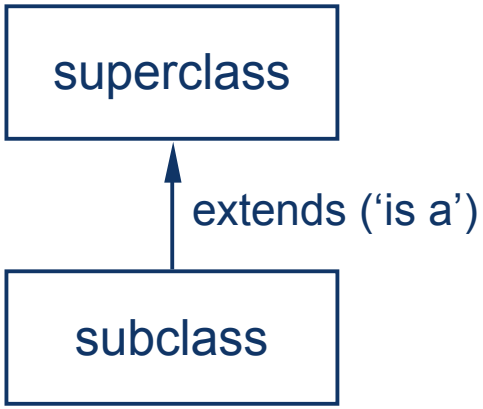
- Primitive types are passed by *value* in Java methods
  - Changes made within a method to a parameter of a primitive type have no effect on the calling program's state
- Objects and arrays are passed by *reference*
  - Changes made to object and array parameters affect the corresponding variables in the calling program

Packages in Java

- Java *packages* are containers for functionally-related classes
- Each package has its own scope and name space
  - Avoids class name collisions
  - (Package name collisions are avoided by naming conventions)
- A **`package`** declaration must occur first in the class's source file for each class in a package
- Packages are imported into classes using the **`import`** keyword
  - Fully-qualified names can be used to disambiguate identically-named members from different packages
- The file folder (directory) structure usually follows the package structure, by convention
  - Package `java.awt.image` is stored in folder `java/awt/image`

What is inheritance?

- **Answer 1:** A way of defining *subclasses* of objects, which are more specialised or complete than those constructed from superclasses, but which share common traits
- **Answer 2:** A way of defining *type hierarchies*, e.g., both integers and reals are numbers, and natural numbers are integers
- In general, objects instantiated from subclasses can do everything that superclass objects can, and sometimes more
  - They may also change (override) some superclass characteristics

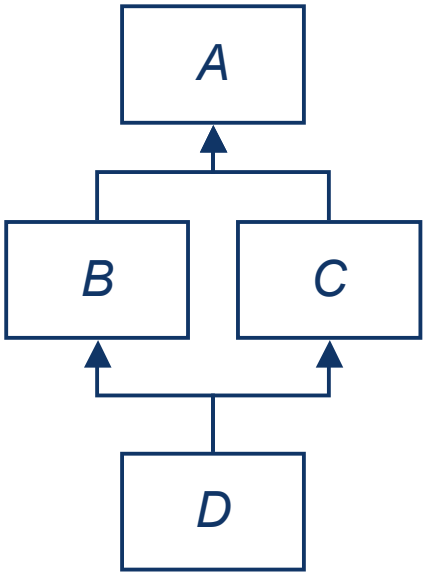


Inheritance in Java

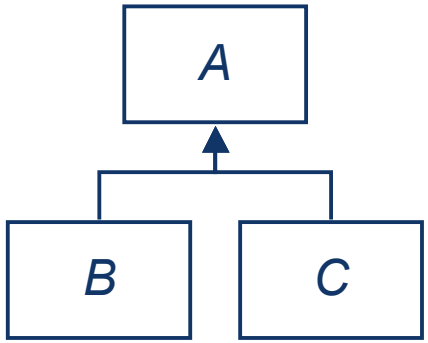
- Inheritance is introduced in Java classes via the **`extends`** keyword, for inheriting from superclasses, and **`implements`** for inheriting from interfaces
- A superclass's constructor is called with **`super`**`()` and its methods with **`super`**`.`*method*`()`
- In Java all classes extend root class `java.lang.Object`

The dilemma of multiple inheritance

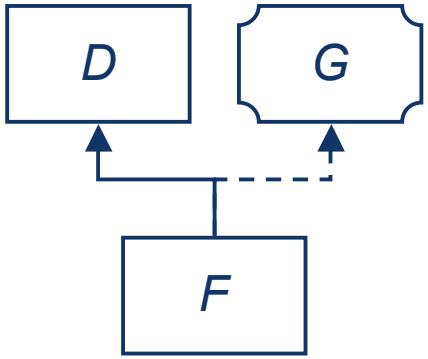
- The diamond import problem:
  1. Classes *B* and *C* inherit fields from class *A*
  2. If a data member is inherited from *A* which instance does class *D* get, *B*'s copy, *C*'s copy or both?



