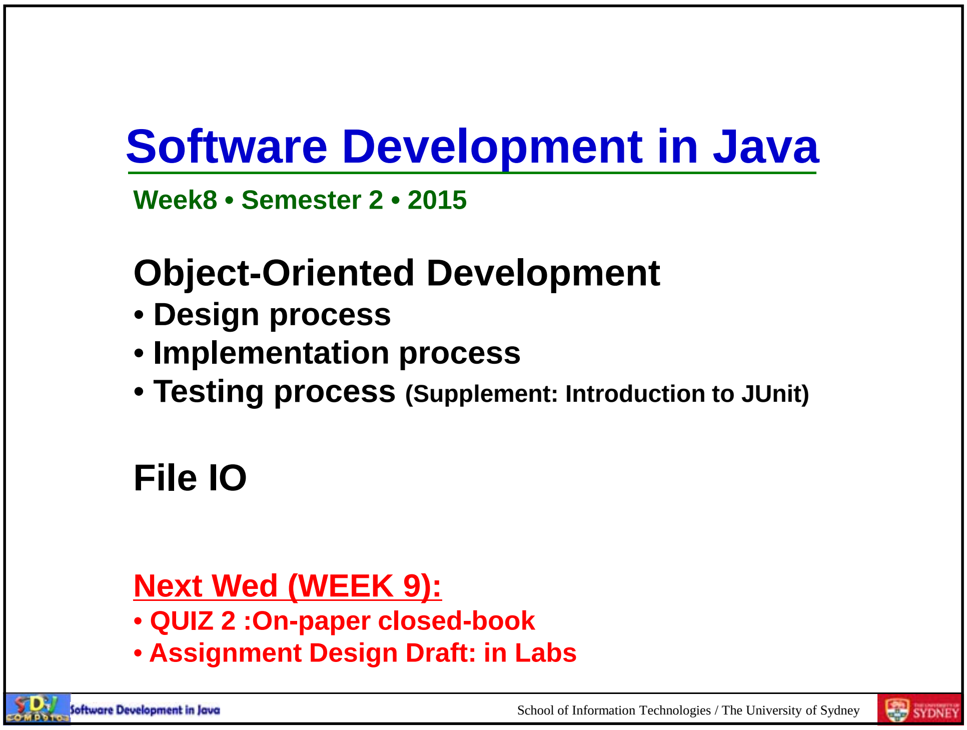


Software Development in Java

Week8 • Semester 2 • 2015

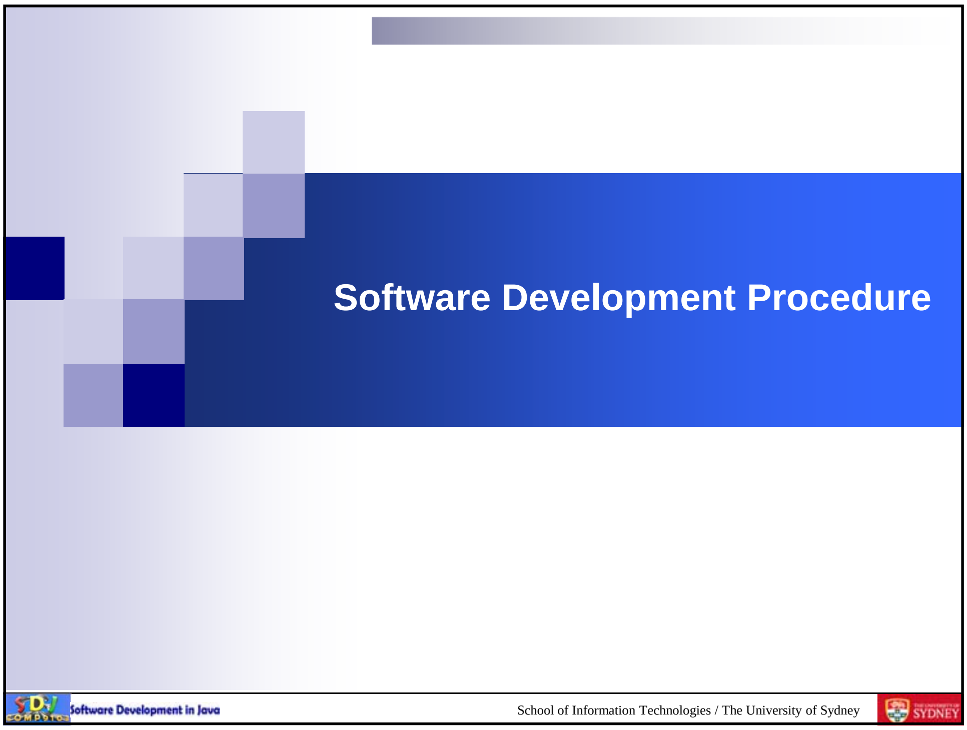
Object-Oriented Development

- Design process
- Implementation process
- Testing process (Supplement: Introduction to JUnit)

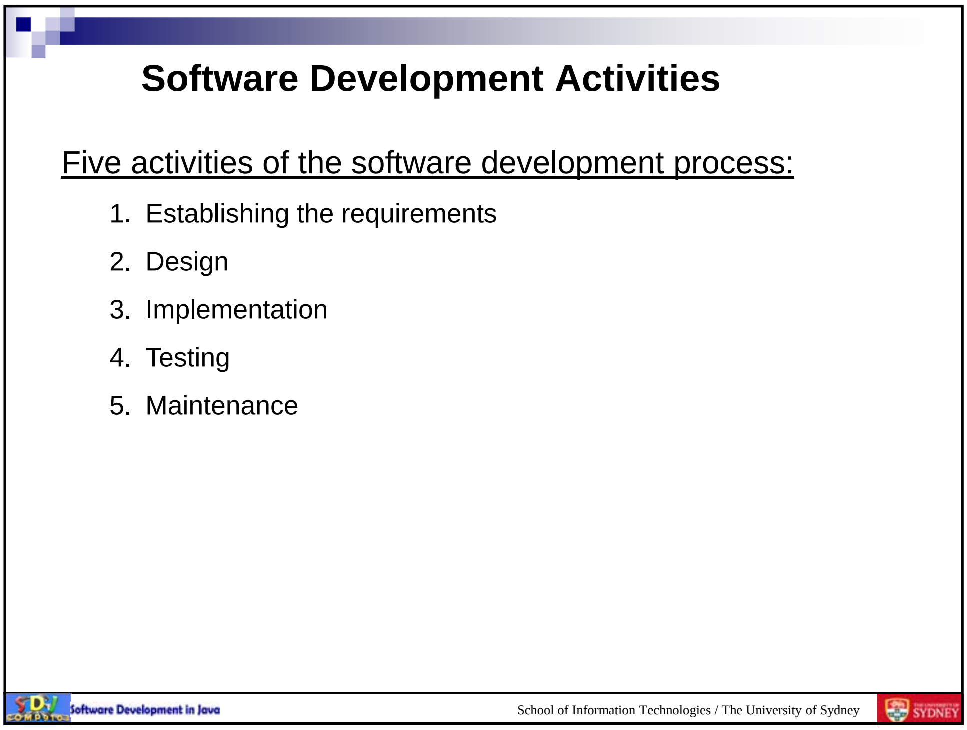
File IO

Next Wed (WEEK 9):

- QUIZ 2 :On-paper closed-book
- Assignment Design Draft: in Labs



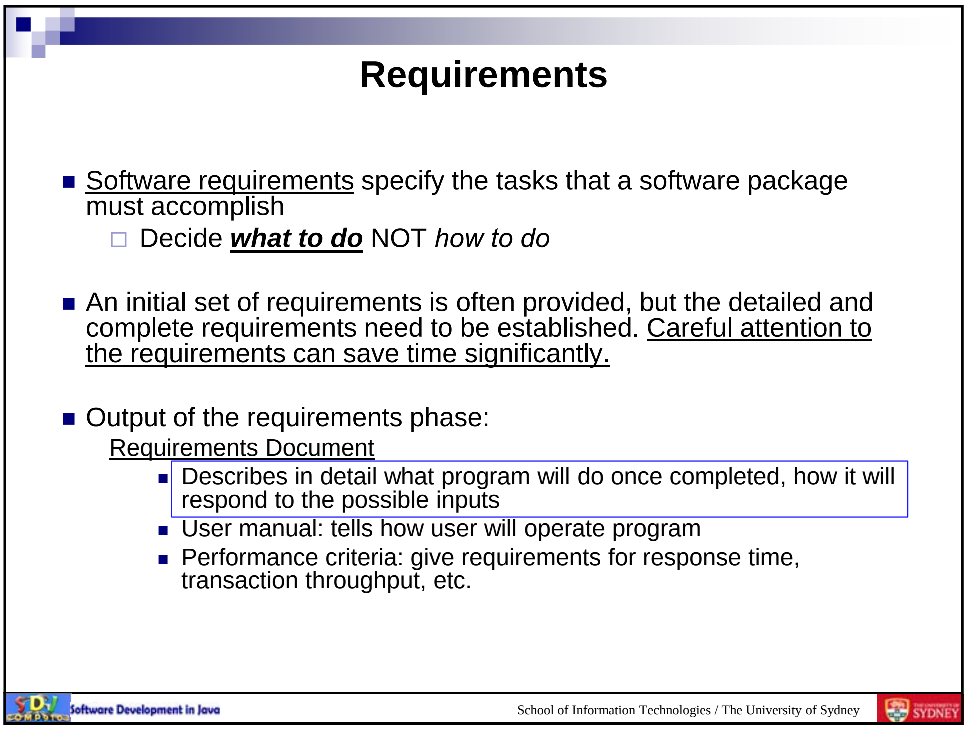
Software Development Procedure



Software Development Activities

Five activities of the software development process:

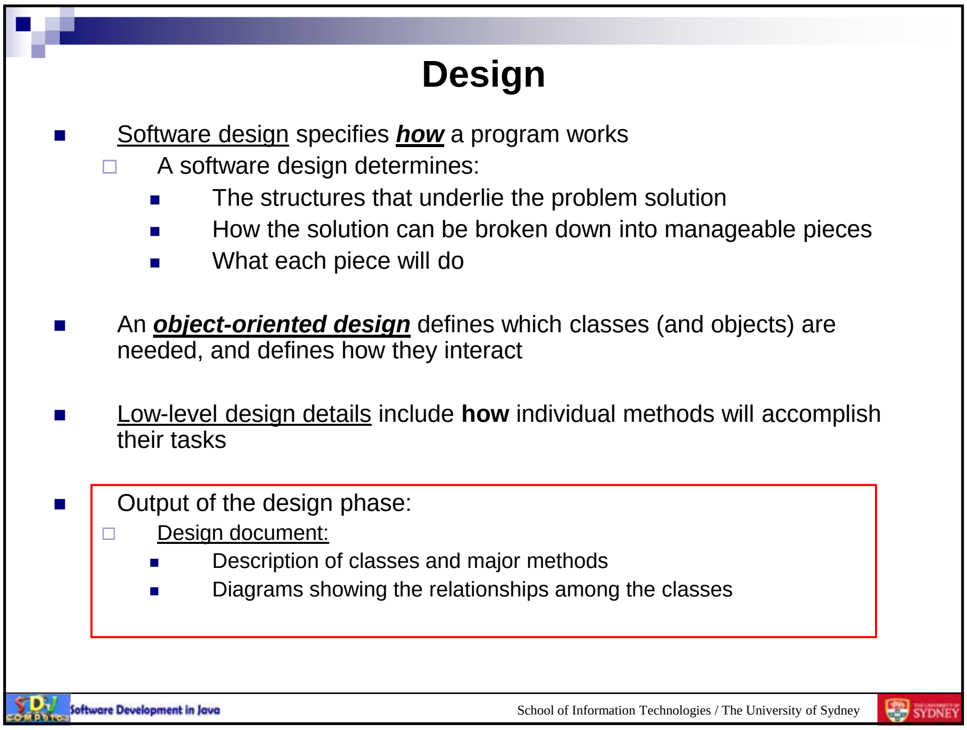
1. Establishing the requirements
2. Design
3. Implementation
4. Testing
5. Maintenance



Requirements

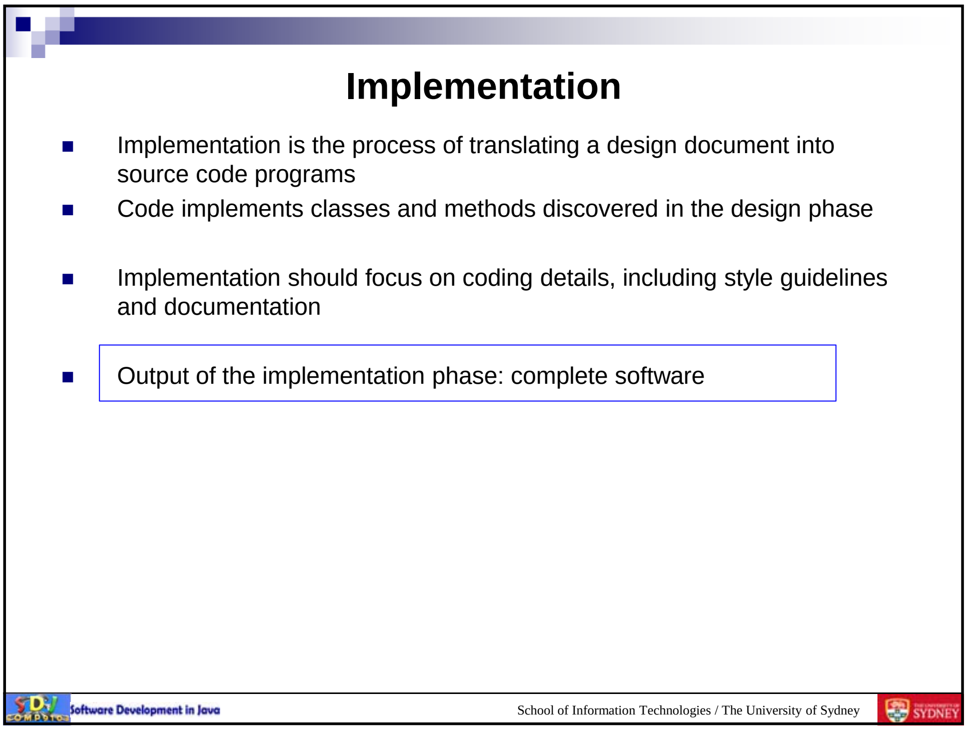
- Software requirements specify the tasks that a software package must accomplish
  - □ Decide ***what to do*** NOT *how to do*
- An initial set of requirements is often provided, but the detailed and complete requirements need to be established. Careful attention to the requirements can save time significantly.
- Output of the requirements phase:

  Requirements Document
  - Describes in detail what program will do once completed, how it will respond to the possible inputs
  - User manual: tells how user will operate program
  - Performance criteria: give requirements for response time, transaction throughput, etc.



