Topic 1

OO Development Overview
Software Quality Overview

Learning Outcomes

- **Describe** the three broad phases of Software Development: Analysis, Design and Implementation
- **Describe** the basic structure of an OO system and how to initially identify classes, methods, attributes and relationships
- **Explain** the distinction between different types of class relationship: Association, Aggregation, Composition and Inheritance
- **Explain** the purpose of Encapsulation and Information Hiding
- **Describe** the meaning of Software Quality and the attributes/metrics used to measure quality
- **Describe** Coupling and Cohesion and **Explain** the Coupling/Cohesion trade-off
- **Describe** the relationship between modelling using UML and implementation in code

Object-Oriented Development

- In general, ***development*** consists of three broad activities
  - These can be further subdivided but conceptually activities generally fall into one of these broader categories
- **Analysis (e.g. OOA)**
  - understanding and defining the problem
- **Design (e.g. OOD)**
  - A high level conceptual solution expressed using diagrams, pseudocode etc.
- **Programming (e.g. OOP)**
  - A detailed solution implemented in a programming language such as Java, C# or C++

Example: a Bank System

Take the example of a **bank**.

- The **bank** stores data about its **customers**.
- A **customer** has a name, a customer number, a password and one or more accounts.
- An **account** is identified by an account number and has a balance.
- A customer can perform withdrawals, deposits and balance queries on their accounts.

How can we **discover classes** (and their relationships)?

How can we **discover attributes** (variables) **and methods for each class**?

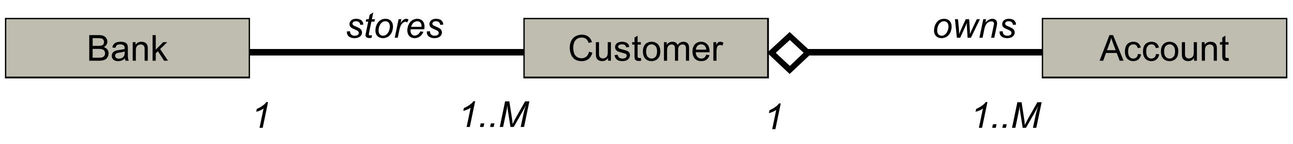
System Structure - Discovering Classes and Relationships

Classes are often either a **concrete entity** (a student, or a course), or an abstract concept or category (a shape, or form of transport.)

A simple starting point (rule of thumb) for discovering classes is to **look for nouns** in the problem specification or significant **concepts** or **artefacts**

Class relationships can take different forms; **association**, **aggregation, composition** and **inheritance (see next slide)**.

When determining cardinality (number of instances in a relationship e.g one to many or many to many) over-estimate rather than underestimate.



System Structure - Discovering Classes and Relationships

Association (or Dependency) – The most generic form of relationship. One class uses another class usually though an attribute level reference (since method parameter or local variable reference implies an even weaker dependency relationship).

Aggregation – Similar to association but only refers to attribute level access (not parameters or local variables). Also implies part/whole semantics but objects on both sides of the relationship have independent lifetimes.

Composition – Same as aggregation but enforces part/whole semantics i.e. if the whole is deleted then so are the parts.

Inheritance – An "is a" or extends relationship implying abstraction or superclass/subclass semantics.

NOTE: In many cases (especially at the design stage) association and inheritance are sufficient.