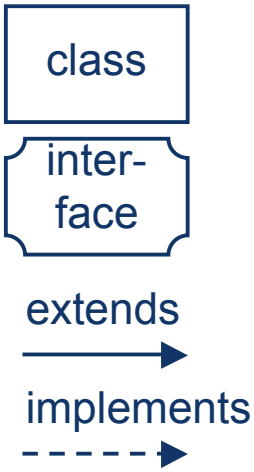
Multiple inheritance in Java

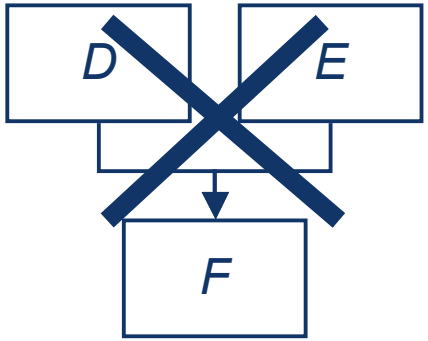



```
class B extends A {...}
class C extends A {...}
```

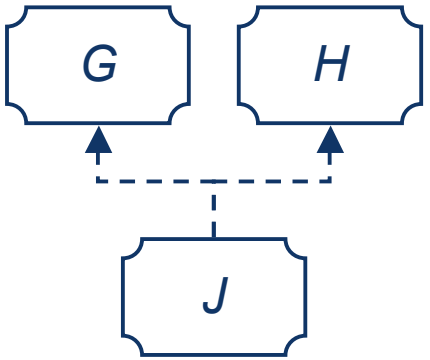



```
class F extends D implements G {...}
```






```
class F extends D, E {...}
```




```
interface J implements G, H {...}
```


Reference:
B. Holmes, *Programming with Java*, Jones and Bartlett Publishers, 1998.

Finality

- The `final` keyword prevents:
  - A class from being extended
  - A method from being overridden
  - A variable's contents from being altered (making it a constant)

What is polymorphism?

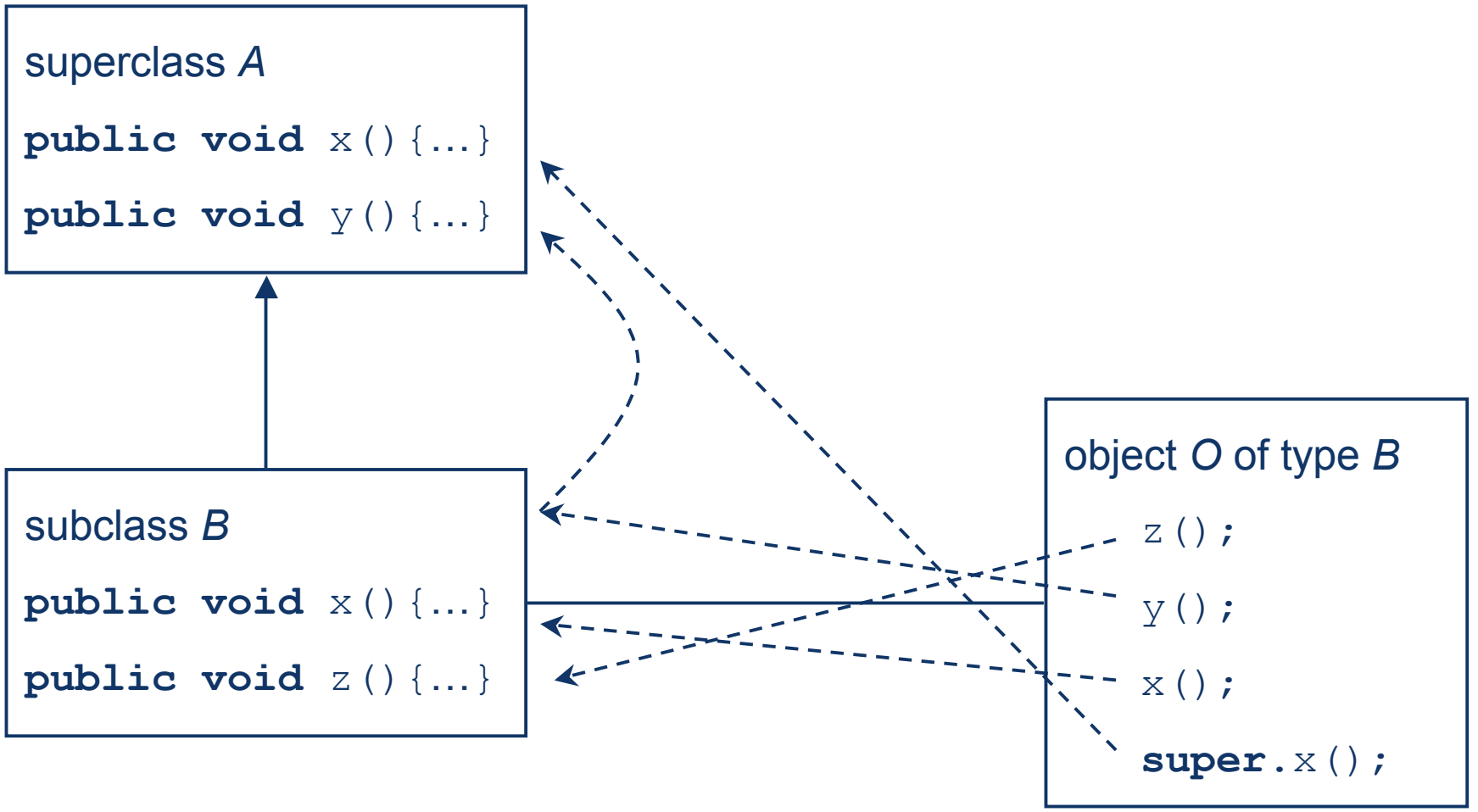
- **Answer 1:** The ability to perform operations (call methods) on objects without needing to know which subclass the object belongs to, provided that we know it extends a superclass which supports that operation
- **Answer 2:** The ability to apply the same operation on values of different types provide they have a common ancestor in the type hierarchy
- Polymorphism makes life easier for programmers by making common operations available in an identical form in otherwise different classes
  - For example, most Java classes provide a `toString` method
  - In practice polymorphism is supported by method *overriding* and *overloading*