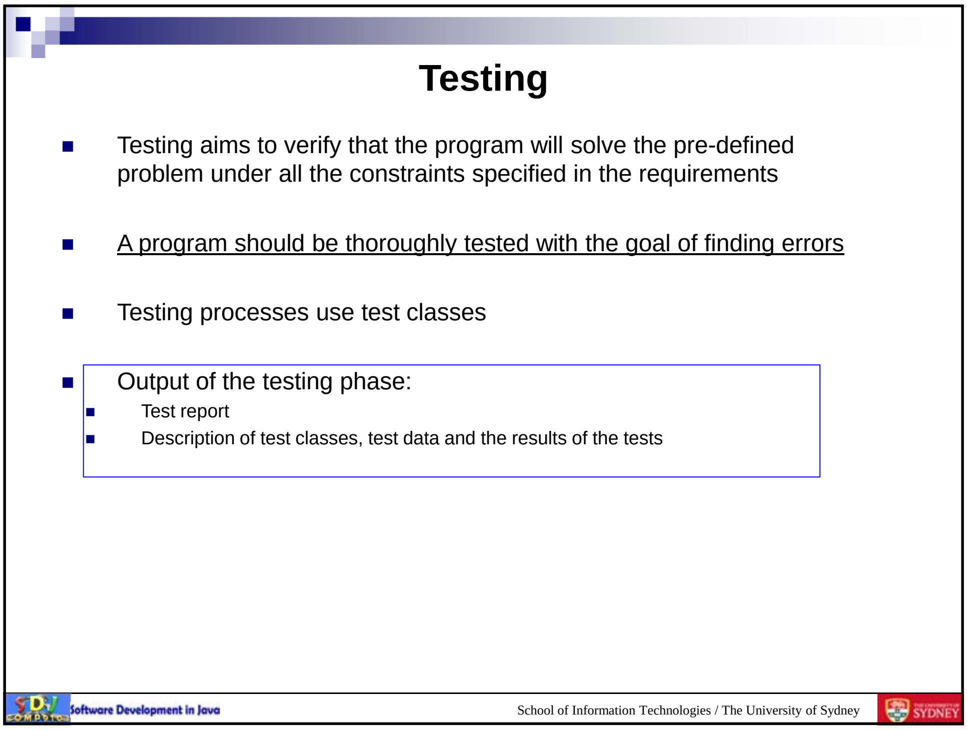
Design

- Software design specifies ***how*** a program works
  - □ A software design determines:
    - The structures that underlie the problem solution
    - How the solution can be broken down into manageable pieces
    - What each piece will do
- An ***object-oriented design*** defines which classes (and objects) are needed, and defines how they interact
- Low-level design details include **how** individual methods will accomplish their tasks
- Output of the design phase:
  - □ Design document:
    - Description of classes and major methods
    - Diagrams showing the relationships among the classes



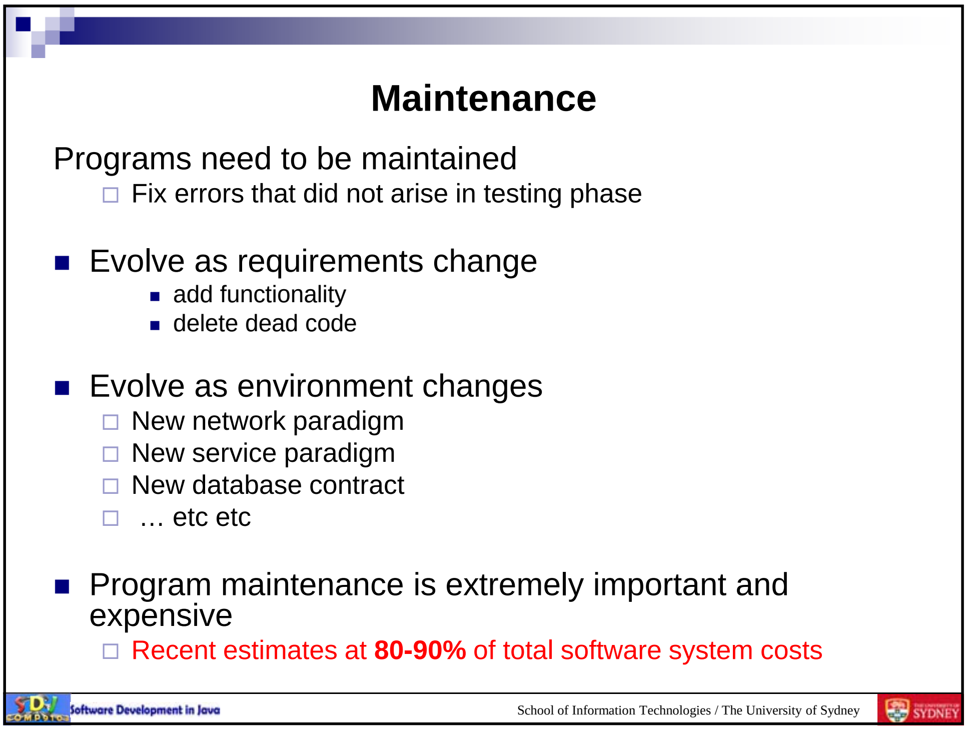
Implementation

- Implementation is the process of translating a design document into source code programs
- Code implements classes and methods discovered in the design phase
- Implementation should focus on coding details, including style guidelines and documentation
- Output of the implementation phase: complete software



Testing

- Testing aims to verify that the program will solve the pre-defined problem under all the constraints specified in the requirements
- A program should be thoroughly tested with the goal of finding errors
- Testing processes use test classes
- Output of the testing phase:
  - Test report
  - Description of test classes, test data and the results of the tests



Maintenance

Programs need to be maintained

- □ Fix errors that did not arise in testing phase

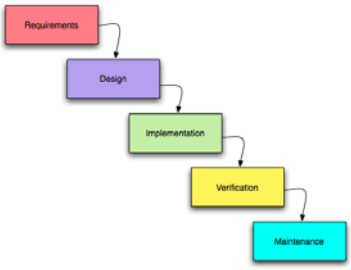
- Evolve as requirements change
  - add functionality
  - delete dead code
- Evolve as environment changes
  - □ New network paradigm
  - □ New service paradigm
  - □ New database contract
  - □ … etc etc
- Program maintenance is extremely important and expensive
  - □ Recent estimates at **80-90%** of total software system costs

Software Process Models

Waterfall model

- In Waterfall model, each software development activity (from requirements to maintenance) will be conducted one after the other sequentially and no revision on the previous step will be included.

- The waterfall model is generally simplistic and unrealistic, but it serves as a reference point and defines important concepts.



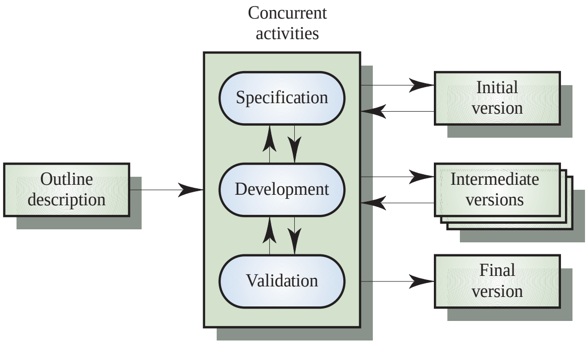
The unmodified "waterfall model". Progress flows from the top to the bottom, like a waterfall. (wikipedia)

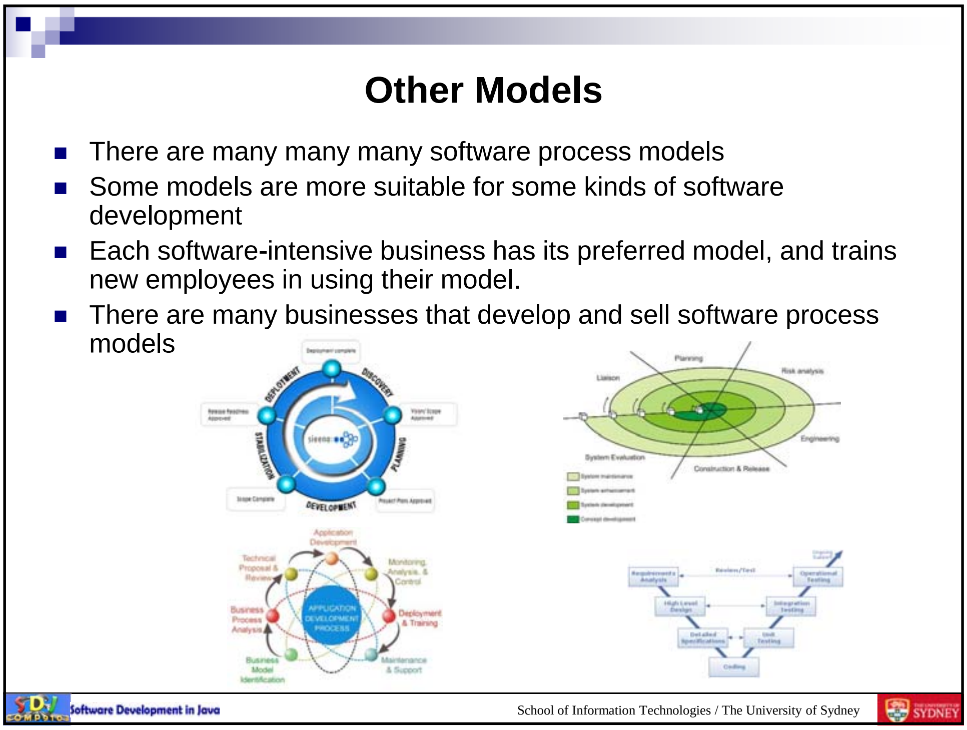
Evolutionary Software Development Model

▪Basic idea:

▪Build prototype version of system, seek user comments, and keep refining until accomplished

▪Other side of formality spectrum from Waterfall Model





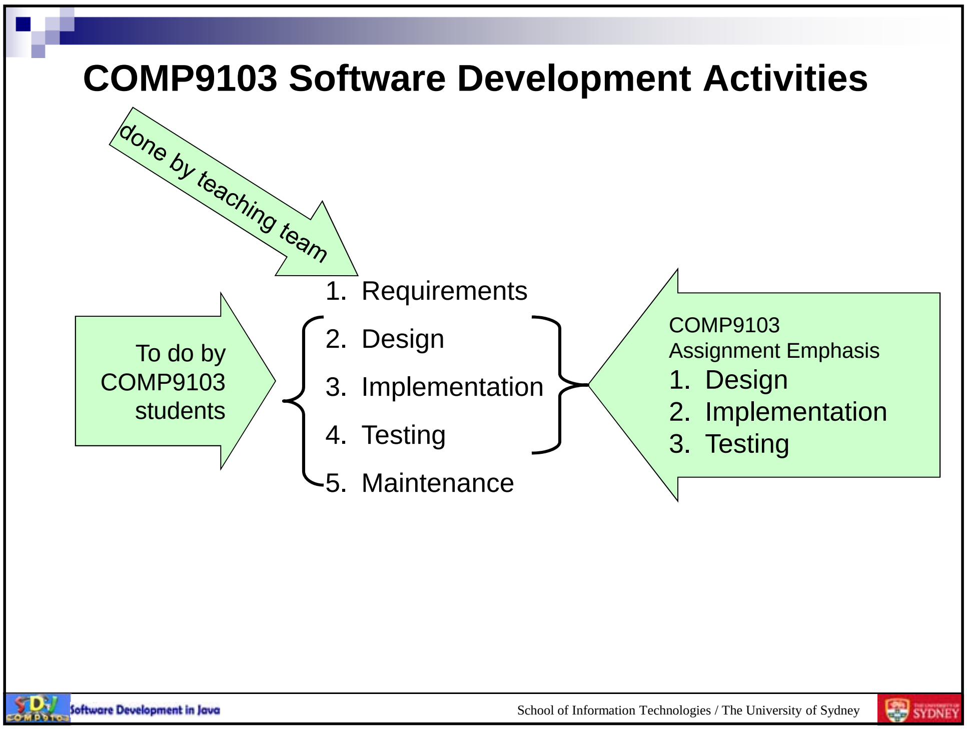
Other Models

- There are many many many software process models
- Some models are more suitable for some kinds of software development
- Each software-intensive business has its preferred model, and trains new employees in using their model.
- There are many businesses that develop and sell software process models

DEPLOYMENT
DISCOVERY
PLANNING
DEVELOPMENT
STABILIZATION
Planning
Risk analysis
Liaison
Engineering
System Evaluation
Construction & Release
Application Development
Technical Proposal & Review
Monitoring, Analysis, & Control
Business Process Analysis
APPLICATION DEVELOPMENT PROCESS
Deployment & Training
Business Model Identification
Maintenance & Support
Requirements Analysis
Review/Test
Operational Testing
High Level Design
Integration Testing
Detailed Specifications
Unit Testing
Coding

Software Development in Java

School of Information Technologies / The University of Sydney



COMP9103 Software Development Activities

done by teaching team

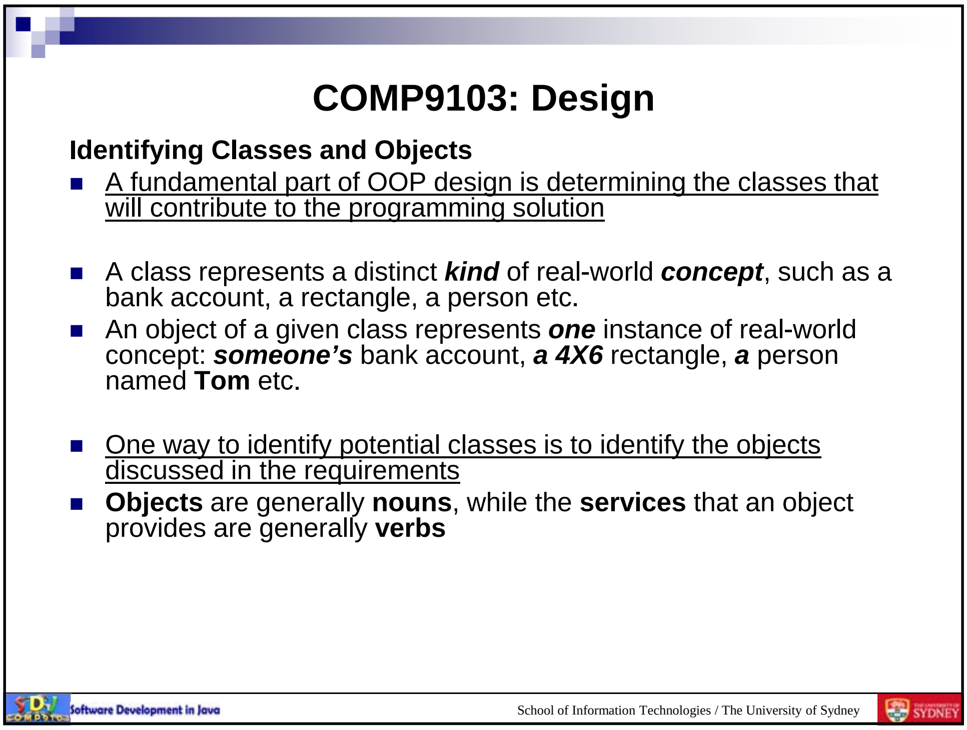
To do by COMP9103 students

1. Requirements
2. Design
3. Implementation
4. Testing
5. Maintenance

COMP9103
Assignment Emphasis
1. Design
2. Implementation
3. Testing

Software Development in Java

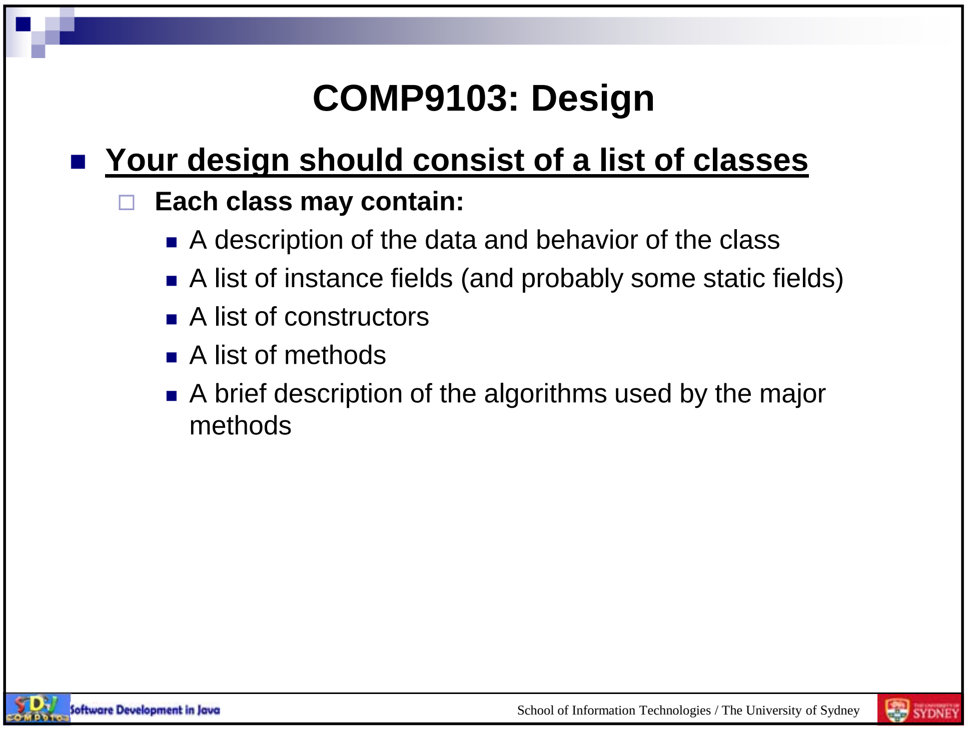
School of Information Technologies / The University of Sydney