Non Inheritance Relationships – Rule of Thumb

Assume *A* has a directed (one way) association with *B* i.e. *A* "has a" (or uses) *B*

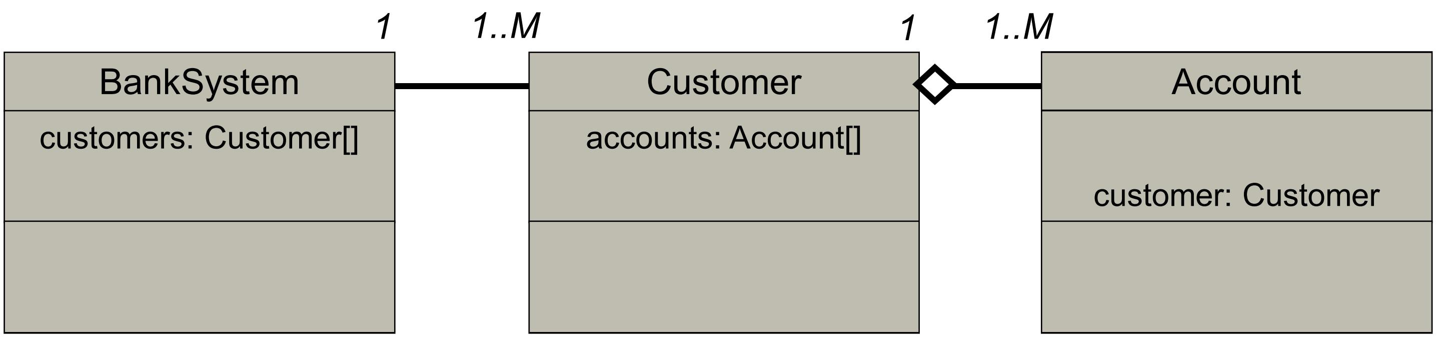
AGGREGATION: *B* requires *A* to operate and in particular *B* is initialized in the constructor of *A*

COMPOSITION: As above but *B* follows the lifecycle of *A* or more specifically it is destroyed in the destructor of *A* (for languages that support this construct) or when *A* is finalised via garbage collection

ASSOCIATION: Neither of the above applies so it is just a general class usage represented as an attribute and directed association

DEPENDENCY: *A* has a parameter or local variable reference to *B*. In practice this distinction from association is not always made at the design stage

Representing Classes and Relationships



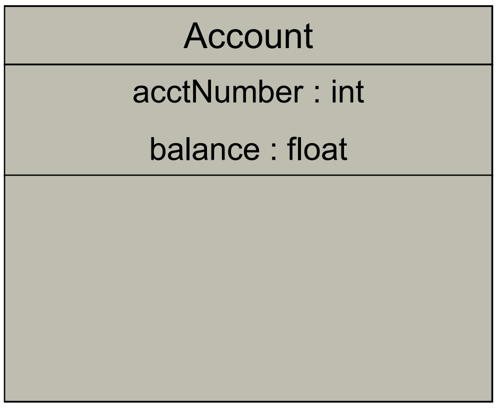
Classes are represented by creating a new class type with the `class` keyword

1..1 relationships are represented by a single attribute.

1..M relationships: the M reference is held in the 1 class by some sort of collection (e.g. an array or Collection [`java.util.Collection`]). M .. M is basically a pair of 1 .. M collections (one each way).

Inheritance is represented using class inheritance in Java (`extends` keyword) and is potentially complex when using polymorphism, interfaces and abstract classes. However it is a powerful and sophisticated technique when done well! (covered further in topic 2).

Discovering Attributes



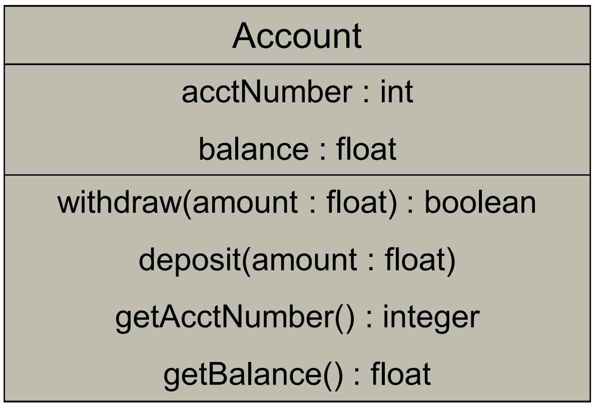
To discover attributes, ask question such as:

"I am an account. What should I know? What do I need to remember? What properties do I have? What <u>states</u> can I be in?"

