Review Lectures

The Final Exam Paper

- Duration: 2 hours and 30 minutes
- Reading: 15 minutes
- Total marks: 65
- Hurdle: 32.5

The Structure

Sections	Questions	Marks
Multiple Choice	25	25
Short Answer	6	15
Long Answer	3	25
Total	34	65

Topics

- Software Engineering (≈5% of 65)

Topics (Cont'd)

- Java and OOP (≈65% of 65)
  - Concepts & Definitions
  - Code fragments
  - Writing a small program

Topics (Cont'd)

- UML and OOD (≈30% of 65)
  - Concepts & Definitions
  - Understanding Diagrams
  - OOD Principles
  - Design Patterns
  - Design using UML

Software Engineering - Introduction

- Software Engineering is an *engineering discipline* which is concerned with *all aspects of software production* from the early stages of system requirements through to maintaining the system after is has gone into use.

Software Process

- Software Process defines the way to produce software. It includes
  - Software life-cycle model
  - Tools to use
  - Individuals building software
- Software life-cycle model defines how different *phases* of the life cycle are managed.

Phases of Software Life-cycle

- Requirements
- Specification (Analysis)
- Design
- Implementation
- Integration
- Maintenance
- Retirement

Life-Cycle Models

- Build-and-fix model
- Waterfall model
- Rapid prototyping model
- Incremental model
- Extreme programming
- Synchronize-and-stabilize model
- Spiral model
- Object-oriented life-cycle models
- Comparison of life-cycle models

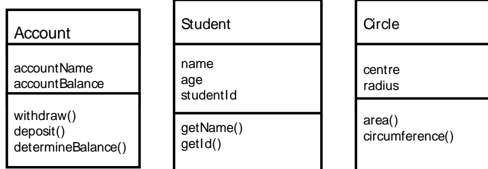
Abstract Data Type (ADT)

- A structure that contains both data and the actions to be performed on that data.
- *Class* is an implementation of an Abstract Data Type.

Object Oriented Design Concepts

Class

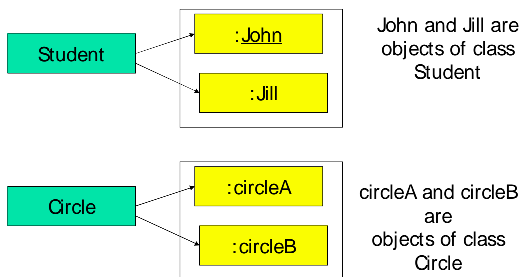
- Class is a set of *attributes* and *operations* that are performed on the attributes.



Objects

- An Object Oriented system is a collection of interacting Objects.
- Object is an instance of a class.

Classes/Objects



Object Oriented Paradigm: Features

- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism
- Persistence
- Delegation

Java Review

Hello World

// HelloWorld.java: Hello World program

```
class HelloWorld
{
    public static void main(String args[])
    {
      System.out.println("Hello World");
    }
}
```

Program Processing

- Compilation

  ```
  # javac HelloWorld.java
  ```
  results in HelloWorld.class

- Execution

  ```
  # java HelloWorld
  ```
  Hello World

Basic Data Types

- Types

  boolean either true of false
char	16 bit Unicode 1.1
byte	8-bit integer (signed)
short	16-bit integer (signed)
int	32-bit integer (signed)
long	64-bit integer (singed)
float	32-bit floating point (IEEE 754-1985)
double	64-bit floating point (IEEE 754-1985)

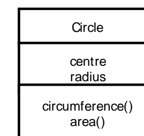
- String (class for manipulating strings)
- Java uses Unicode to represent characters internally

Control Flow

- Control Flow Statements in JAVA
  - while loop
  - for loop
  - do-while loop
  - if-else statement
  - switch statement
- JAVA does not support a *goto* statement

Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.



Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- The basic syntax for a class definition:

```
class ClassName [extends
SuperClassName]
{
        [fields declaration]
        [methods declaration]
}
```

- Bare bone class – no fields, no methods

```
public class Circle {
    // my circle class
}
```


Constructors

- Constructor is a method that gets invoked at object creation time.
- Constructors have the same name as the class.
- Constructors cannot return values.
- Constructors are normally used for initializing objects.
- A class can have more than one constructor – with different input arguments.