COMP9103: Design

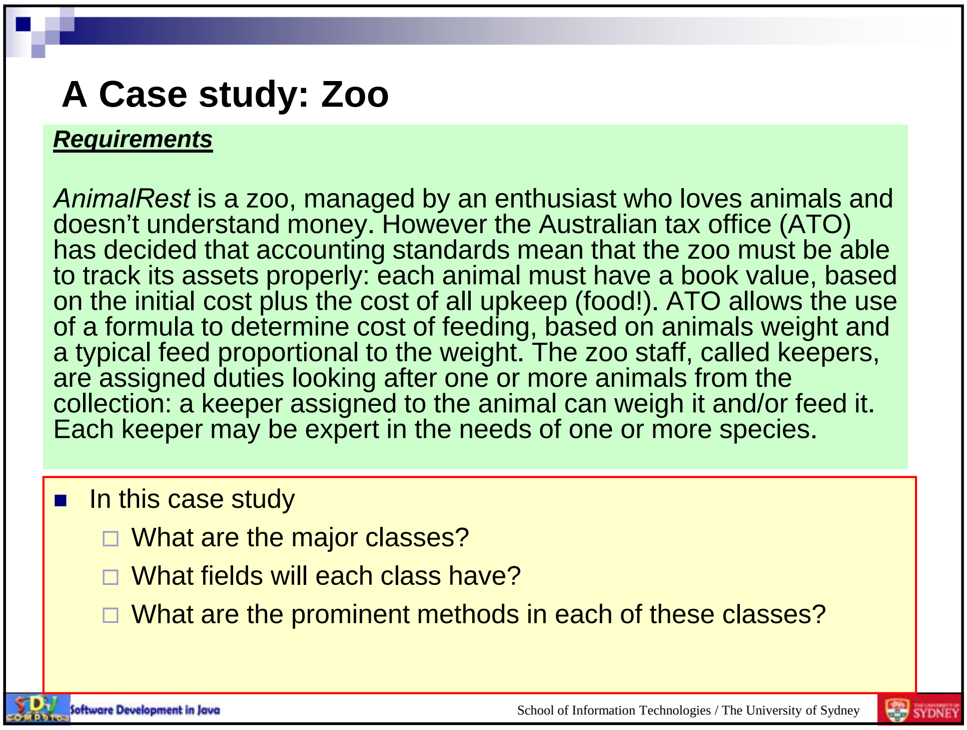
Identifying Classes and Objects

- A fundamental part of OOP design is determining the classes that will contribute to the programming solution
- A class represents a distinct ***kind*** of real-world ***concept***, such as a bank account, a rectangle, a person etc.
- An object of a given class represents ***one*** instance of real-world concept: ***someone's*** bank account, ***a 4X6*** rectangle, ***a*** person named **Tom** etc.
- One way to identify potential classes is to identify the objects discussed in the requirements
- **Objects** are generally **nouns**, while the **services** that an object provides are generally **verbs**



COMP9103: Design

- **Your design should consist of a list of classes**
  - □ **Each class may contain:**
    - A description of the data and behavior of the class
    - A list of instance fields (and probably some static fields)
    - A list of constructors
    - A list of methods
    - A brief description of the algorithms used by the major methods

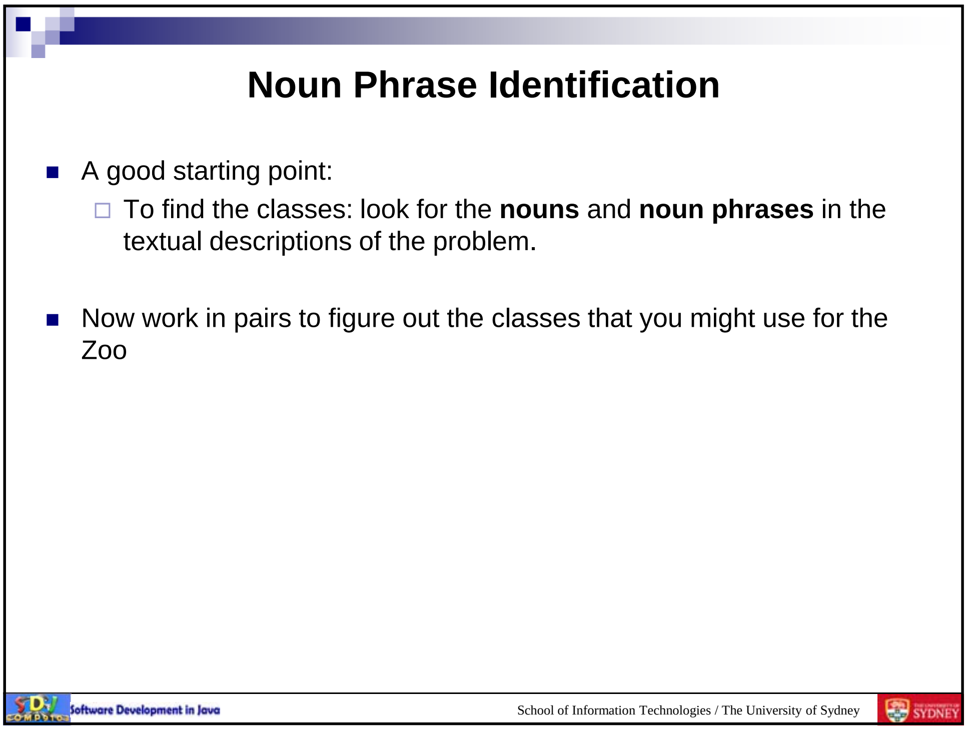


A Case study: Zoo

Requirements

AnimalRest is a zoo, managed by an enthusiast who loves animals and doesn’t understand money. However the Australian tax office (ATO) has decided that accounting standards mean that the zoo must be able to track its assets properly: each animal must have a book value, based on the initial cost plus the cost of all upkeep (food!). ATO allows the use of a formula to determine cost of feeding, based on animals weight and a typical feed proportional to the weight. The zoo staff, called keepers, are assigned duties looking after one or more animals from the collection: a keeper assigned to the animal can weigh it and/or feed it. Each keeper may be expert in the needs of one or more species.

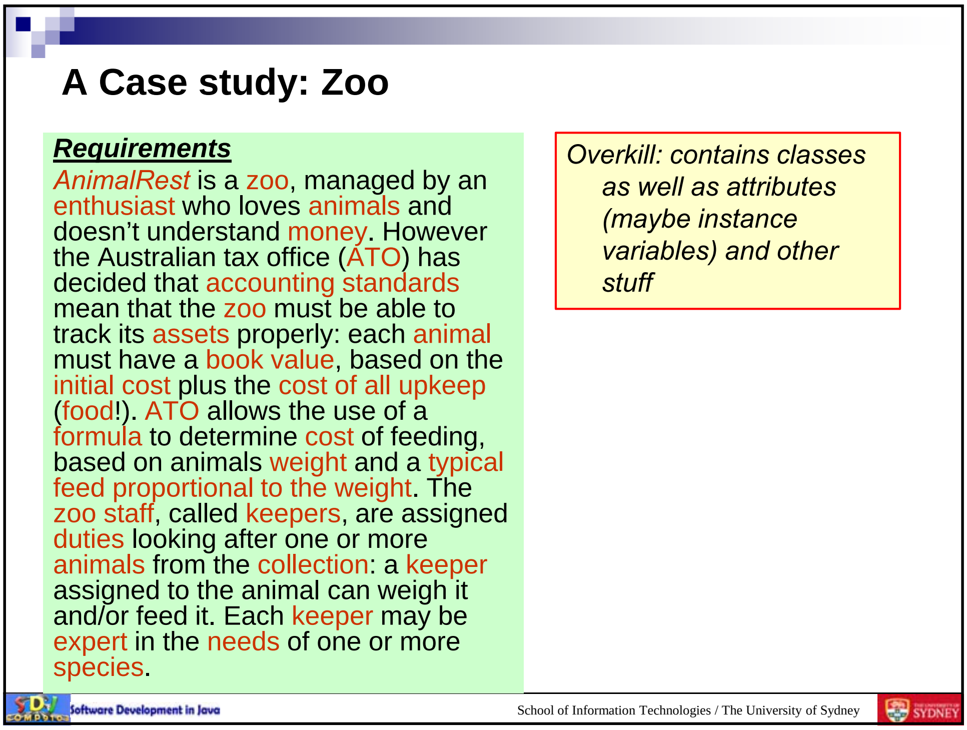
- In this case study
  - □ What are the major classes?
  - □ What fields will each class have?
  - □ What are the prominent methods in each of these classes?



Noun Phrase Identification

- A good starting point:
  - □ To find the classes: look for the **nouns** and **noun phrases** in the textual descriptions of the problem.

- Now work in pairs to figure out the classes that you might use for the Zoo



A Case study: Zoo

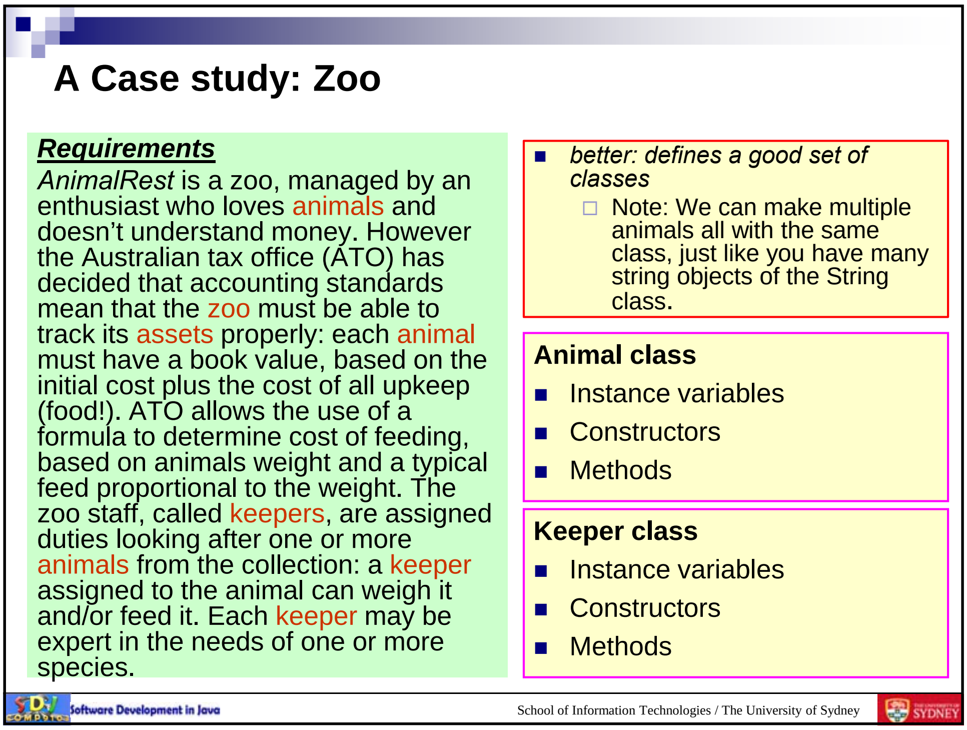
Requirements

AnimalRest is a zoo, managed by an enthusiast who loves animals and doesn't understand money. However the Australian tax office (ATO) has decided that accounting standards mean that the zoo must be able to track its assets properly: each animal must have a book value, based on the initial cost plus the cost of all upkeep (food!). ATO allows the use of a formula to determine cost of feeding, based on animals weight and a typical feed proportional to the weight. The zoo staff, called keepers, are assigned duties looking after one or more animals from the collection: a keeper assigned to the animal can weigh it and/or feed it. Each keeper may be expert in the needs of one or more species.

Overkill: contains classes as well as attributes (maybe instance variables) and other stuff

Software Development in Java

School of Information Technologies / The University of Sydney



A Case study: Zoo

Requirements

AnimalRest is a zoo, managed by an enthusiast who loves animals and doesn't understand money. However the Australian tax office (ATO) has decided that accounting standards mean that the zoo must be able to track its assets properly: each animal must have a book value, based on the initial cost plus the cost of all upkeep (food!). ATO allows the use of a formula to determine cost of feeding, based on animals weight and a typical feed proportional to the weight. The zoo staff, called keepers, are assigned duties looking after one or more animals from the collection: a keeper assigned to the animal can weigh it and/or feed it. Each keeper may be expert in the needs of one or more species.

- *better: defines a good set of classes*
  - Note: We can make multiple animals all with the same class, just like you have many string objects of the String class.

Animal class

- Instance variables
- Constructors
- Methods

Keeper class

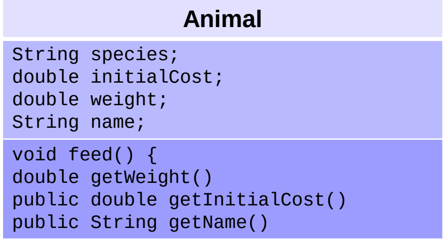
- Instance variables
- Constructors
- Methods

Software Development in Java

School of Information Technologies / The University of Sydney

UML Class Diagrams

- UML (Unified Modeling Language) notation is for conceptual models, ideal for OO code structure!
- Usually written *before* class is defined
- Used by the programmer defining the class
  - □ Contrast with the interface used by programmer who uses the class
- the UML class diagram represents class with box which contain three parts:
  - □ the name;
  - □ the fields; &
  - □ the methods



Animal

```
String species;
double initialCost;
double weight;
String name;
```

```
void feed() {
double getWeight()
public double getInitialCost()
public String getName()
```

Implementing a class

- **Determine instance fields for `Animal` class**

```
private String species;
private double initialCost;
private double weight;
private String name;
```

- **Access these *instance variables* using "getters"** and **"setters".**

Implementing a class

- **How can we construct an animal?**
  - □ A ***constructor*** is a piece of code that can be used to **make a new *instance* of a class**.
  - □ In order to *construct* an object that is more complex than a primitive type, we will need to initialize any instance fields it has.
  - □ Our fields are *species, initialCost, weight and name*.
    - We *could* decide to initialize them all in one go, so we could make a new animal like this:

```
Animal Bruce = new Animal("Lion", 60000, 190, “Bruce”);
```

    - *or* we could initialize them with no arguments and then set the values later:

```
Animal Leo = new Animal();
```