- my account number?
- my account balance?
- my owner?
- my bank?
- etc.

For each attribute, you need to decide on its name, data type, visibility (public, protected, private or default (package private), also any modifiers i.e. should it be a constant (final) or variable, and have instance or class scope (static).

System Behaviour - Discovering Methods and Relationships



Tip: look for verbs in the problem specification.

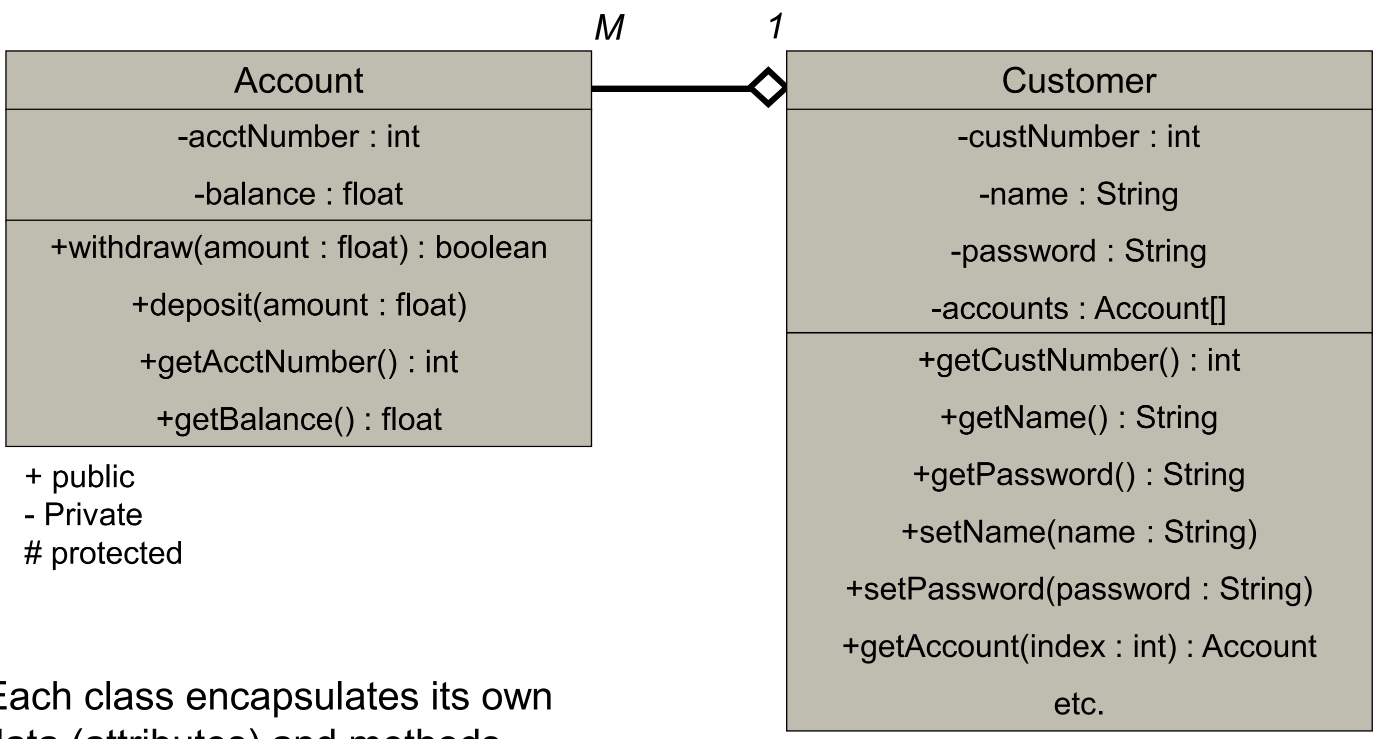
Ask question such as:

"I am an account. What services should I provide?" "What expertise should I have?" "What is my role?" "Who will be using me?" "Who do I interact with?" "What can be done with the attributes?"

