18 CONCEPTS

Terry Marris 3 June 2001

abstract	An abstract class cannot be instantiated. It is usually a super class and has fields and methods that are common to all its subclasses. Only its (concrete) subclasses can be instantiated. But you can have object variables of an abstract class.
	abstract class Account {
	   ...
	}
	
	public class CurrentAccount extends Account { // concrete class
	   ...
	}
	
	Account anAccount; // object variable
	The process of abstraction is to focus on the important and significant points and ignoring or deferring irrelevant detail. A customer may have an account, but we are not concerned with the customer's state of health.

argument	An argument is the value supplied with a message. 100 is the argument in *anAccount.increaseBalance(100);* Argument values are passed to method parameters.
association	An association is the relationship between classes. A person may have a number of accounts. An account is for just one person.
attribute	An attribute is derived from what an object has. An account has an account number and a balance. *accountNumber* and *balance* are attributes of an *Account* class.
behaviour	An objects behaviour is the things it can do. Typical behaviours of an account object include telling you what its account number is and increasing its balance by a given amount. Behaviour is the way an object reacts to a message.
class	A class is a template, blueprint or prototype that defines the variables and methods common to a collection of similar objects (i.e. objects of the same type). For example, your bank account is just one of many bank accounts. An *Account* class would define variables *accountNumber* and *balance*, and methods *getAccountNumber()* and *increaseBalance()* that would be featured in all bank accounts.

collaboration	Collaboration is the interaction between objects and occurs when one object sends a message to another. Objects collaborate to implement a use case. e.g. Use case: "Increase the balance of the given customer by the given amount". main() sends a message to accountBase: *accountBase.add("007", 100));* *accountBase* looks through all its *Account* objects. For each account object asks: is this your account number? *if (anAccount.hasNumber(givenAccountNumber))* if yes, increments balance in the account by the given amount *anAccount.increaseBalance(givenAmount);*
collection	A collection (sometimes called a container) is an object that groups multiple elements into a single unit e.g. *HashSet*. Collections are used to store and retrieve data. Typically, collections represent data items that form a natural group e.g. *AccountBase* - a collection of *Account* objects.

concrete	A concrete class may be instantiated. If *CurrentAccount* is a (concrete) subclass of (abstract) *Account*,
	abstract class Account {
	   ...
	}
	
	public class CurrentAccount extends Account { // concrete class
	   ...
	}
	Account anAccount = new CurrentAccount("Bond", 100);

constructor	A constructor gives a newly created object its initial state. It initialises instance variables.
	Account(String name, int balance)
	{
	  this.name = name;
	  this.balance = balance;
	}

delegation	Delegation is using one class to implement behaviour of an object from another class, without using inheritance. You might send the message *accountBase.increaseBalance("007", 50)*. After finding the appropriate account from its collection of *Account* objects, to completely implement its behaviour, the *increaseBalance()* method would send a message to the *Account* object: *anAccount.increaseBalance(50);*

encapsulation	Encapsulation is the practice of bundling data and operations, and hiding the data so that only chosen parts may be accessed only through given methods. e.g. encapsulate instance variables and methods in a class, protect instance variables by making them private, access instance variables through given methods.
	class Account {
	  private int accountNumber;
	  ...
	}
	  public int getAccountNumber()
	  ...
	}

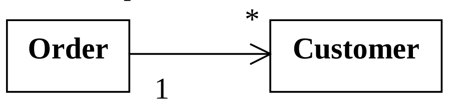
extending	A class extends another by inheriting its instance variables and methods and providing additional functionality.
	class Account {
	  private int accountNumber;
	  private int balance;
	  ...
	  public int getAccountNumber()
	  {
	    return accountNumber;
	  }
	  ...
	}
	class CurrentAccount extends Account { // inherits fields & methods
	                                                            // from Account
	  private int creditLimit; // additional extra field
	  ...
	   String toString() // extending a method
	  {
	    return super.toString() + " Credit Limit: " + creditLimit;
	  }
	}

fields	Attributes are implemented as fields (or instance variables) in Java.
	class Account {
	  private int accountNumber; // a field

generalisation/ specialisation	The further down an inheritance hierarchy a class is, the more specialised its behaviour. The class at the head of an inheritance hierarchy is the most general in its behaviour.
	abstract class Account { // more generalised
	   ...
	}
	
	public class CurrentAccount extends Account { // more specialised
	   ...
	}
	The common features of several classes may be placed in a new class (generalisation). A new class may inherit fields and methods from an existing class and then provide extra functionality (specialisation).