Implementing a class

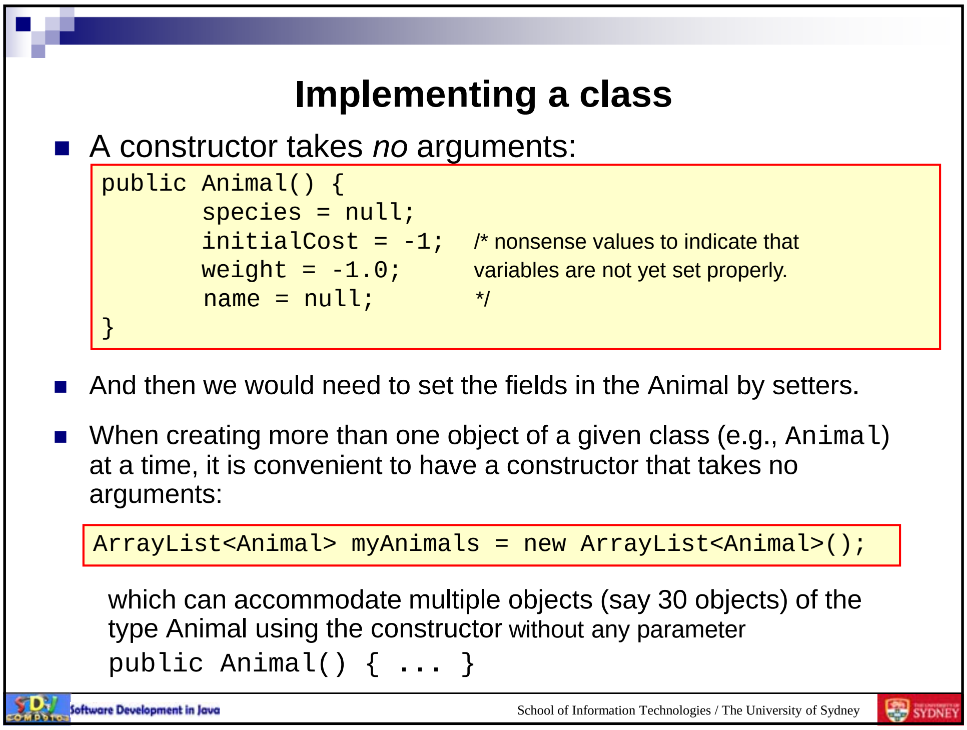
- Let’s write a constructor now:

```
public Animal(String s, double cost, double w, String n) {
       species = s;
       initialCost = cost;
       weight = m;
        name = n;
}
```

- Now we can make a new `Animal` like this:

```
Animal Zee = new Animal("Ant", 1, 0.000001,”Z”);
```

  - □ *Zee.species = "Ant";*
  - □ *Zee.initialCost = 1;*
  - □ *Zee.weight = 0.000001 =* 10^{-6}
  - □ *Zee.name = “Z”*



Implementing a class

- A constructor takes *no* arguments:

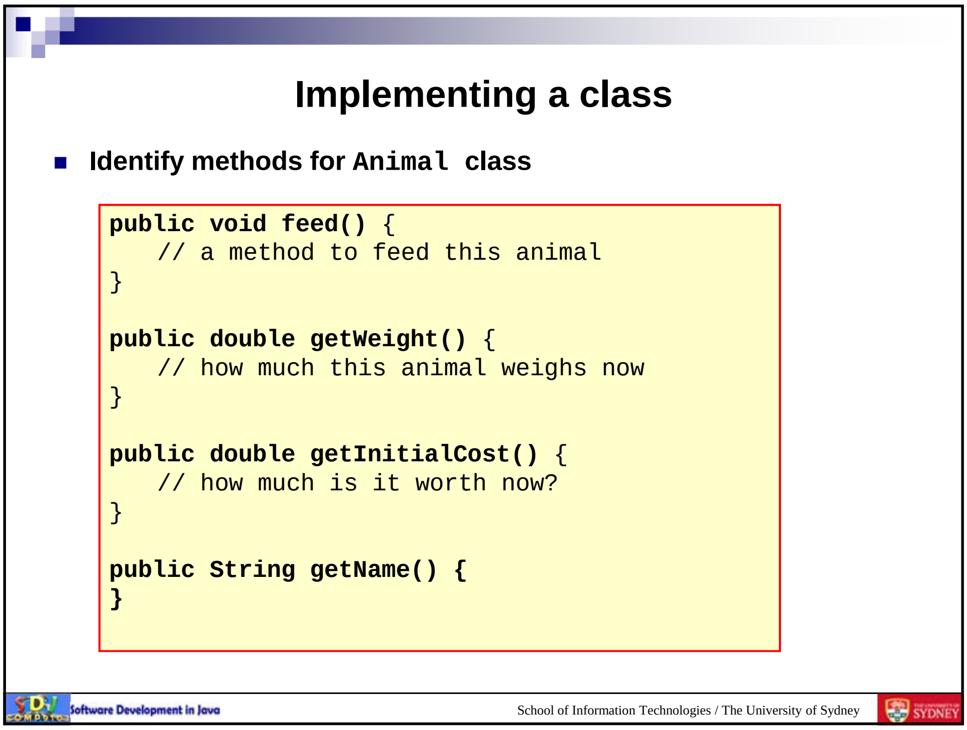
```
public Animal() {
       species = null;
       initialCost = -1;    /* nonsense values to indicate that
       weight = -1.0;       variables are not yet set properly.
       name = null;         */
}
```

- And then we would need to set the fields in the Animal by setters.
- When creating more than one object of a given class (e.g., `Animal`) at a time, it is convenient to have a constructor that takes no arguments:

```
ArrayList<Animal> myAnimals = new ArrayList<Animal>();
```

which can accommodate multiple objects (say 30 objects) of the type Animal using the constructor without any parameter

```
public Animal() { ... }
```



Implementing a class

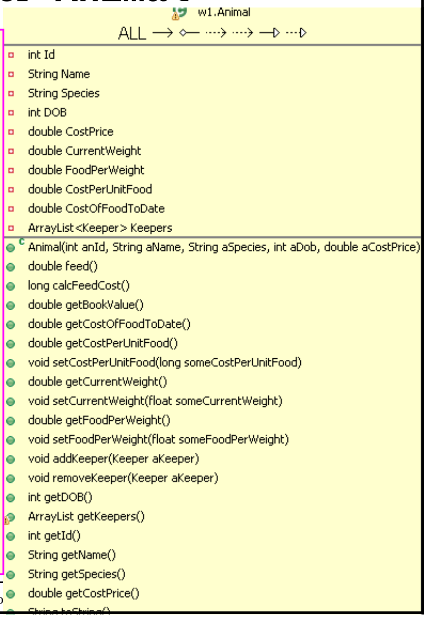
- **Identify methods for `Animal` class**

```
public void feed() {
   // a method to feed this animal
}

public double getWeight() {
   // how much this animal weighs now
}

public double getInitialCost() {
   // how much is it worth now?
}

public String getName() {
}
```

More Ingredients for `Animal`

Our class needs a few more ingredients:

- some instance variables:
  - □ an `ID` (for the records)
  - □ a `date of birth`
  - □ a `measure of its appetite`
  - □ `how much the food costs`
  - □ (maybe) *a `list of the keepers`* who can look after it
- and some methods:
  - □ `getBookValue()`
  - □ `calcFeedCost()`
  - □ `getVetCosts()`
  - □ `getTotalCostsToDate()`
  - □ . . .



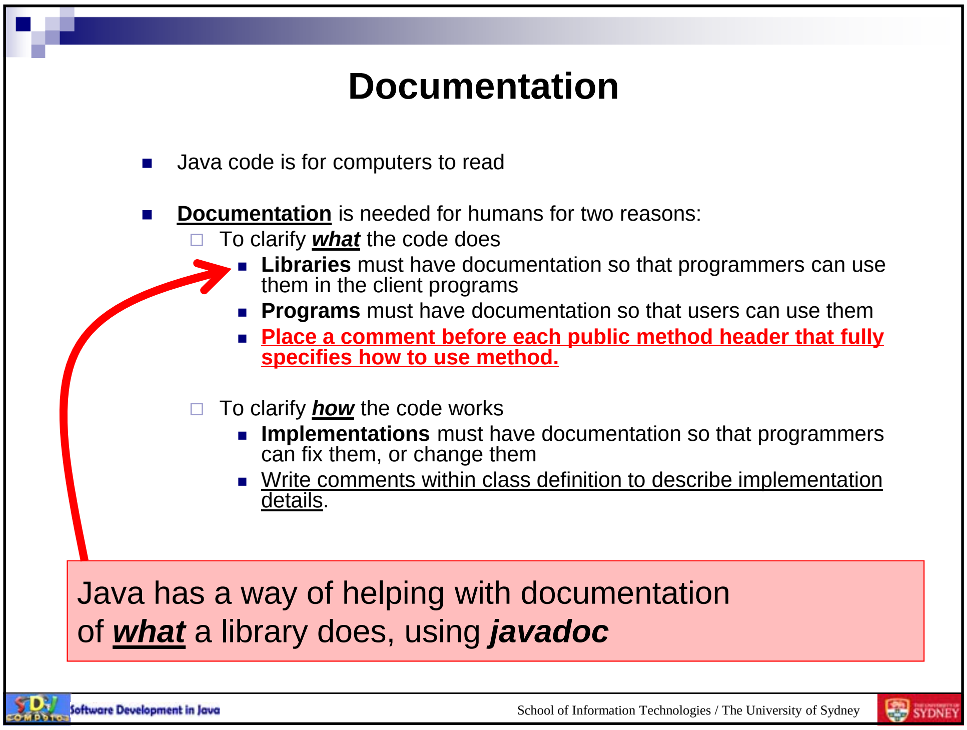
w1.Animal

ALL → ◇— ⇢ ⇢ —▷ ⋯▷

```
int Id
String Name
String Species
int DOB
double CostPrice
double CurrentWeight
double FoodPerWeight
double CostPerUnitFood
double CostOfFoodToDate
ArrayList<Keeper> Keepers
Animal(int anId, String aName, String aSpecies, int aDob, double aCostPrice)
double feed()
long calcFeedCost()
double getBookValue()
double getCostOfFoodToDate()
double getCostPerUnitFood()
void setCostPerUnitFood(long someCostPerUnitFood)
double getCurrentWeight()
void setCurrentWeight(float someCurrentWeight)
double getFoodPerWeight()
void setFoodPerWeight(float someFoodPerWeight)
void addKeeper(Keeper aKeeper)
void removeKeeper(Keeper aKeeper)
int getDOB()
ArrayList getKeepers()
int getId()
String getName()
String getSpecies()
double getCostPrice()
String toString()
```

API of a class

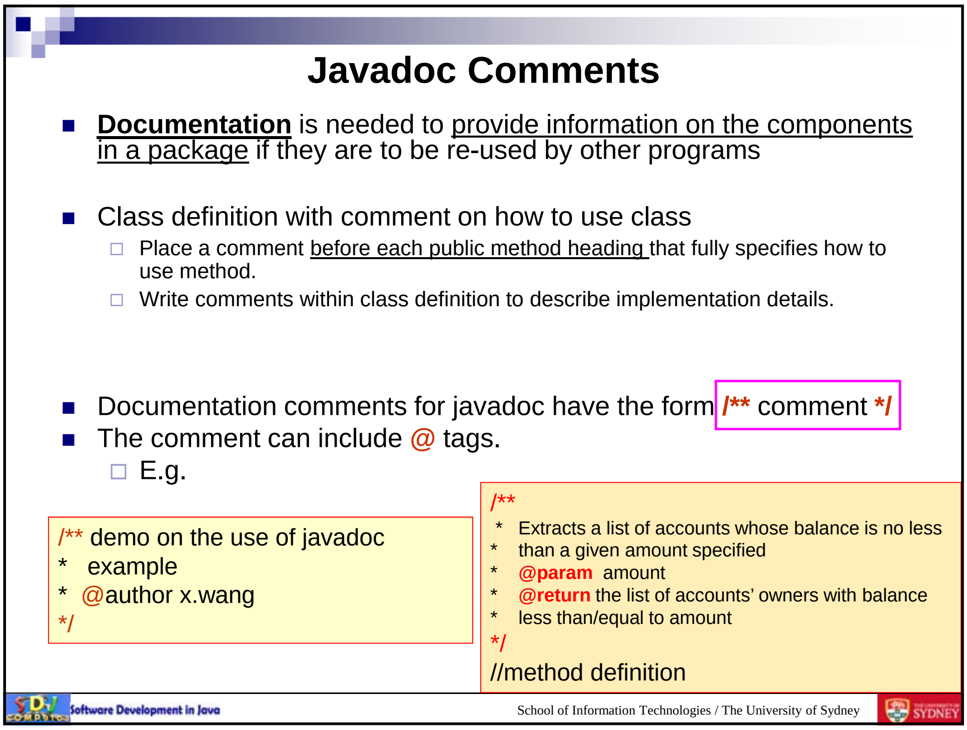
- **Application Programming Interface (API)** gives all the method profiles (its name, arguments, and return type) but not how each one works.
- The API allows any user to use the library without having to examine the code in the implementation.
- The guiding principle in **API design** is *to provide information on the methods to clients*:
  - □ what methods are in the library/class
  - □ what parameters they use
  - □ what tasks they perform



Documentation

- Java code is for computers to read
- **Documentation** is needed for humans for two reasons:
  - To clarify ***what*** the code does
    - **Libraries** must have documentation so that programmers can use them in the client programs
    - **Programs** must have documentation so that users can use them
    - **Place a comment before each public method header that fully specifies how to use method.**
  - To clarify ***how*** the code works
    - **Implementations** must have documentation so that programmers can fix them, or change them
    - Write comments within class definition to describe implementation details.

Java has a way of helping with documentation of ***what*** a library does, using ***javadoc***



Javadoc Comments

- **Documentation** is needed to provide information on the components in a package if they are to be re-used by other programs
- Class definition with comment on how to use class
  - Place a comment before each public method heading that fully specifies how to use method.
  - Write comments within class definition to describe implementation details.
- Documentation comments for javadoc have the form **/**** comment ***/**
- The comment can include @ tags.
  - E.g.

```
/** demo on the use of javadoc
*   example
*  @author x.wang
*/
```

```
/**
 *   Extracts a list of accounts whose balance is no less
*    than a given amount specified
*    @param  amount
*    @return the list of accounts' owners with balance
*    less than/equal to amount
*/
//method definition
```

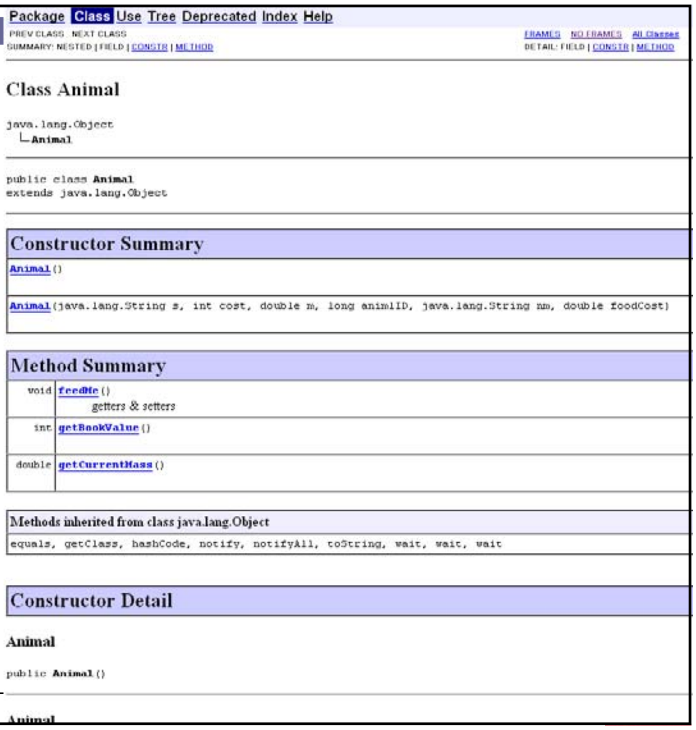
Javadoc

- ***javadoc*** extracts information from **class and method headers** and the **programmer's javadoc comments**
- Typing

  javadoc *.java

  will run javadoc on all your java files and produce ***HTML*** files
- We also can generate javadoc by using Eclipse IDE
  - □Choose ***Project → Generate Javadoc***
  - □You will practice on this in lab