- query my account number?
- query my account balance?
- change my account number?
- change my balance?
- deposit or withdraw money?
- etc.

Like attributes, for each method you also need to decide on its name, data types (for parameters and return value), visibility (public, protected, private or default (package)], and whether it should be final (not overridable) or static (called at the class level).

Discovering Methods and Relationships (2)



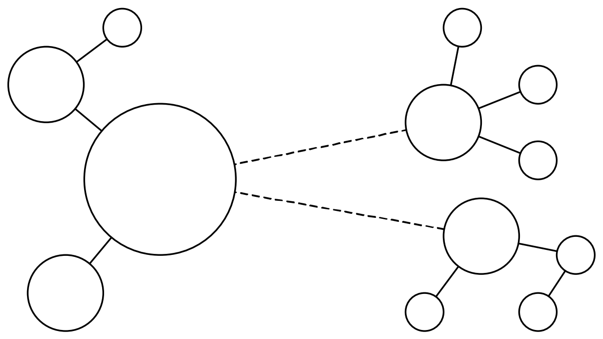
+ public
- Private
# protected

Each class encapsulates its own data (attributes) and methods according to its own specific semantics.

Object-Oriented Design

As we have seen **object-oriented programming** is based on a **number of objects working together** to perform a function.

This kind of approach to developing code can give a number of benefits, such as easier **code maintenance** and enhanced **re-usability**.



a developers' object-oriented program

existing, re-usable software components

However, a **poorly designed** object-oriented program can be **just as, if not more difficult to use and maintain** than if it were developed using procedural or functional approaches.

Encapsulation and Information Hiding

Now that Java's basic object capabilities have been detailed, we can review the fundamentals of **good OO design** with a **focus on implementation**.

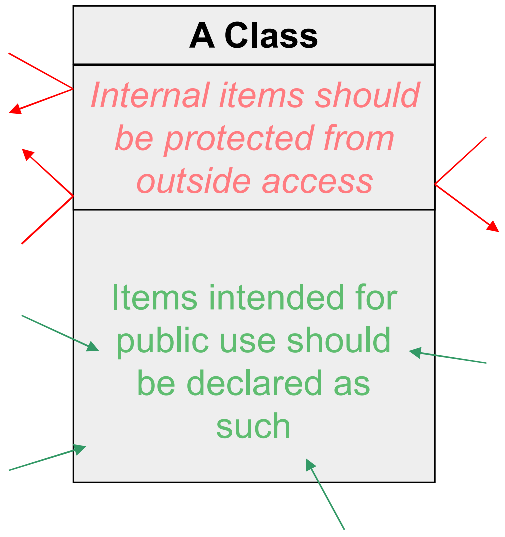
Encapsulation means that both attributes (state) and methods (behaviour) are contained (encapsulated) within a single code module (class).

Information hiding means that only methods of a class have access to private information (usually the attributes (state) of the class but can have private methods as well)

Therefore outside access to the class' internals are on a **"need to know basis"**

TIP: If in doubt restrict visibility (private) and open up later if really necessary.

– programming by **contract (pre/post conditions and invariants)** .. assertions (`assert`) in Java or 3rd party library


A Class
Internal items should be protected from outside access
Items intended for public use should be declared as such

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

From Liang, Introduction to Java Programming, Sixth Edition, (c) 2007 Pearson Education, Inc. All rights reserved. 0-13-222158-6

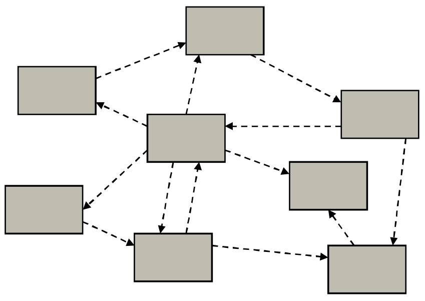
Coupling

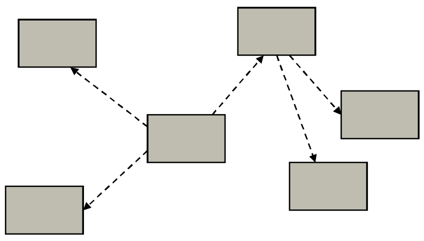
Classes should be properly **encapsulated**. This refers to each class holding all the data/functionality it requires to fulfil its purpose, without being unduly dependent on others.