Defining a Constructor

- Like any other method

```
public class ClassName {

    // Data Fields…

    // Constructor
    public ClassName()
    {
        // Method Body Statements initialising Data Fields
    }

    //Methods to manipulate data fields
}
```

- Invoking:
  - There is NO explicit invocation statement needed: When the object creation statement is executed, the constructor method will be executed automatically.

Method Overloading

- Constructors all have the same name?
- In Java, methods are distinguished by:
  - name
  - number of arguments
  - type of
  - position of arguments
- Not *method overriding* (coming up), *method overloading:*

Polymorphism

- Allows a single *method or operator* associated with different meaning depending on the type of data passed to it. It can be realised through:
  - Method Overloading
  - Operator Overloading (Supported in C++, but not in Java)
- Defining the same *method* with different argument types (method overloading) - polymorphism.
- The method body can have different logic depending on the date type of arguments.

Scenario

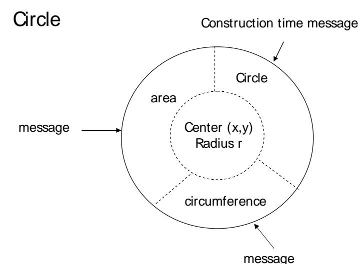
- A Program needs to find a maximum of two numbers or Strings. Write a separate function for each operation.
  - In C:
    - int max_int(int a, int b)
    - int max_string(char *s1, char *s2)
    - max_int (10, 5) or max_string (“melbourne”, “sydney”)
  - In Java:
    - int max(int a, int b)
    - int max(String s1, String s2)
    - max(10, 5) or max(“melbourne”, “sydney”)
  - Which is better ? Readability and intuitive wise ?

Data Hiding and Encapsulation

- Java provides control over the *visibility* of variables and methods, *encapsulation,* safely sealing data within the *capsule* of the class
- Prevents programmers from relying on details of class implementation, so you can update without worry
- Keeps code elegant and clean (easier to maintain)

Visibility

Circle



Parameter passing

- Method parameters which are objects are passed by reference.
- Copy of the reference to the object is passed into method, original value unchanged.

31

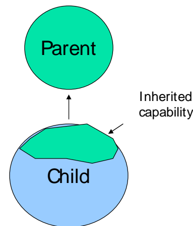
Delegation

- Ability for a class to delegate its responsibilities to another class.
- A way of making an object invoking services of other objects through containership.

32

Inheritance

- Ability to define a class as a subclass of another class.
- Subclass inherits properties from the parent class.



33

Subclassing

- Subclasses created by the keyword *extends*:

```
public class GraphicCircle extends Circle {
      // automatically inherit all the variables and methods
      // of Circle, so only need to put in the 'new stuff'

      Color outline, fill;
      public void draw(DrawWindow dw) {
            dw.drawCircle(x,y,r,outline,fill);
      }
}
```

- Each *GraphicCircle* object is also a *Circle*!

34

Abstract Classes

- An *Abstract* class is a conceptual class.
- An Abstract class cannot be instantiated – objects cannot be created.
- Abstract classes provides a common root for a group of classes, nicely tied together in a package:

35

Abstract Classes

```
package shapes;

public abstract class Shape {
      public abstract double area();
      public abstract double circumference();
      public void move() {
          // impementation
      }
}
```

36

Abstract Classes

```
public Circle extends Shape {
    protected double r;
    protected static final double PI =3.1415926535;
    public Circle() { r = 1.0; )
    public double area() { return PI * r * r; }
…
}
public Rectangle extend Shape {
    protected double w, h;
    public Rectangle() { w = 0.0; h=0.0; }
    public double area() { return w * h; }
}
```


Abstract Classes

- Any class with an abstract method is automatically abstract
- A class declared abstract, even with no abstract methods can not be instantiated
- A subclass of an abstract class can be instantiated if it overrides each of the abstract methods, with an implementation for each
- A subclass that does not implement all of the superclass abstract methods is itself abstract

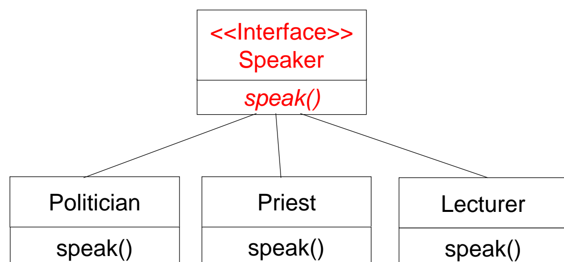
Interfaces

- *Interface* is a conceptual entity similar to a Abstract class.
- Can contain only constants (final variables) and abstract method (no implementation) - Different from Abstract classes.
- Use when a number of classes share a common interface.
- Each class should implement the interface.