Animal API


Package
Class
Use
Tree
Deprecated
Index
Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Class Animal

```
java.lang.Object
  └Animal
```

```
public class Animal
extends java.lang.Object
```

Constructor Summary
Animal()
Animal(java.lang.String s, int cost, double m, long animlID, java.lang.String nm, double foodCost)

Method Summary	
void	feedMe() getters & setters
int	getBookValue()
double	getCurrentMass()

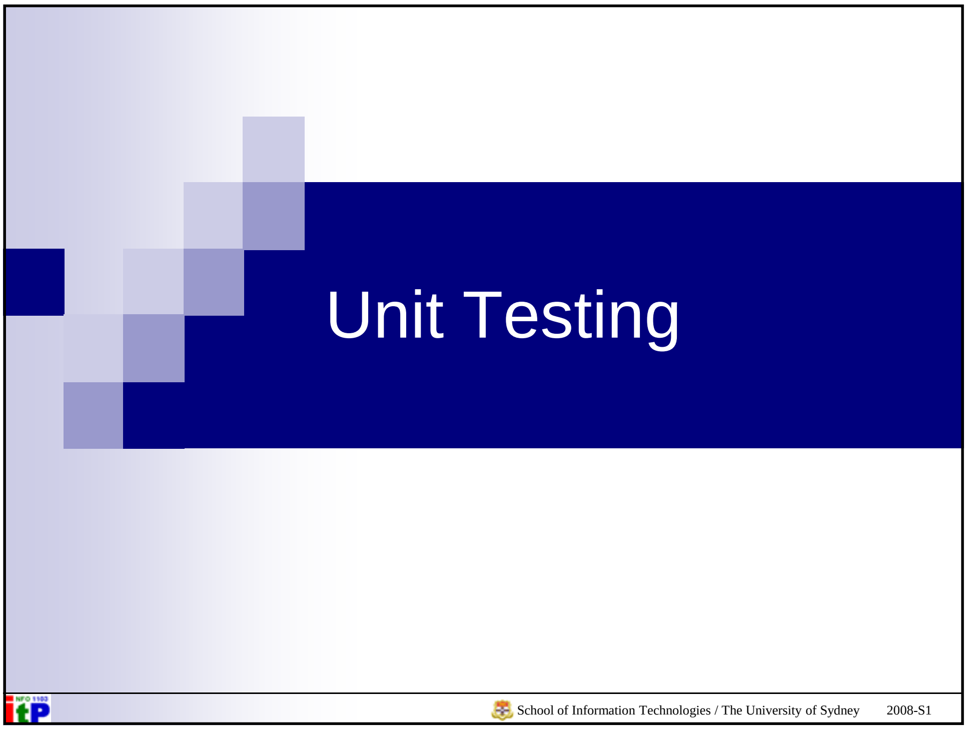
Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

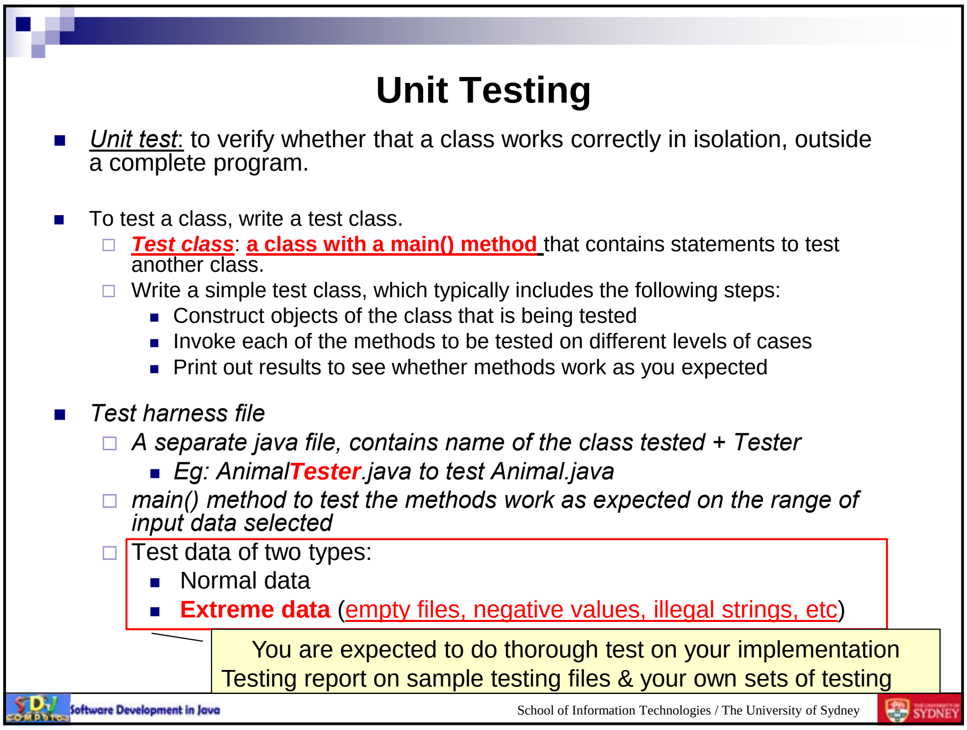
Animal

```
public Animal()
```

Animal



Unit Testing



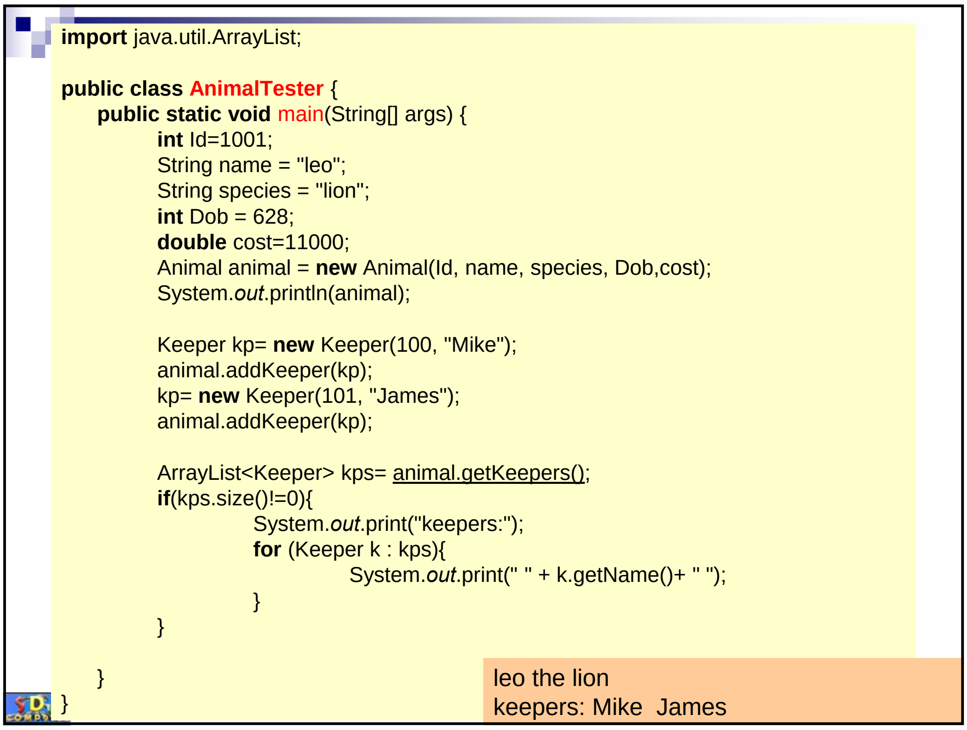
Unit Testing

- *Unit test:* to verify whether that a class works correctly in isolation, outside a complete program.

- To test a class, write a test class.
  - ***Test class***: **a class with a main() method** that contains statements to test another class.
  - Write a simple test class, which typically includes the following steps:
    - Construct objects of the class that is being tested
    - Invoke each of the methods to be tested on different levels of cases
    - Print out results to see whether methods work as you expected

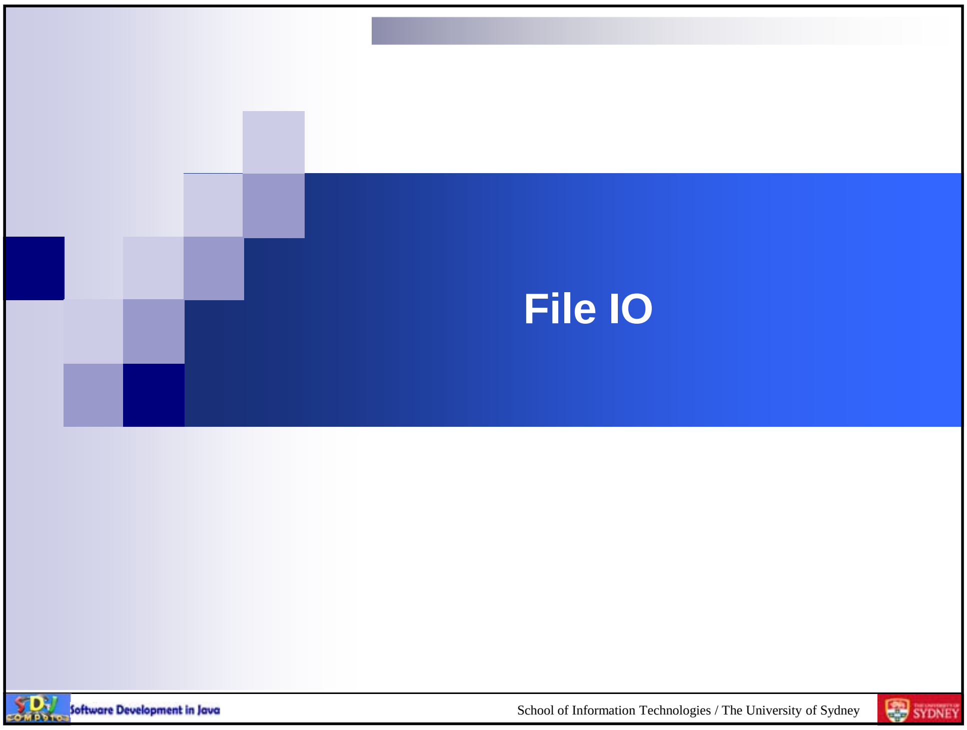
- *Test harness file*
  - *A separate java file, contains name of the class tested + Tester*
    - *Eg: Animal**Tester**.java to test Animal.java*
  - *main() method to test the methods work as expected on the range of input data selected*
  - Test data of two types:
    - Normal data
    - **Extreme data** (empty files, negative values, illegal strings, etc)

You are expected to do thorough test on your implementation
Testing report on sample testing files & your own sets of testing



```
import java.util.ArrayList;

public class AnimalTester {
    public static void main(String[] args) {
        int Id=1001;
        String name = "leo";
        String species = "lion";
        int Dob = 628;
        double cost=11000;
        Animal animal = new Animal(Id, name, species, Dob,cost);
        System.out.println(animal);

        Keeper kp= new Keeper(100, "Mike");
        animal.addKeeper(kp);
        kp= new Keeper(101, "James");
        animal.addKeeper(kp);

        ArrayList<Keeper> kps= animal.getKeepers();
        if(kps.size()!=0){
            System.out.print("keepers:");
            for (Keeper k : kps){
                System.out.print(" " + k.getName()+ " ");
            }
        }

    }
}
```

```
leo the lion
keepers: Mike  James
```

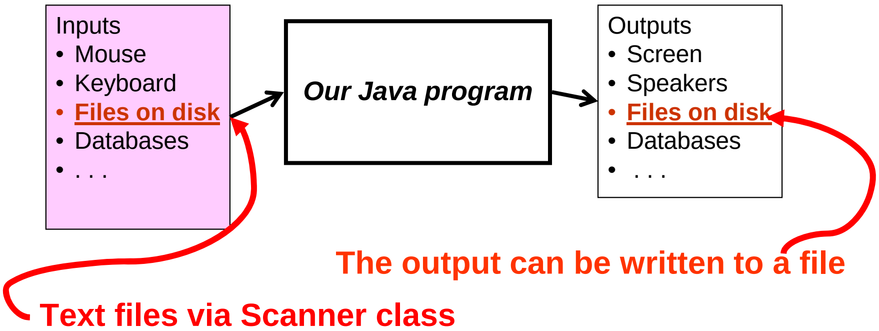


File IO

Software Development in Java

School of Information Technologies / The University of Sydney

Java I/O


Inputs
- Mouse
- Keyboard
- **Files on disk**
- Databases
- . . .

Our Java program

Outputs
- Screen
- Speakers
- **Files on disk**
- Databases
- . . .

The output can be written to a file

Text files via Scanner class

Advantages of File I/O

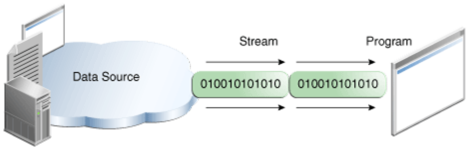
Input can be automated (rather than being entered manually)

Permanent copy

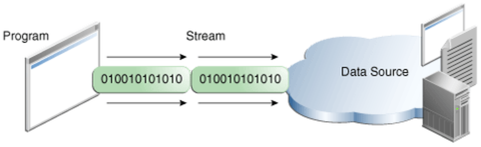
Output from one program can be input to another

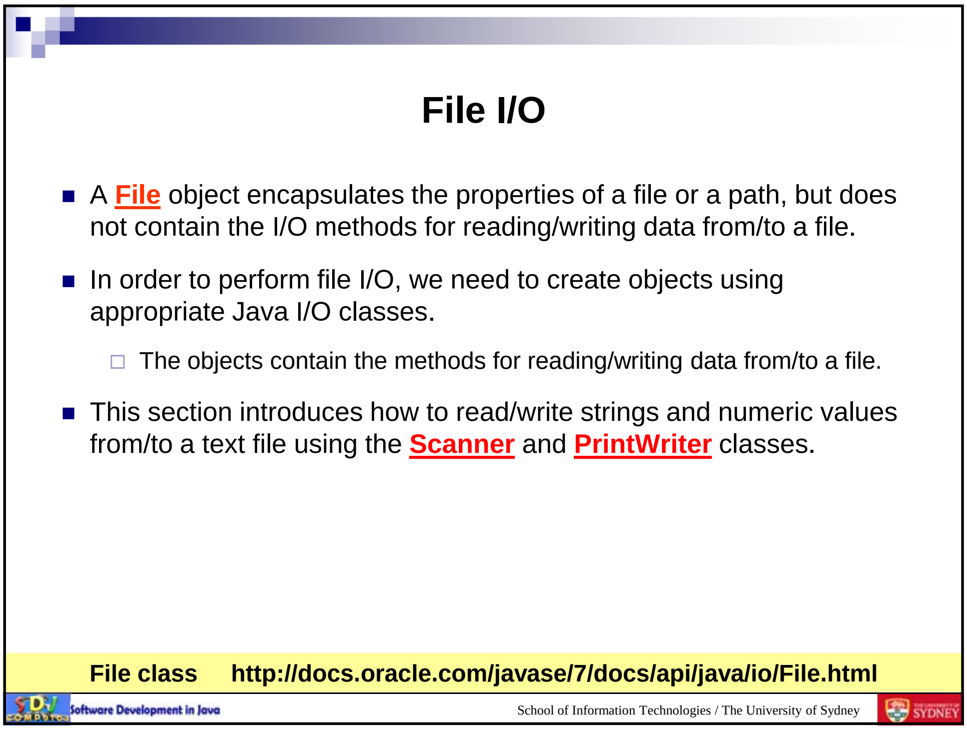
Java I/O Streams

- **An I/O Stream** represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files.
- A stream is a sequence of data.
- A program uses an input stream to read data from a source, one item at a time:



- A program uses an output stream to write data to a destination, one item at time:



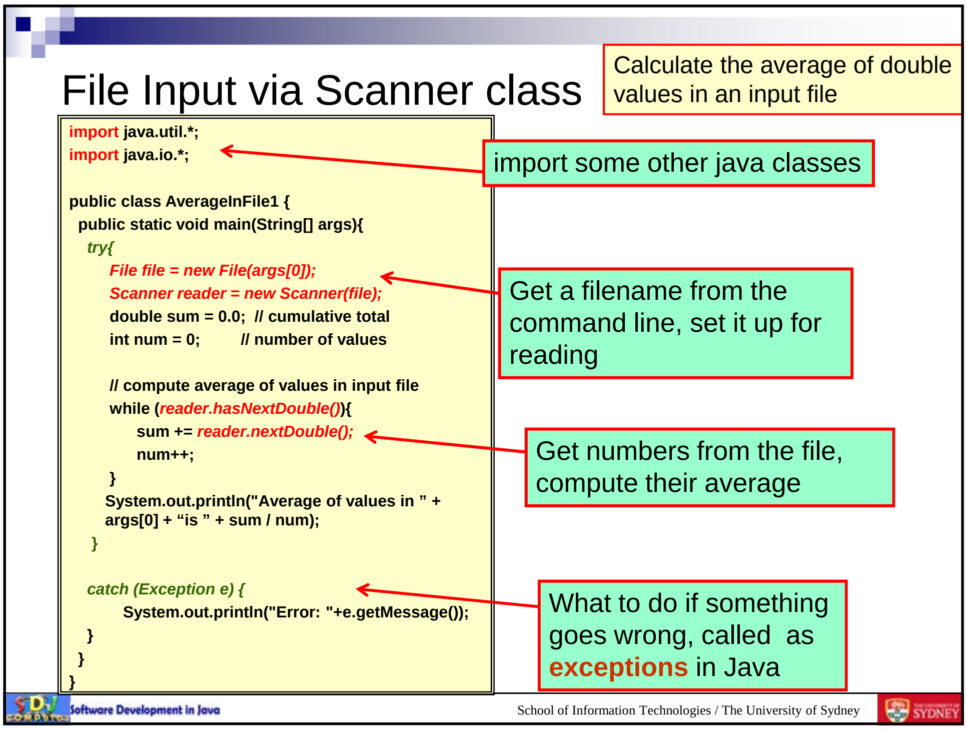


File I/O

- A **File** object encapsulates the properties of a file or a path, but does not contain the I/O methods for reading/writing data from/to a file.
- In order to perform file I/O, we need to create objects using appropriate Java I/O classes.
  - The objects contain the methods for reading/writing data from/to a file.
- This section introduces how to read/write strings and numeric values from/to a text file using the **Scanner** and **PrintWriter** classes.

File class http://docs.oracle.com/javase/7/docs/api/java/io/File.html

Software Development in Java School of Information Technologies / The University of Sydney



File Input via Scanner class

Calculate the average of double values in an input file

```
import java.util.*;
import java.io.*;

public class AverageInFile1 {
  public static void main(String[] args){
    try{
        File file = new File(args[0]);
        Scanner reader = new Scanner(file);
        double sum = 0.0;  // cumulative total
        int num = 0;        // number of values

        // compute average of values in input file
        while (reader.hasNextDouble()){
              sum += reader.nextDouble();
              num++;
        }
       System.out.println("Average of values in ” +
       args[0] + “is ” + sum / num);
    }

    catch (Exception e) {
          System.out.println("Error: "+e.getMessage());
    }
  }
}
```

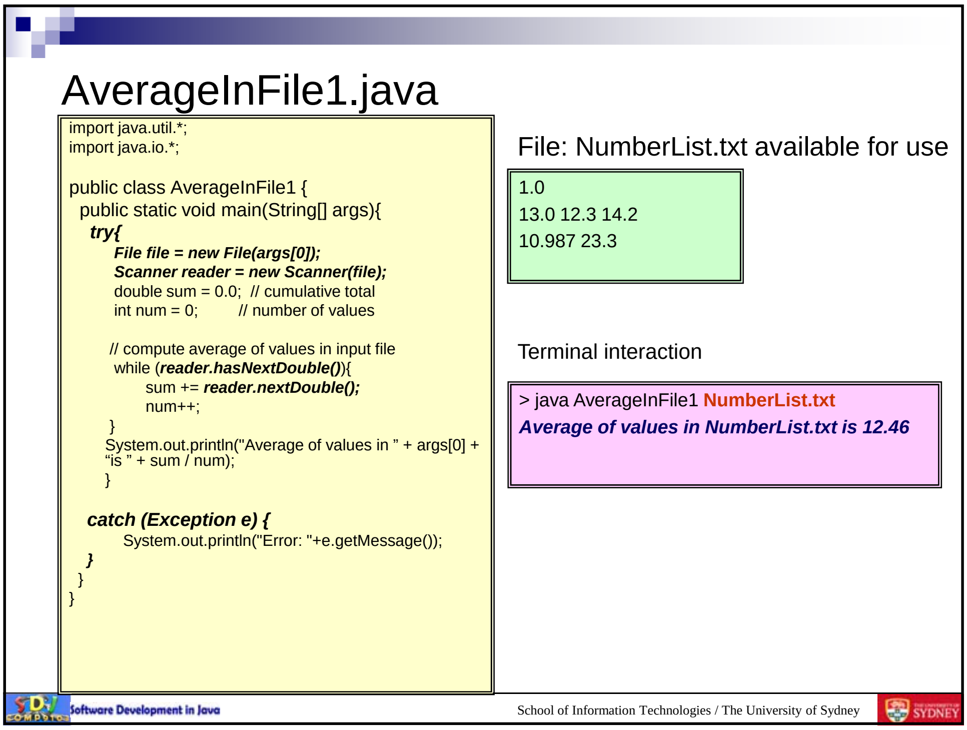
import some other java classes

Get a filename from the command line, set it up for reading

Get numbers from the file, compute their average

What to do if something goes wrong, called as **exceptions** in Java

Software Development in Java School of Information Technologies / The University of Sydney



AverageInFile1.java

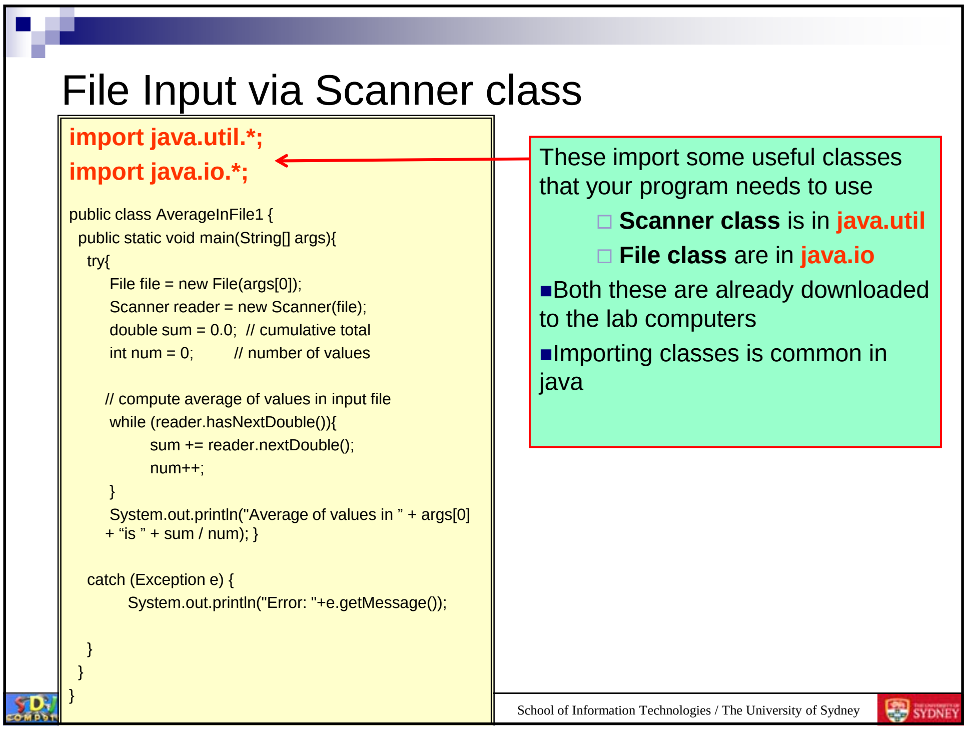
```
import java.util.*;
import java.io.*;

public class AverageInFile1 {
  public static void main(String[] args){
    try{
        File file = new File(args[0]);
        Scanner reader = new Scanner(file);
        double sum = 0.0;  // cumulative total
        int num = 0;         // number of values

       // compute average of values in input file
        while (reader.hasNextDouble()){
               sum += reader.nextDouble();
               num++;
       }
      System.out.println("Average of values in " + args[0] +
      “is ” + sum / num);
      }

    catch (Exception e) {
           System.out.println("Error: "+e.getMessage());
    }
  }
}
```

File: NumberList.txt available for use

```
1.0
13.0 12.3 14.2
10.987 23.3
```

Terminal interaction

```
> java AverageInFile1 NumberList.txt
Average of values in NumberList.txt is 12.46
```



File Input via Scanner class

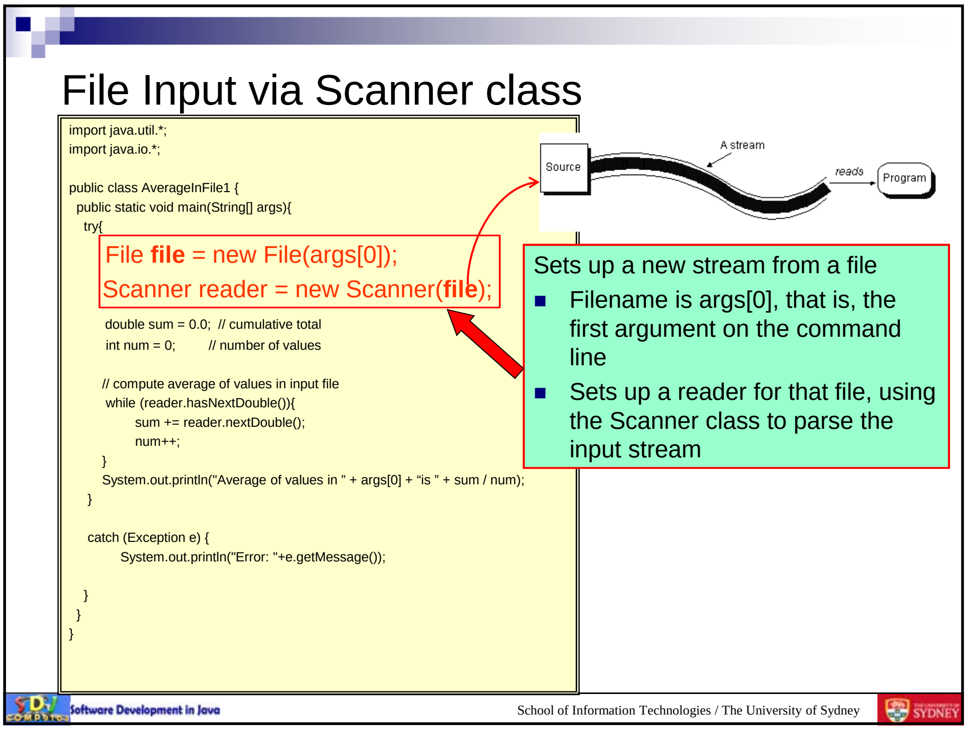
```
import java.util.*;
import java.io.*;

public class AverageInFile1 {
  public static void main(String[] args){
    try{
        File file = new File(args[0]);
        Scanner reader = new Scanner(file);
        double sum = 0.0;  // cumulative total
        int num = 0;         // number of values

       // compute average of values in input file
        while (reader.hasNextDouble()){
                sum += reader.nextDouble();
                num++;
        }
        System.out.println("Average of values in ” + args[0]
       + “is ” + sum / num); }

    catch (Exception e) {
            System.out.println("Error: "+e.getMessage());

    }
  }
}
```

These import some useful classes that your program needs to use

- □ **Scanner class** is in **java.util**
- □ **File class** are in **java.io**

■Both these are already downloaded to the lab computers

■Importing classes is common in java

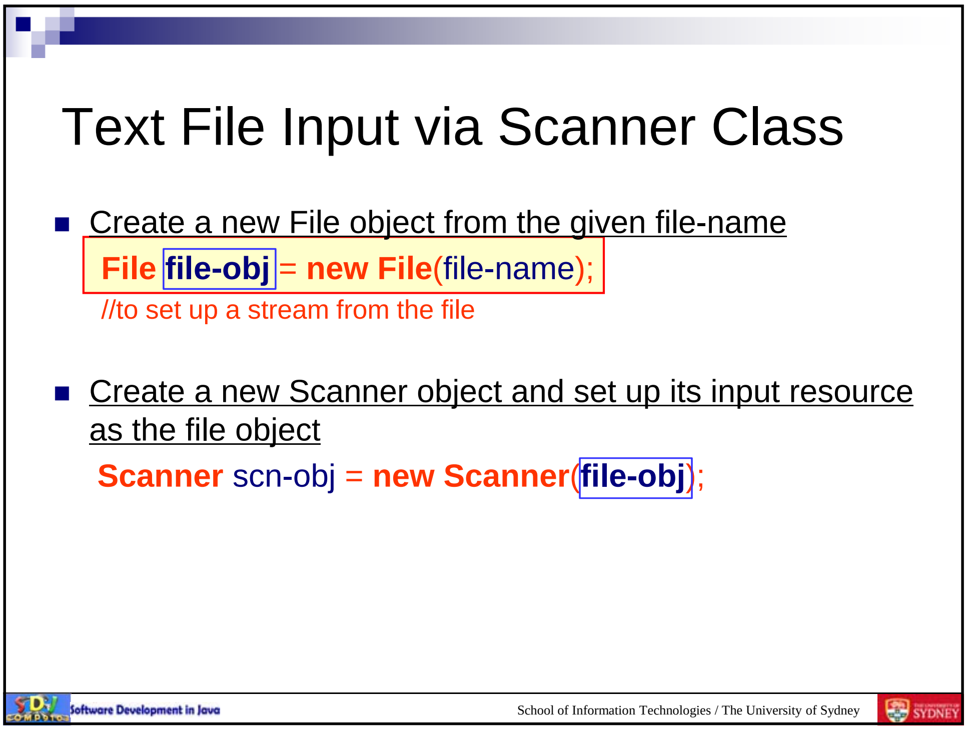


File Input via Scanner class

```
import java.util.*;
import java.io.*;

public class AverageInFile1 {
  public static void main(String[] args){
    try{
      File file = new File(args[0]);
      Scanner reader = new Scanner(file);

      double sum = 0.0;  // cumulative total
      int num = 0;        // number of values

      // compute average of values in input file
      while (reader.hasNextDouble()){
          sum += reader.nextDouble();
          num++;
      }
      System.out.println("Average of values in " + args[0] + "is " + sum / num);
    }

    catch (Exception e) {
         System.out.println("Error: "+e.getMessage());

    }
  }
}
```

Sets up a new stream from a file

- Filename is args[0], that is, the first argument on the command line
- Sets up a reader for that file, using the Scanner class to parse the input stream