guard	A guard is a boolean expression that determines whether a branch of a selection statement is executed, or whether an iteration is executed.
	if (anAccount.getBalance() < 0) System.out.println("OVERDRAWN");
	while (iter.hasNext()) // loop for as long as iter.hasNext() remains true
	  s += iter.next();

inheritance	Inheritance models the is-a-kind-of relationship. *CurrentAccount*, *SavingsAccount*, *StoreCardAccount*, ..., are all kinds of accounts. *CurrentAccount*, *SavingsAccount*, *StoreCardAccount* are all subclasses of *Account*. *Account* is the superclass to *CurrentAccount*, *SavingsAccount* and *StoreCardAccount*. Each subclass inherits state (in the form of variable declarations) and methods from the superclass. All subclasses of Account have variables *accountNumber* and *balance*, which are defined in *Account*. All subclasses of *Account* have methods *getAccountNumber()* and *increaseBalance()* which are defined in *Account*. Subclasses can also override inherited methods and provide specialised implementations e.g. *toString()*.
instance	An instance is an object of a given class. *Account anAccount = new CurrentAccount("Bond", 100);* *anAccount* is an instance of the *CurrentAccount* class.
instantiation	Instantiation is a process that creates an object. *Account anAccount = new CurrentAccount("Bond", 500);* The name of the class, *Account*, is the name of the class to instantiate. The *new* operator returns a reference to the allocated region of memory where the object is stored. The constructor initialises the new object's instance variables.

interface	An interface defines a protocol (rules) of behaviour. A Java interface is a set of method signatures along with a set of class constants. *interface Account {* *int getAccountNumber();* *void increaseBalance(int amount);* *}* Classes that implement an interface must implement (in some way) each of its methods.
message	A message is a request to an object to perform one of its methods. For example, you might send the message *increaseBalance(100)* to *anAccount* object like this: *anAccount.increaseBalance(100);*
method	A method defines HOW an operation is executed. *public void increaseBalance(int amount)* *{* *balance += amount;* *}*
model	A model is a simplification of reality that focuses on the features that are considered significant or important. An *Account* class might model the essential features of an account (*accountNumber*, *balance*, etc) but leaves out the social interaction between bank customer and bank office staff for example. A collection of classes may model a data processing system.
multiplicity	Multiplicity is an indication of how many objects may participate in a given relationship. A person may have zero, one or many accounts. An account belongs to just one person.
navigability	Navigability is the direction of an association between two objects. It indicates responsibilities.  *Order* has the responsibility for telling you which customer it is for. *Customer* has no corresponding responsibility for telling you what its order are. *Order* will have a *Customer* field in its implementation.

object	An object is an instance of a class. Your bank account could be an instance of an *Account* class. An object has state and behaviour. A *BankAccount* object might have state *number=1473099*, *owner="ReneThom"*, *balance=1003*. A *BankAccount* object might have behaviours that include *telling what its account number is* and *increasing its balance by a given amount.*
object variable	An object variable is a variable whose type is a class or an interface. *Account anAccount;* Such a declaration does not create an object. A call to *new* along with a constructor is required to create the object.
operation	Operations define WHAT a class does. An operation has no implementation. A method implements an operation; an operation specifies HOW an operation performs its task.
overriding	Overriding occurs when a subclass method re-implements (and therefore replaces) its superclass method.

parameter	A parameter is like a local variable; it receives the argument value passed to it in a message. The parameter is *amount* in
	public void increaseBalance(int amount)
	{
	  balance += amount;
	}
	The argument-parameter pair is the mechanism for passing data into a method.

post-condition	The state that is true if all pre-conditions are met.
pre-condition	A statement of what must be true if a method is to perform to its specification.
responsibility	The responsibilities of an object are the obligations it has. A *CurrentAccount* instance will have a responsibility for telling you its account number.
scenario	A scenario is a single path through a use case described by referring to specific objects. "James opens a new current account with opening balance 100." A scenario is one *instance* of a use case.

state	On objects state is the data values it has. *name = bond* *account number = 007* *balance = 100*
subclass	Every instance of a subclass is also an instance of its superclass. Every *CurrentAccount* is also an *Account*. A subclass instance is a kind of superclass instance.
superclass	The parent class from which subclassses inherit attributes and operations.
type	There are two kinds of type: primitive and reference. Primitive types include *int*, *char* and *boolean*. Reference types include *CurrentAccount AccountBase* and *Set*. A type determines the kind of values that can be stored and the operations that can be performed on these values.
UML	Unified Modelling Language - a graphical notation used to express designs.
use case	A use case is a typical interaction between a user an a computer system. *Create a new account* and *increase the balance in an account* are two example use cases. A use case captures a user's goal.
use case scenario	An instance of a use case. "James Bond opens a new current account with initial balance 100".
user	Sometimes known as client - the object that sends messages to a given object.
variable	A variable is a location in memory. It has a type e.g. int, an identifier (name) e.g. number and a value e.g. 7. The variable name refers to the data it contains. The variable's type determines the kind of values that it can hold and the operations that can be performed on it.
visibility	Visibility is the section of code where a class's member methods and variables may be used directly. package: within classes in its own package public: within any class from any package protected: within its own class and its subclasses private: within its own class only

FURTHER READING

www.ibiblio.org/javafaq
www.java.sun.com/docs/books/tutorial
FOWLER & SCOTT *UML Distilled*