Interfaces: An informal way of realising multiple inheritance

- An interface is basically a kind of class—it contains methods and variables, but they have to be only abstract classes and final fields/variables.
- Therefore, it is the responsibility of the class that implements an interface to supply the code for methods.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realising multiple inheritance in Java.

Interface - Example



Interfaces Definition

- Syntax (appears like abstract class):

```
interface InterfaceName {
    // Constant/Final Variable Declaration
    // Methods Declaration – only method body
}
```

- Example:

```
interface Speaker {
    public void speak( );
}
```

Error Handling

- Any program can find itself in unusual circumstances – Error Conditions.
- A “good” program should be able to handle these conditions gracefully.
- Java provides a mechanism to handle these error condition - *exceptions*

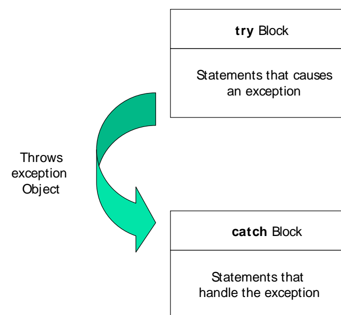
Exceptions in Java

- A method can signal an error condition by throwing an exception – *throws*
- The calling method can transfer control to a exception handler by catching an exception - *try, catch*
- Clean up can be done by - *finally*

Common Java Exceptions

- ArithmeticException
- ArrayIndexOutOfBoundException
- ArrayStoreException
- FileNotFoundException
- IOException – general I/O failure
- NullPointerException – referencing a null object
- OutOfMemoryException
- SecurityException – when applet tries to perform an action not allowed by the browser’s security setting.
- StackOverflowException
- StringIndexOutOfBoundException

Exception Handling Mechanism

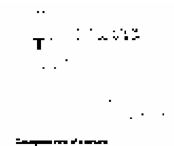


Syntax of Exception Handling Code

```
…
…
try {
    // statements
}
catch( Exception-Type e)
{
    // statements to process exception
}
..
..
```

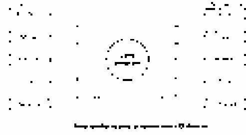

Streams

- Java Uses the concept of Streams to represent the ordered sequence of data, a common characteristic shared by all I/O devices.
- Streams presents a uniform, easy to use, object oriented interface between the program and I/O devices.
- A stream in Java is a path along which data flows (like a river or pipe along which water flows).



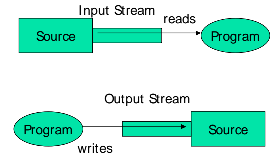
I/O and Data Movement

- The flow of data into a program (input) may come from different devices such as keyboard, mouse, memory, disk, network, or another program.
- The flow of data out of a program (output) may go to the screen, printer, memory, disk, network, another program.
- Both input and output share a certain common property such as unidirectional movement of data – a sequence of bytes and characters and support to the sequential access to the data.



Stream Types

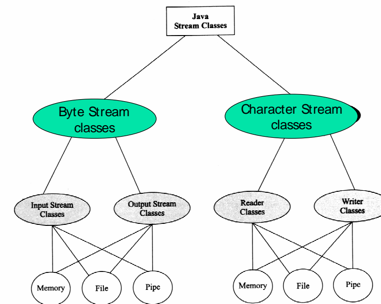
- The concepts of sending data from one stream to another (like a pipe feeding into another pipe) has made streams powerful tool for file processing.
- Connecting streams can also act as filters.
- Streams are classified into two basic types:
  - Input Steam
  - Output Stream