Reference:
K. Lambert and M. Osborne. *Java: A Framework for Programming and Problem Solving*. PWS Publishing, 1999.

Method overriding in Java

- When an abstract class or concrete class implements a method and a subclass also has a method with the same signature (method name and parameter list), the subclass method is said to *override* the superclass's method
  - To determine which method to apply the Java Virtual Machine begins at the bottom of the type hierarchy and searches upwards until a match is found
  - Within a subclass an overridden method in a superclass can still be called as `super.`*method*`()`

Method overriding in Java



Method overloading in Java

- Java allows several method declarations to have the same name, provided their parameter lists (and hence signatures) differ
  - This is also allowed for class constructors
- Which specific method is to be called can be determined statically at compile time, by the type of the arguments
  - This is not necessarily true of overriden methods
  - (Some authorities do not consider method overloading to be true 'polymorphism')

Part B — Documentation

The importance of documentation

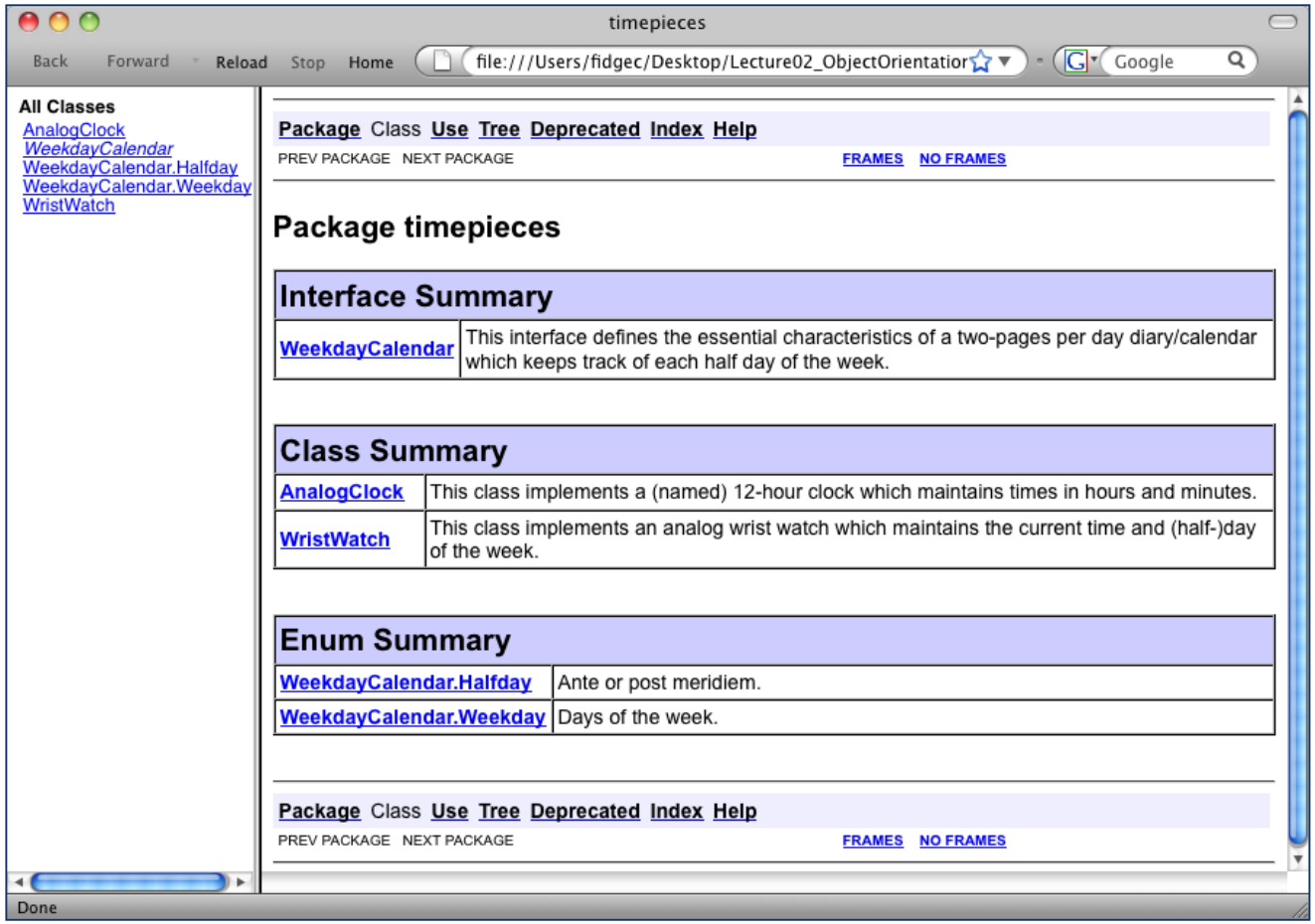
- Documenting your software is an essential task in large-scale software development
  - Comments in the code help the person who has to later modify your code (probably not you!)
  - Application Programming Interface documentation is essential for other programmers who want to use your code
- Keeping documentation current has long been a major problem
  - The pressure to fix bugs in the code is much stronger than the desire to keep documentation up-to-date
  - Generating documentation from code automatically helps solve this problem