Text File Input via Scanner Class

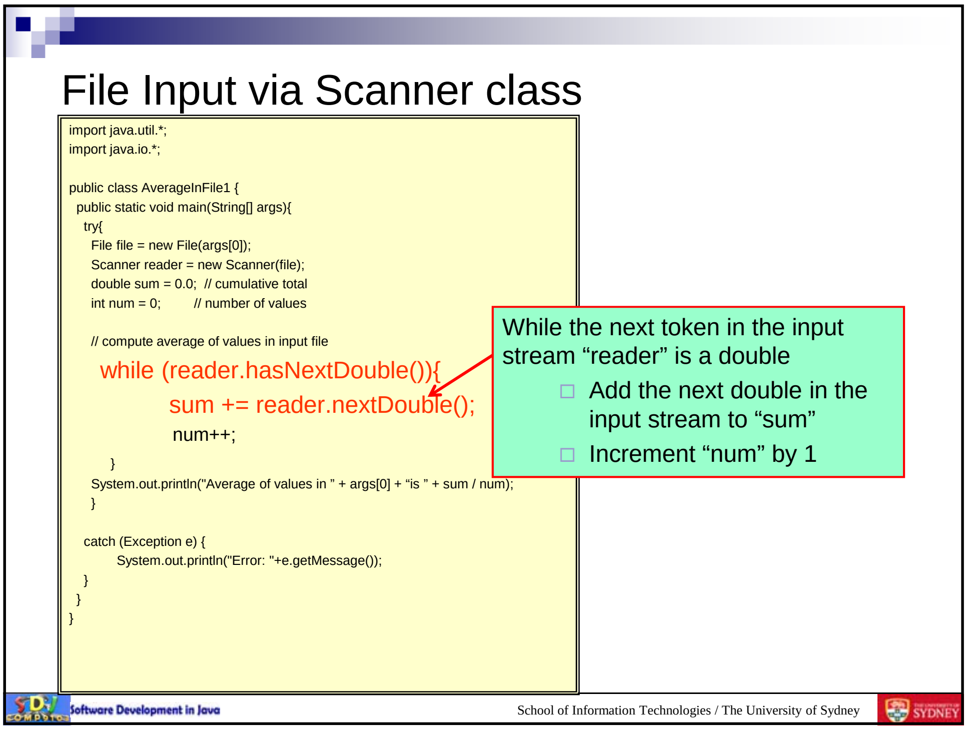
- Create a new File object from the given file-name

  File file-obj = new File(file-name);

  //to set up a stream from the file

- Create a new Scanner object and set up its input resource as the file object

  Scanner scn-obj = **new Scanner**(**file-obj**);



File Input via Scanner class

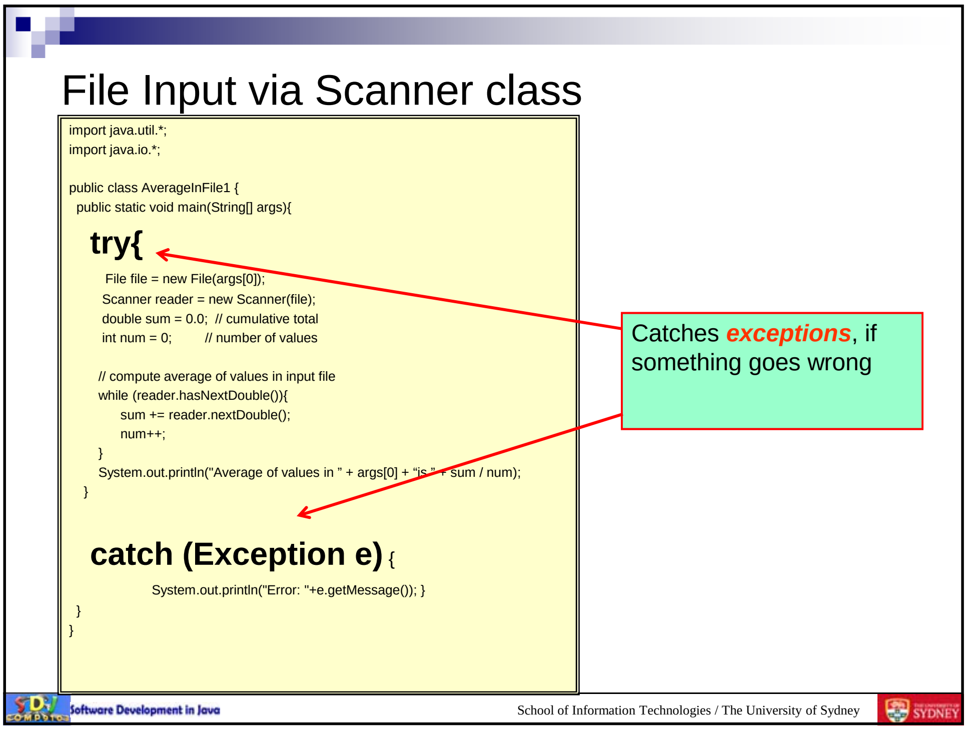
```
import java.util.*;
import java.io.*;

public class AverageInFile1 {
  public static void main(String[] args){
    try{
      File file = new File(args[0]);
      Scanner reader = new Scanner(file);
      double sum = 0.0;  // cumulative total
      int num = 0;        // number of values

      // compute average of values in input file
        while (reader.hasNextDouble()){
                  sum += reader.nextDouble();
                  num++;
        }
      System.out.println("Average of values in " + args[0] + “is ” + sum / num);
      }

    catch (Exception e) {
          System.out.println("Error: "+e.getMessage());
    }
  }
}
```

While the next token in the input stream “reader” is a double

- □ Add the next double in the input stream to “sum”
- □ Increment “num” by 1

Software Development in Java

School of Information Technologies / The University of Sydney



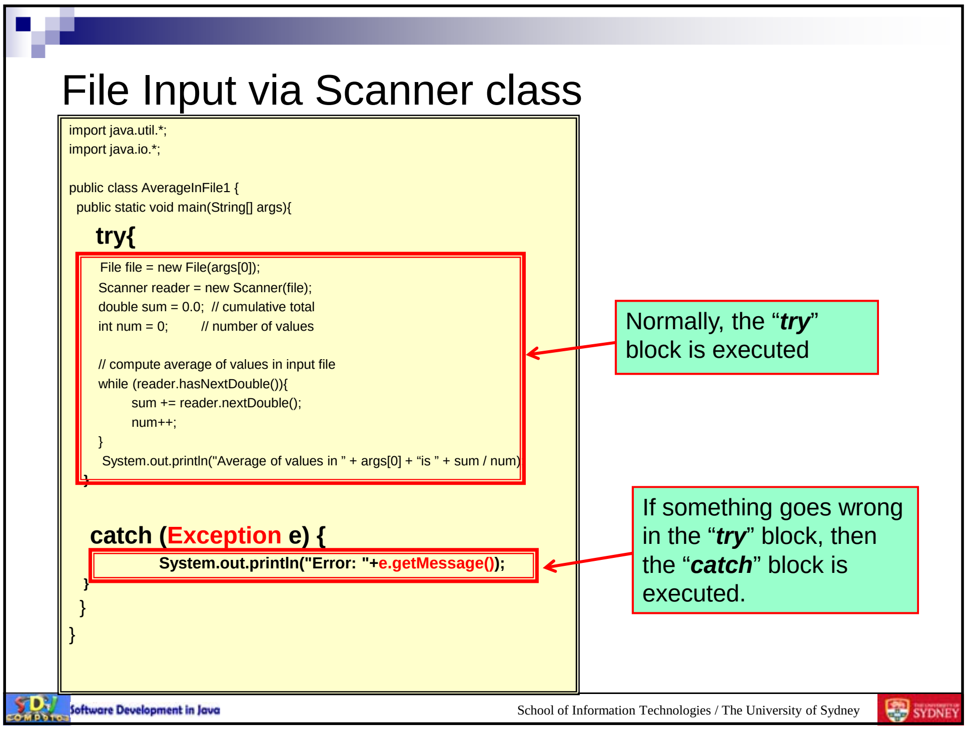
File Input via Scanner class

```
import java.util.*;
import java.io.*;

public class AverageInFile1 {
  public static void main(String[] args){
   try{
        File file = new File(args[0]);
       Scanner reader = new Scanner(file);
       double sum = 0.0;  // cumulative total
       int num = 0;        // number of values

      // compute average of values in input file
      while (reader.hasNextDouble()){
            sum += reader.nextDouble();
            num++;
      }
      System.out.println("Average of values in " + args[0] + “is ” + sum / num);
    }

   catch (Exception e) {
                System.out.println("Error: "+e.getMessage()); }
  }
}
```

Catches ***exceptions***, if something goes wrong

Software Development in Java

School of Information Technologies / The University of Sydney



File Input via Scanner class

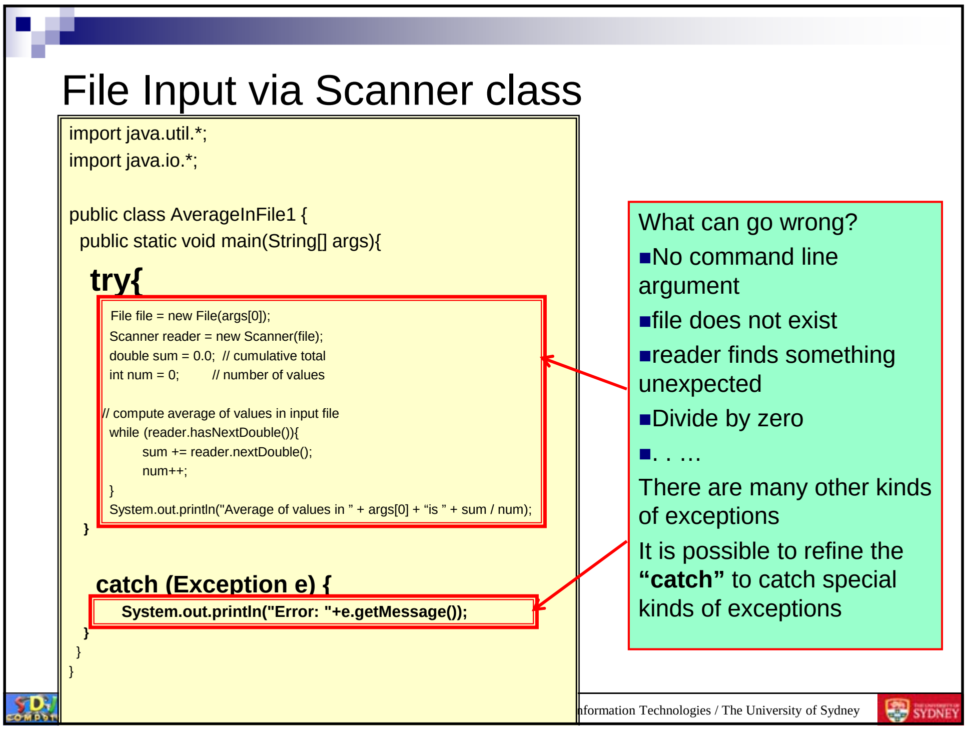
```
import java.util.*;
import java.io.*;

public class AverageInFile1 {
  public static void main(String[] args){
    try{
      File file = new File(args[0]);
      Scanner reader = new Scanner(file);
      double sum = 0.0;  // cumulative total
      int num = 0;         // number of values

      // compute average of values in input file
      while (reader.hasNextDouble()){
              sum += reader.nextDouble();
              num++;
      }
       System.out.println("Average of values in " + args[0] + "is " + sum / num);
    }

    catch (Exception e) {
              System.out.println("Error: "+e.getMessage());
    }
  }
}
```

Normally, the "***try***" block is executed

If something goes wrong in the "***try***" block, then the "***catch***" block is executed.

Software Development in Java

School of Information Technologies / The University of Sydney



File Input via Scanner class

```
import java.util.*;
import java.io.*;

public class AverageInFile1 {
  public static void main(String[] args){
    try{
      File file = new File(args[0]);
      Scanner reader = new Scanner(file);
      double sum = 0.0;  // cumulative total
      int num = 0;         // number of values

    // compute average of values in input file
      while (reader.hasNextDouble()){
              sum += reader.nextDouble();
              num++;
      }
      System.out.println("Average of values in " + args[0] + "is " + sum / num);
    }

    catch (Exception e) {
      System.out.println("Error: "+e.getMessage());
    }
  }
}
```

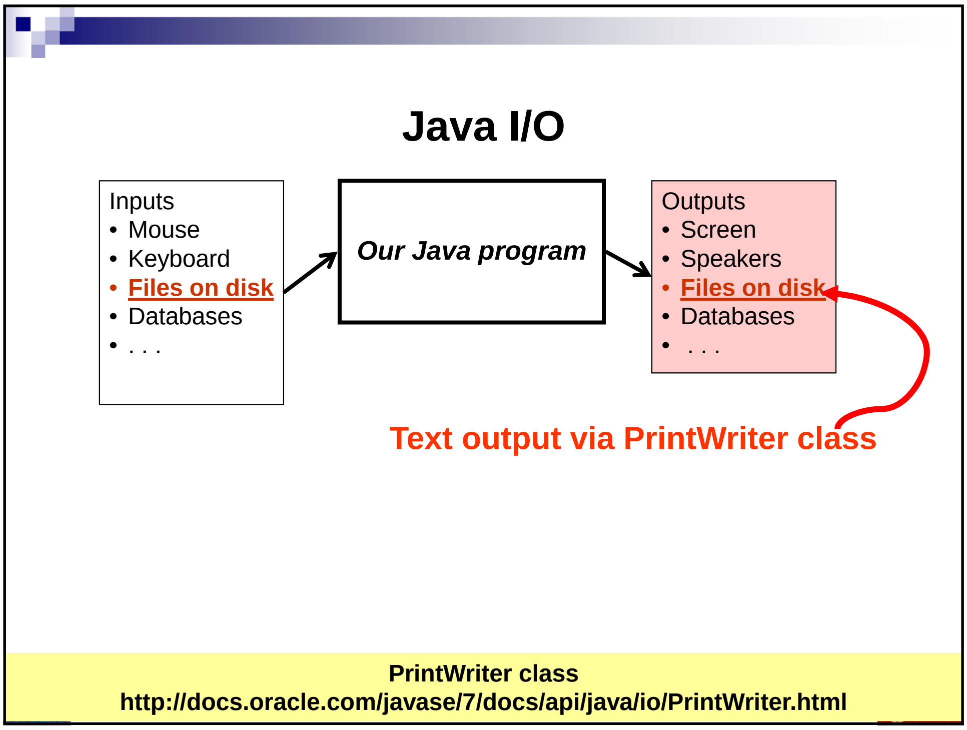
What can go wrong?

- No command line argument
- file does not exist
- reader finds something unexpected
- Divide by zero
- . . …

There are many other kinds of exceptions

It is possible to refine the **"catch"** to catch special kinds of exceptions

nformation Technologies / The University of Sydney



Java I/O

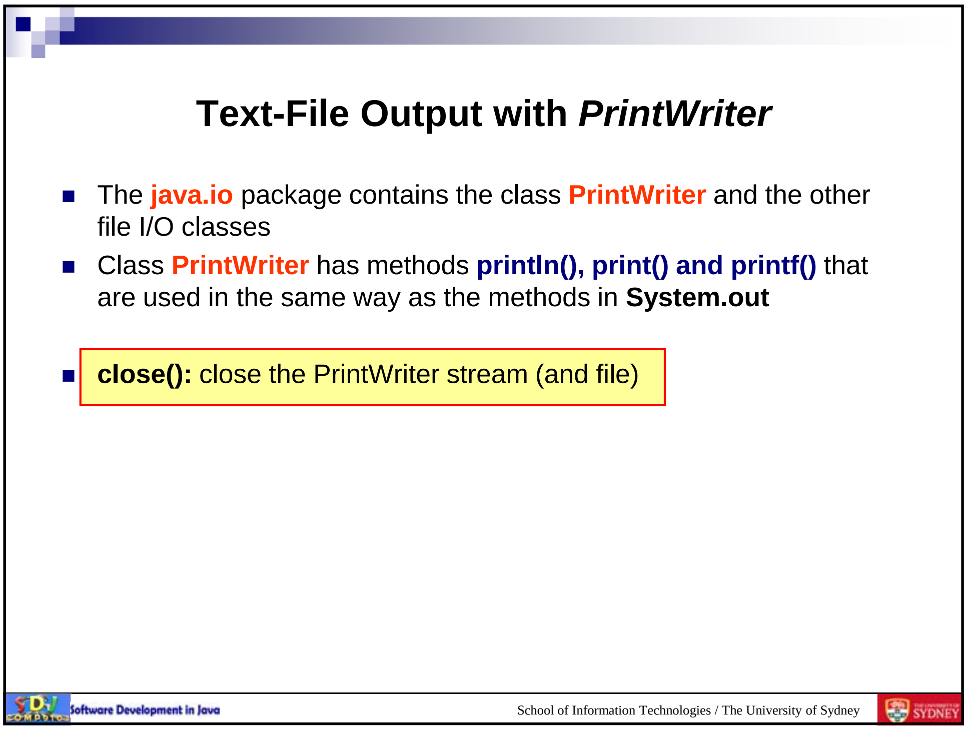
Inputs
- Mouse
- Keyboard
- **Files on disk**
- Databases
- . . .