The dependence of some classes on others in this context is called **coupling**. This reliance should be minimised in your class designs wherever possible.

Excessive coupling between classes **can create problems** when items are modified – with so many dependencies, changing one will imply that many others may also need to be changed.

Coupling can be objectively measured with automated code analysis tools.


excessively high coupling


low coupling

Cohesion

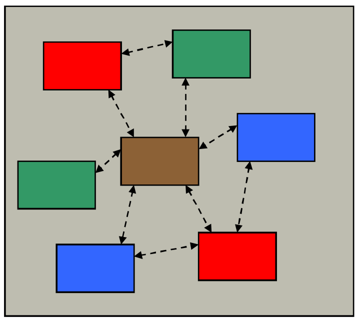
When a class is designed, each of its elements should have a **logical and well-defined role** in the class being able to fulfil its purpose. This **tight relation** of internal components is known as **cohesion**.

A **highly cohesive object** will **not** have items that are **only loosely related** to its role – the object will define **one** concept or perform one task well.

Cohesion is a **good** thing – it reflects a design that is **very clear and pure** to its intended purpose.

Cohesion is harder to measure automatically since it requires semantic understanding.

Coupling and Cohesion is a tradeoff – increasing cohesion (good) can increase coupling (bad).


A highly cohesive object will have elements that are closely inter-related.

Cohesion and Coupling Trade-off

- Cohesion and coupling are a trade-off
  - improving one may have a negative impact on the other
- e.g. Consider a program with only one class
  - it has low coupling (good) since there are no other classes to couple to
  - however cohesion will be low (bad) because the class will be responsible for everything
  - e.g. user interface, database management, unrelated domain logic (accounts, customers etc.)
  - => lots of unrelated methods = not cohesive
- TIP: Write cohesive classes first then try to minimise coupling

Cohesion and Coupling Trade-off (continued)

- Conversely imagine a system with a large number of very small and highly cohesive classes that only have one method
  - One method <u>implies</u> only a single functionality (although the method itself may not be cohesive) so cohesion is high (good).
  - However this system will be highly coupled (bad) because all the small classes have to interact to get work done

‘Good’ Object Oriented Program Design

An object oriented program design should:

- have all internal information well protected from others
- have classes that have a clear and focused purpose (maximise cohesion)
- consist of classes that are highly independent (minimise coupling)
- TIP: a common beginners mistake is to have **too few** classes. IDE’s (e.g. Eclipse) make class management easy.

Adhering to these fundamental design principles should help design object-oriented programs that are **easily understood**, **easily maintainable**, and consist of **highly reusable** components (i.e. quality software, see next slide).

A good program design can **save a lot of time** when it comes to implementation, by eliminating the need for “hacking and patching” code and class models to make them work.

Software Quality (ISO/IEC 9126-1)

Functionality - *A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.*

Suitability	Accuracy
Interoperability	Security
Compliance	

Reliability - *A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.*

Maturity	Fault Tolerance
Recoverability	Compliance

Usability - *A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.*

Understandability	Learnability
Operability	Attractiveness
Compliance	

Software Quality (ISO/IEC 9126-1)

Efficiency - *A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.*

Time Behaviour	Resource Utilisation
Efficiency	

Maintainability* - *A set of attributes that bear on the effort needed to make specified modifications.*