Javadoc

- The Java Development Kit includes a `javadoc` command to generate Hyper-Text Markup Language documentation from source code
  - Can be applied to individual classes or whole packages
  - Can be called from within Eclipse
    - (Just as Eclipse calls the JDK's `javac` and `java` commands to compile and run your program, respectively)
- The directory structure of the generated documents follows the hiearchical class structure

Javadoc

- Allows you to publish your own Application Programming Interface



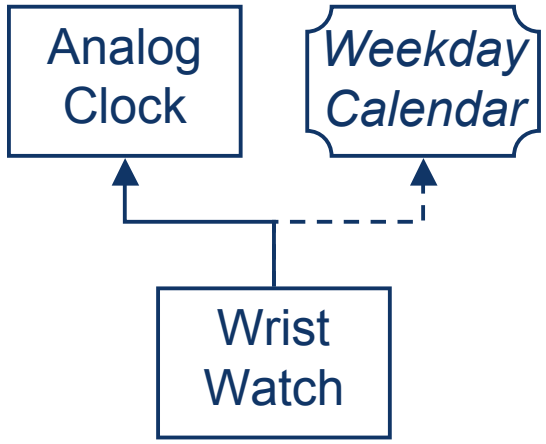
Javadoc

- Javadoc processes *annotations* in special '`/** … */`' comments preceding the declaration of classes and methods:
  - `@author` — the class's author
  - `@version` — the version of the class
  - `@param` — information about a method's parameters (and pre-conditions)
  - `@return` — information about a method's return value (and post-conditions)
  - `@throws` — which exceptions a method may throw
  - `@deprecated` — alerts the reader to the fact that this class or method is outdated (and should indicate which new class or method should be used instead)
  - `@see` — points the reader to another relevant class

Part C — Demonstrations

Demonstrations

1. Developing an `AnalogClock.java` class using Eclipse
2. Generating API documentation for the class using Javadoc
3. Designing a `WeekdayCalendar.java` interface
4. Developing a `WristWatch.java` class that extends and implements the two classes above, respectively
5. Demonstrating the difference between call-by-value and call-by-reference



Homework

- Read Sun's Java Tutorials:
  - Object-oriented programming concepts: `http://java.sun.com/docs/books/tutorial/java/concepts/`
  - Classes and objects: `http://java.sun.com/docs/books/tutorial/java/javaOO/`
  - Interfaces and inheritance: `http://java.sun.com/docs/books/tutorial/java/IandI/`
  - Packages: `http://java.sun.com/docs/books/tutorial/java/package/`

Homework

- Browse the Javadoc reference pages:
  - `http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javadoc.html`
  - `http://java.sun.com/j2se/javadoc/writingdoccomments/`