Java Stream Classes

- Input/Output related classes are defined in java.io package.
- Input/Output in Java is defined in terms of streams.
- A *stream* is a sequence of data, of no particular length.
- Java classes can be categorised into two groups based on the data type one which they operate:
  - *Byte streams*
  - *Character Streams*

Classification of Java Stream Classes



Classification of Java stream classes

Graphical User Interface (GUI) Applications

Abstract Windowing Toolkit (AWT)

Events Handling

Applets

AWT - Abstract Windowing Toolkit

- Single Windowing Interface on Multiple Platforms
- Supports functions common to all window systems
- Uses Underlying Native Window system
- AWT provides
  - GUI widgets
  - Event Handling
  - Containers for widgets
  - Layout managers
  - Graphic operations

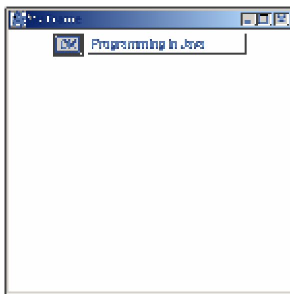
Building Graphical User Interfaces

- import java.awt.*;
- Assemble the GUI
  - use GUI components,
    - basic components (e.g., Button, TextField)
    - containers (Frame, Panel)
  - set the positioning of the components
    - use Layout Managers
- Attach events

A sample GUI program

```
import java.awt.*;
public class MyGui
{
    public static void main(String args[] )
    {
        Frame f = new  Frame ("My Frame");
        Button b = new Button("OK");
        TextField tf = new TextField("Programming in Java", 20);
        f.setLayout(new FlowLayout());
        f.add(b);
        f.add(tf);
        f.setSize(300, 300);
        f.setVisible(true);
    }
}
```

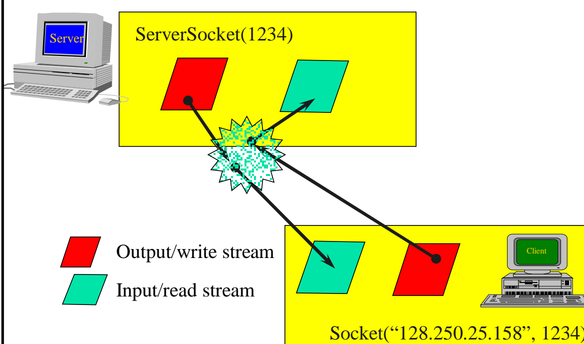

Output



Sockets and Java Socket Classes

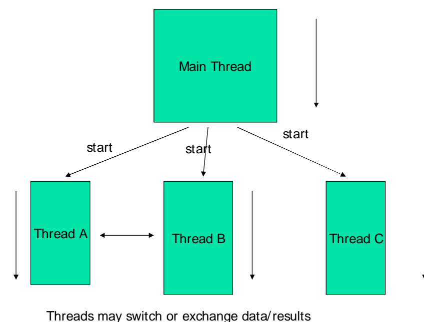
- A socket is an endpoint of a two-way communication link between two programs running on the network.
- A socket is bound to a port number so that the TCP layer can identify the application that data destined to be sent.
- Java's .net package provides two classes:
  - Socket – for implementing a client
  - ServerSocket – for implementing a server

Java Sockets



It can be host_name like "mandroo.cs.mu.oz.au"

A Multithreaded Program



Threads may switch or exchange data/results

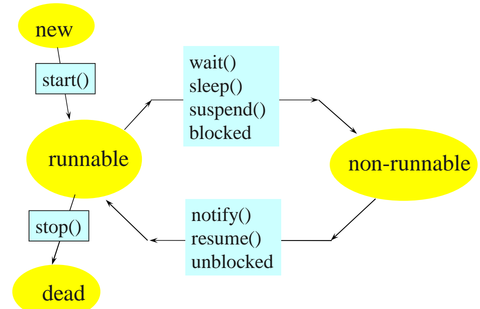
An example

```
class MyThread extends Thread {     // the thread
        public void run() {
                System.out.println(" this thread is running ... ");
        }
} // end class MyThread

class ThreadEx1 {                   // a program that utilizes the thread
        public static void main(String [] args ) {
            MyThread t = new MyThread();
            // due to extending the Thread class (above)
            // I can call start(), and this will call
            // run(). start() is a method in class Thread.
            t.start();
        } // end main()
}       // end class ThreadEx1
```