Analyzability	Changeability
Stability	Testability
Compliance	

*** Emphasised in SADI! Most related to Software Structure (Coupling and Cohesion)**

Portability - *A set of attributes that bear on the ability of software to be transferred from one environment to another.*

Adaptability	Installability
Co-Existence	Replaceability
Compliance	

OOD Notation – A Diagrammatic Approach

Class diagrams show the structure of a program at the class level, showing

methods and attributes of classes

relationship between classes,

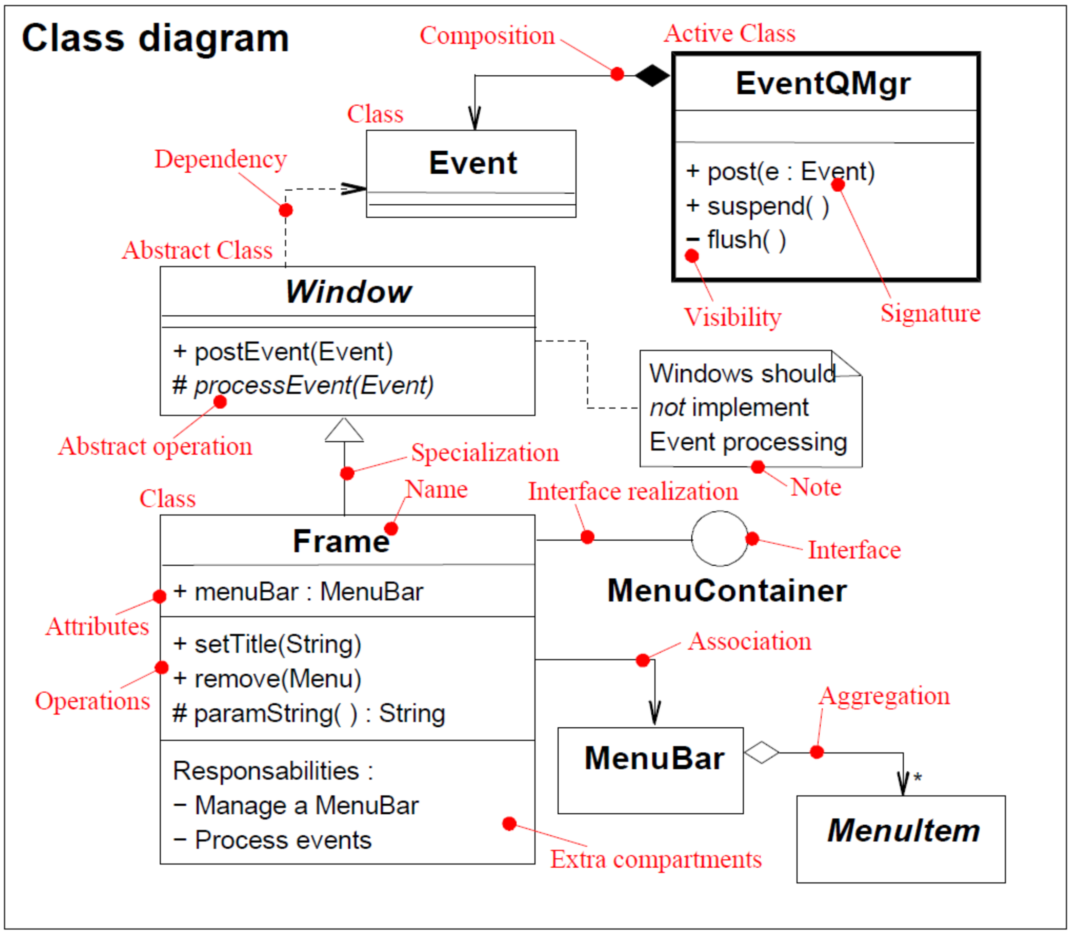
eg inheritance, association

Can range from low detail (classes and relationships only) to high detail (e.g. full method signatures, visibility etc.)