61

Life Cycle of Thread



62

Unified Modeling Language

63

Software Development Process and Unified Modeling Language (UML)

- A software development process is a set of phases that are followed to bring a product or a system from conception to delivery.
- In the Unified Process, there are four of these phases:
  - Inception (*Analysis phase*)
    - identify the system we are going to develop, including what it contains and its business case.
    - UML: use-case diagrams
  - Elaboration (*Design phase*):
    - perform detailed design and identify the foundation of system from "use case diagram", which eventually lead to classes.
    - UML: classes, objects, class diagrams, sequence diagram, collaboration diagrams etc.
  - Construction (*Implementation phase*): write software using Java/C++
    - the actual building of the product from the design of the system.
  - Transition (*Rolling out phase*): Deliver the system/product to the users. Includes maintenance, upgrades, and so on until phasing out.

64

UML™ – Diagrams – cont..

Structural	Behavioral
Class Diagram	Use case Diagram
Object Diagram	Sequence Diagram
Component Diagram	Collaboration Diagram
Deployment Diagram	Statechart Diagram
	Activity Diagram

65

Use Case Diagrams

- Use Case diagrams show the various activities the users can perform on the system.
  - System is something that performs a function.
- They model the dynamic aspects of the system.
- Provides a *user's* perspective of the system.

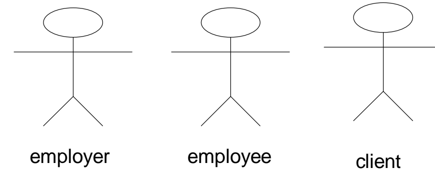
66

Use Case Diagrams

- A set of *ACTORS*: roles users can play in interacting with the system.
  - An actor is used to represent something that users our system.
- A set of *USE CASES*: each describes a possible kind of interaction between an actor and the system.
  - Uses cases are actions that a user takes on a system
- A number of *RELATIONSHIPS* between these entities (Actors and Use Cases).
  - Relationships are simply illustrated with a line connecting actors to use cases.

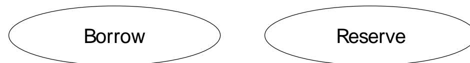
Use Case Diagrams - Actors

- An *actor* is a user of the system playing a particular role.
- Actor is shown with a stick figure.

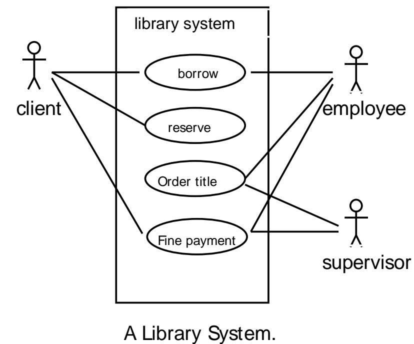


Use Case Diagrams – Use Cases

- Use case is a particular activity a user can do on the system.
- Is represented by an ellipse.
- Following are two use cases for a library system.



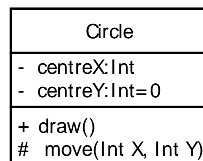
Use Case Diagram – Example1 (Library)



A Library System.

Class Visibility

- public level +
- protected level #
- private level -



Class Relationships

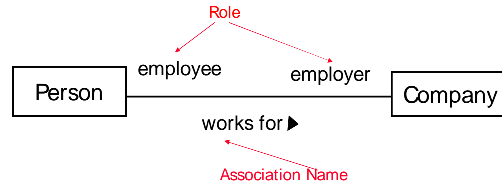
- Classes can related to each other through different relationships:
  - Association (delegation)
  - Generalization (inheritance)
  - Realization (interfaces)
  - Dependency

Association

- Association describes a link, a link being a connection among objects between classes.
- Association is shown by a solid line between classes.

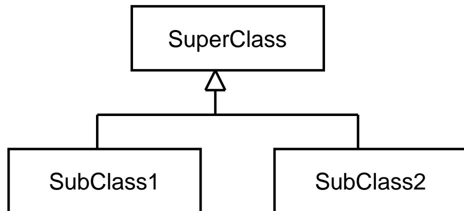
Association - Example

- A Person works for a Company.