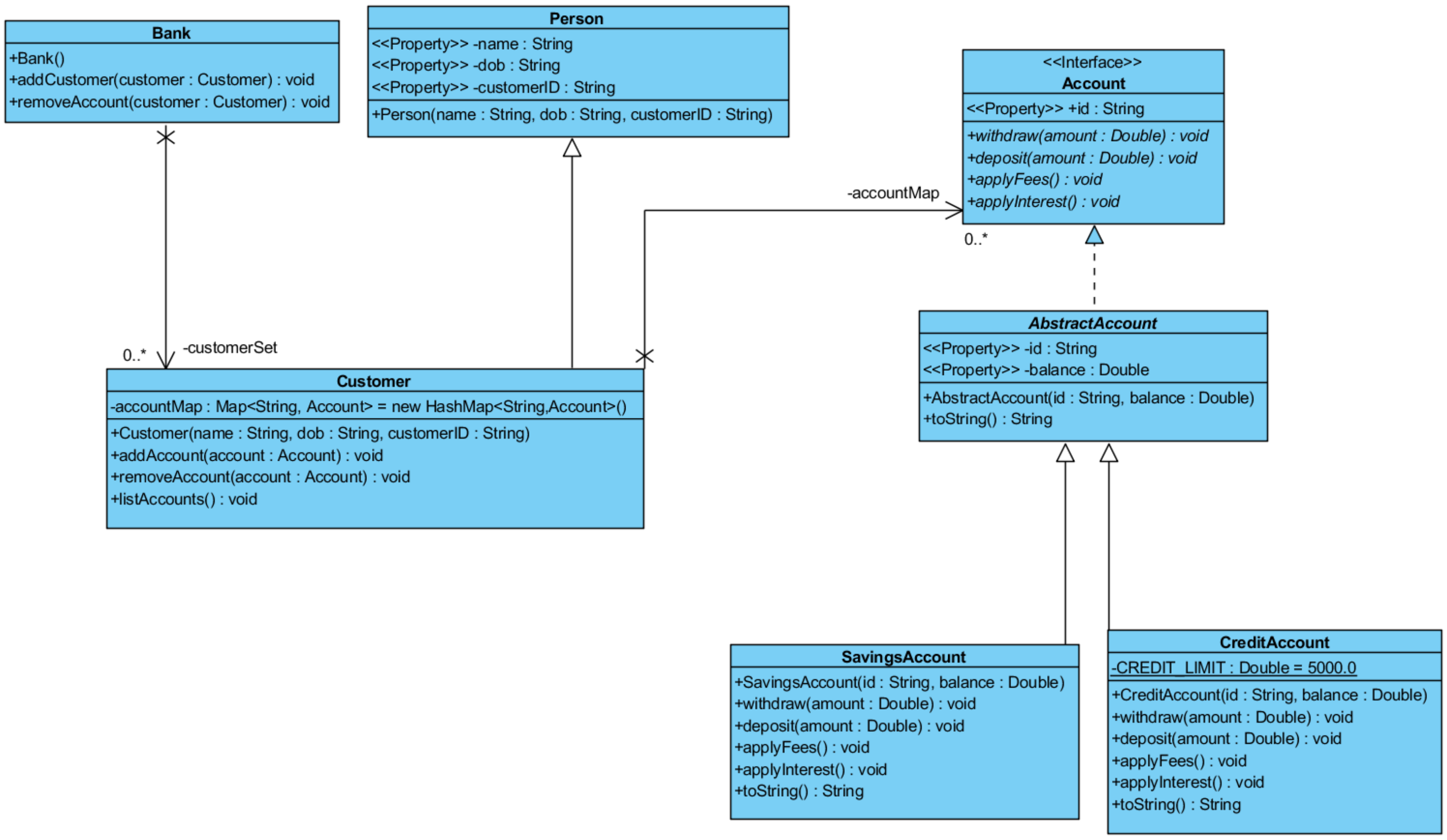
Class diagrams are part of a modelling language (UML).

Class diagrams can be generated from code and skeleton code can be generated from diagrams (e.g.Visual Paradigm tool).

Details on next slide and see additional UML references on Blackboard



Example UML Class Diagram


Bank
+Bank()
+addCustomer(customer : Customer) : void
+removeAccount(customer : Customer) : void
Person
<<Property>> -name : String
<<Property>> -dob : String
<<Property>> -customerID : String
+Person(name : String, dob : String, customerID : String)
<<Interface>>
Account
<<Property>> +id : String
+withdraw(amount : Double) : void
+deposit(amount : Double) : void
+applyFees() : void
+applyInterest() : void
-accountMap
0..*
0..*
-customerSet
Customer
-accountMap : Map<String, Account> = new HashMap<String,Account>()
+Customer(name : String, dob : String, customerID : String)
+addAccount(account : Account) : void
+removeAccount(account : Account) : void
+listAccounts() : void
AbstractAccount
<<Property>> -id : String
<<Property>> -balance : Double
+AbstractAccount(id : String, balance : Double)
+toString() : String
SavingsAccount
+SavingsAccount(id : String, balance : Double)
+withdraw(amount : Double) : void
+deposit(amount : Double) : void
+applyFees() : void
+applyInterest() : void
+toString() : String
CreditAccount
-CREDIT_LIMIT : Double = 5000.0
+CreditAccount(id : String, balance : Double)
+withdraw(amount : Double) : void
+deposit(amount : Double) : void
+applyFees() : void
+applyInterest() : void
+toString() : String

Using an Integrated Development Environment (Eclipse)

Integrated development environments provide automation for a number of common programming tasks

- ➢ Managing packaging and file locations
- ➢ Performing common operations such as creating setters/getters, constructors etc.
- ➢ Refactoring such as renaming and structural changes (extracting interfaces etc.)
- ➢ Real-time Debugging with breakpoints and variable and stack inspection
- ➢ Integration with third party tools (e.g. Visual Paradigm for UML design)
- ➢ Managing run configurations, build path, included libraries, VM parameters etc.

Javadoc Comments

- ➢ Javadoc is an in-built commenting format used to document code in Java
- ➢ It consists of two parts:
  - ➢ in-code annotations within a comment of the form /** ... */
  - ➢ A generator program javadoc.exe which creates html documentation (complete with structure and hyper-linking) from the markup comments in the code
- ➢ The API documentation for Java has been built using this approach
- ➢ Eclipse provides automation
  - ➢ type /** then press enter to get started or use ALT + SHIFT +J
  - ➢ Generate the documentation from eclipse with Project-> Generate Javadoc ...
- ➢ Can generate at class or package granularity
- ➢ The SADI assignment contains javadoc comments on interfaces to help you get started
- ➢ Some of the common annotations are given on the next slide

Javadoc Comments

Tag & Parameter	Usage	Applies to	Since
@author *John Smith*	Describes an author.	Class, Interface, Enum	
@version *version*	Provides software version entry. Max one per Class or Interface.	Class, Interface, Enum	
@since *since-text*	Describes when this functionality has first existed.	Class, Interface, Enum, Field, Method	
@see *reference*	Provides a link to other element of documentation.	Class, Interface, Enum, Field, Method	
@param *name description*	Describes a method parameter.	Method	
@return *description*	Describes the return value.	Method	
@exception *classname description* **@throws** *classname description*	Describes an exception that may be thrown from this method.	Method	
@deprecated *description*	Describes an outdated method.	Method	
{**@inheritDoc**}	Copies the description from the overridden method.	Overriding Method	1.4.0
{**@link** *reference*}	Link to other symbol.	Class, Interface, Enum, Field, Method	
{**@value** *#STATIC_FIELD*}	Return the value of a static field.	Static Field	1.4.0

from http://en.wikipedia.org/wiki/Javadoc .. follow link for more examples and refs