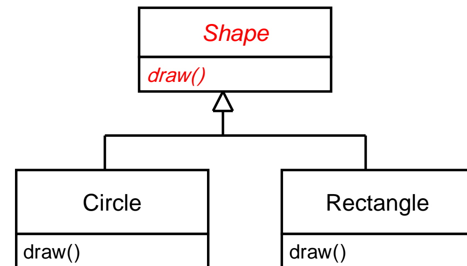


Generalization (Inheritance)

- Child class is a special case of the parent class

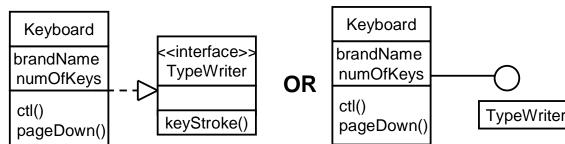


Abstract Methods (Operations)

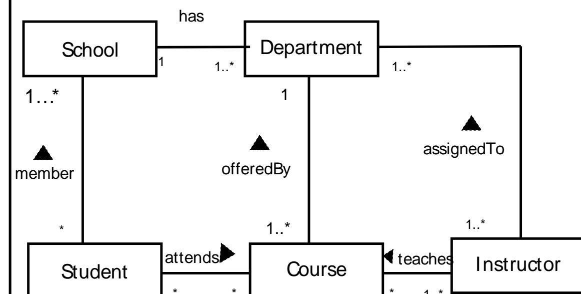


Realization- Interface

- Interface is a set of operation the class carries out



Class Diagram Example

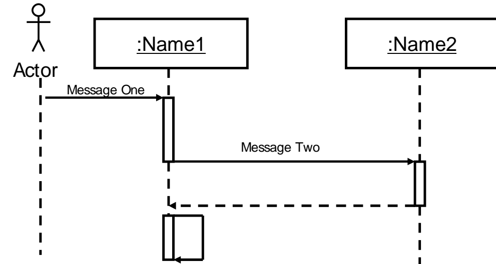


Sequence Diagram

- Shows how objects communicate with each other over time.
- The sequence diagram consists of *OBJECTS*, *MESSAGES* represented as solid-line arrows, and *TIME* represented as a vertical progression

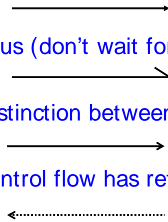
Sequence Diagram – Time & Messages

- Messages are used to illustrate communication between different active objects of a sequence diagram.

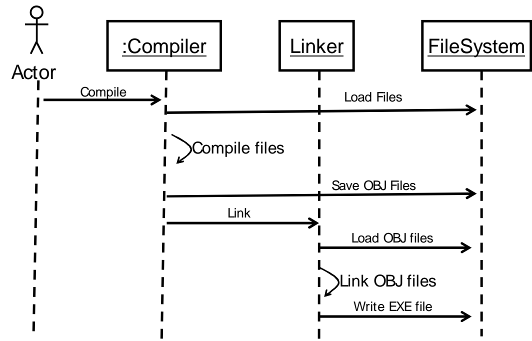


Types of Messages

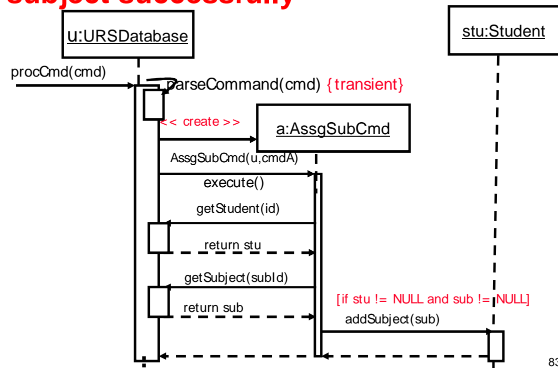
- Synchronous (flow interrupt until the message has completed.
- Asynchronous (don't wait for response)
- Flat – no distinction between sysn/async
- Return – control flow has returned to the caller.



Sequence Diagram – Compilation



Sequence Diagram – Enroll Student for subject successfully



Collaboration Diagram – Enroll Student in Subject Scenario