Our Java program

Outputs
- Screen
- Speakers
- **Files on disk**
- Databases
- . . .

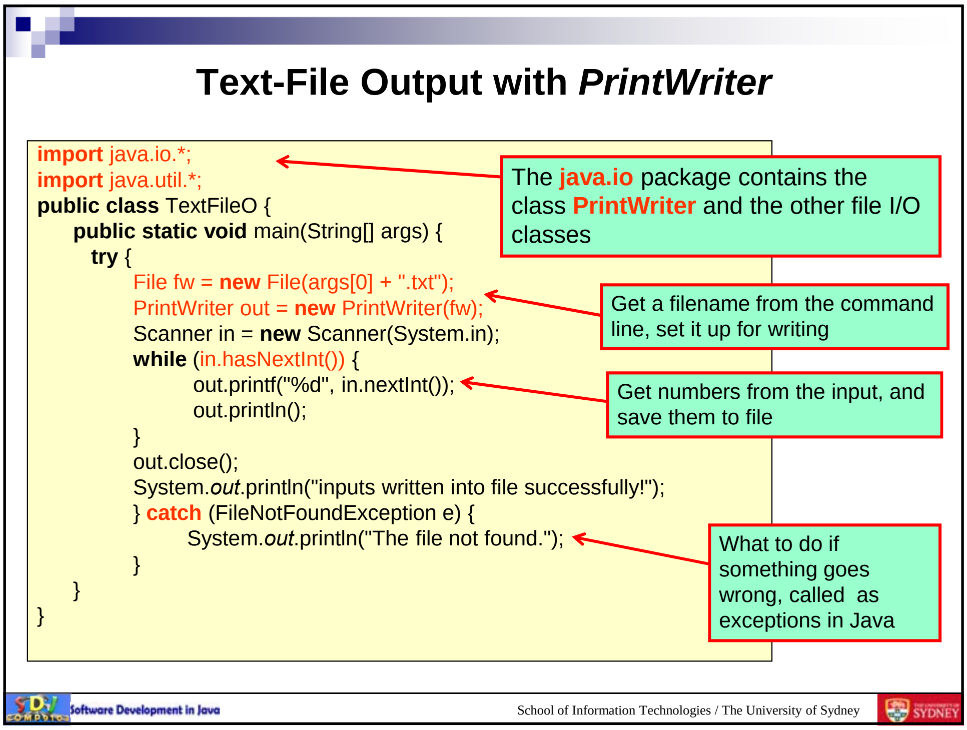
Text output via PrintWriter class

PrintWriter class
http://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html



Text-File Output with *PrintWriter*

- The **java.io** package contains the class **PrintWriter** and the other file I/O classes
- Class **PrintWriter** has methods **println(), print() and printf()** that are used in the same way as the methods in **System.out**
- **close():** close the PrintWriter stream (and file)



Text-File Output with *PrintWriter*

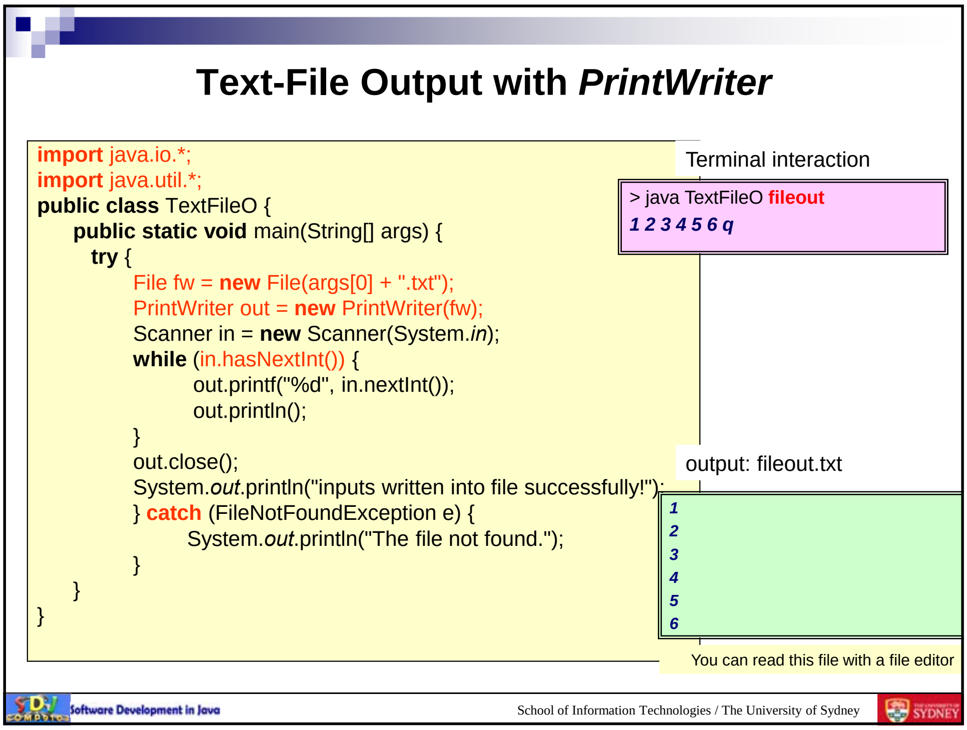
```
import java.io.*;
import java.util.*;
public class TextFileO {
     public static void main(String[] args) {
        try {
              File fw = new File(args[0] + ".txt");
              PrintWriter out = new PrintWriter(fw);
              Scanner in = new Scanner(System.in);
              while (in.hasNextInt()) {
                      out.printf("%d", in.nextInt());
                      out.println();
              }
              out.close();
              System.out.println("inputs written into file successfully!");
              } catch (FileNotFoundException e) {
                      System.out.println("The file not found.");
              }
     }
}
```

The **java.io** package contains the class **PrintWriter** and the other file I/O classes

Get a filename from the command line, set it up for writing

Get numbers from the input, and save them to file

What to do if something goes wrong, called as exceptions in Java



Text-File Output with *PrintWriter*

```
import java.io.*;
import java.util.*;
public class TextFileO {
     public static void main(String[] args) {
        try {
              File fw = new File(args[0] + ".txt");
              PrintWriter out = new PrintWriter(fw);
              Scanner in = new Scanner(System.in);
              while (in.hasNextInt()) {
                      out.printf("%d", in.nextInt());
                      out.println();
              }
              out.close();
              System.out.println("inputs written into file successfully!");
              } catch (FileNotFoundException e) {
                      System.out.println("The file not found.");
              }
     }
}
```

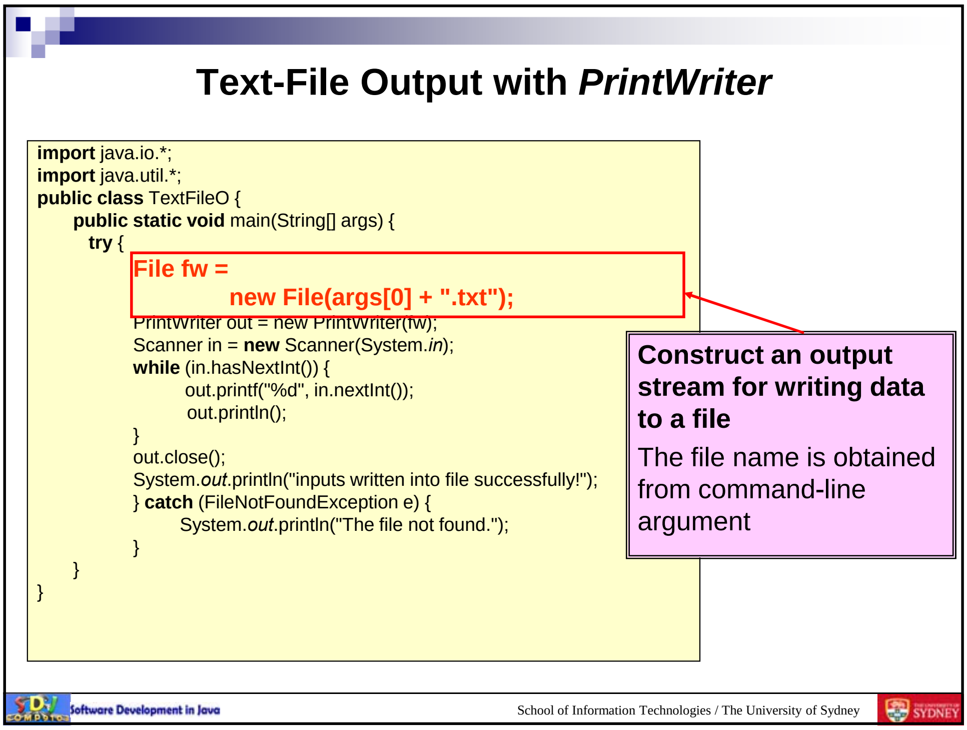
Terminal interaction

```
> java TextFileO fileout
1 2 3 4 5 6 q
```

output: fileout.txt

```
1
2
3
4
5
6
```

You can read this file with a file editor

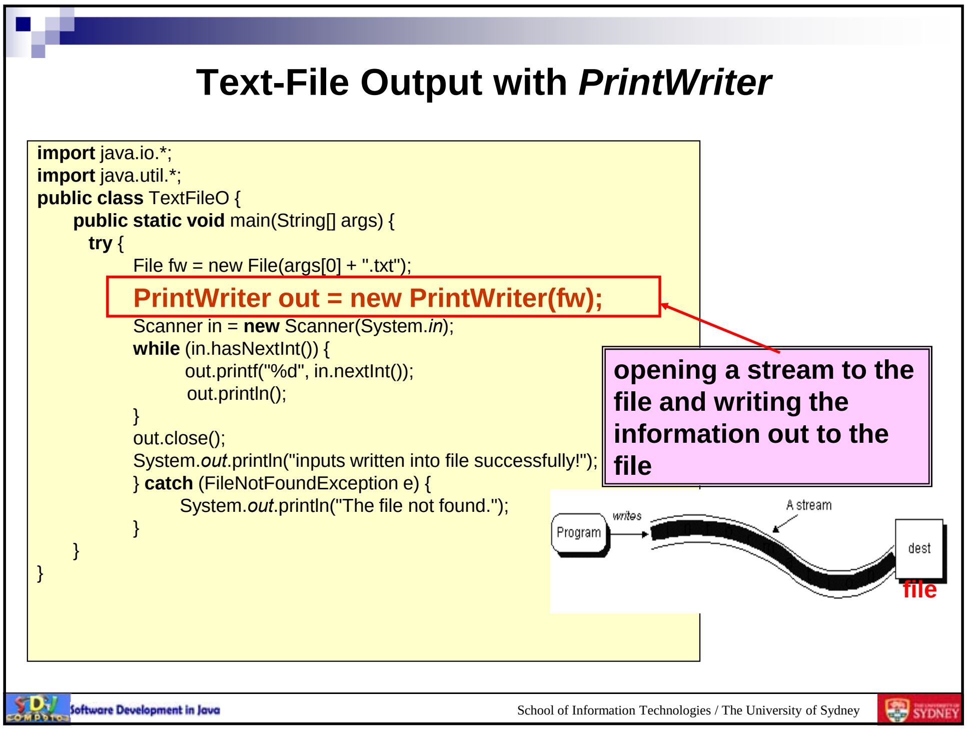


Text-File Output with *PrintWriter*

```
import java.io.*;
import java.util.*;
public class TextFileO {
      public static void main(String[] args) {
         try {
                File fw =
                         new File(args[0] + ".txt");
                PrintWriter out = new PrintWriter(fw);
                Scanner in = new Scanner(System.in);
                while (in.hasNextInt()) {
                         out.printf("%d", in.nextInt());
                         out.println();
                }
                out.close();
                System.out.println("inputs written into file successfully!");
                } catch (FileNotFoundException e) {
                        System.out.println("The file not found.");
                }
      }
}
```

Construct an output stream for writing data to a file

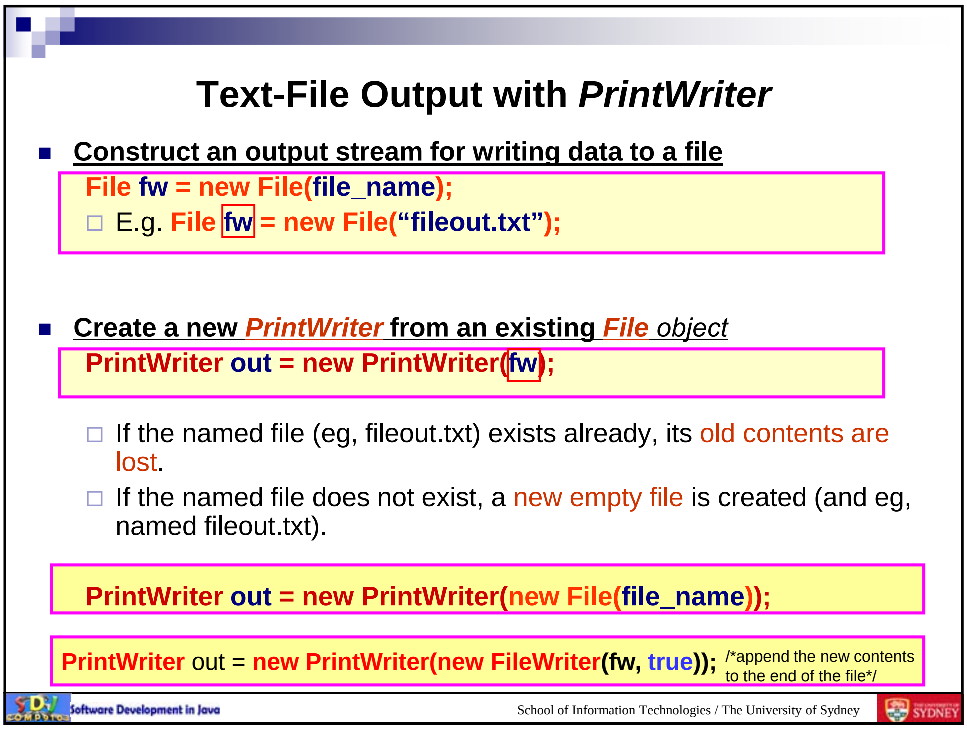
The file name is obtained from command-line argument



Text-File Output with *PrintWriter*

```
import java.io.*;
import java.util.*;
public class TextFileO {
      public static void main(String[] args) {
         try {
                File fw = new File(args[0] + ".txt");
                PrintWriter out = new PrintWriter(fw);
                Scanner in = new Scanner(System.in);
                while (in.hasNextInt()) {
                         out.printf("%d", in.nextInt());
                         out.println();
                }
                out.close();
                System.out.println("inputs written into file successfully!");
                } catch (FileNotFoundException e) {
                        System.out.println("The file not found.");
                }
      }
}
```

opening a stream to the file and writing the information out to the file



Text-File Output with *PrintWriter*

- Construct an output stream for writing data to a file

  File fw = new File(file_name);

  □ E.g. File fw = new File(“fileout.txt”);

- Create a new *PrintWriter* from an existing *File object*

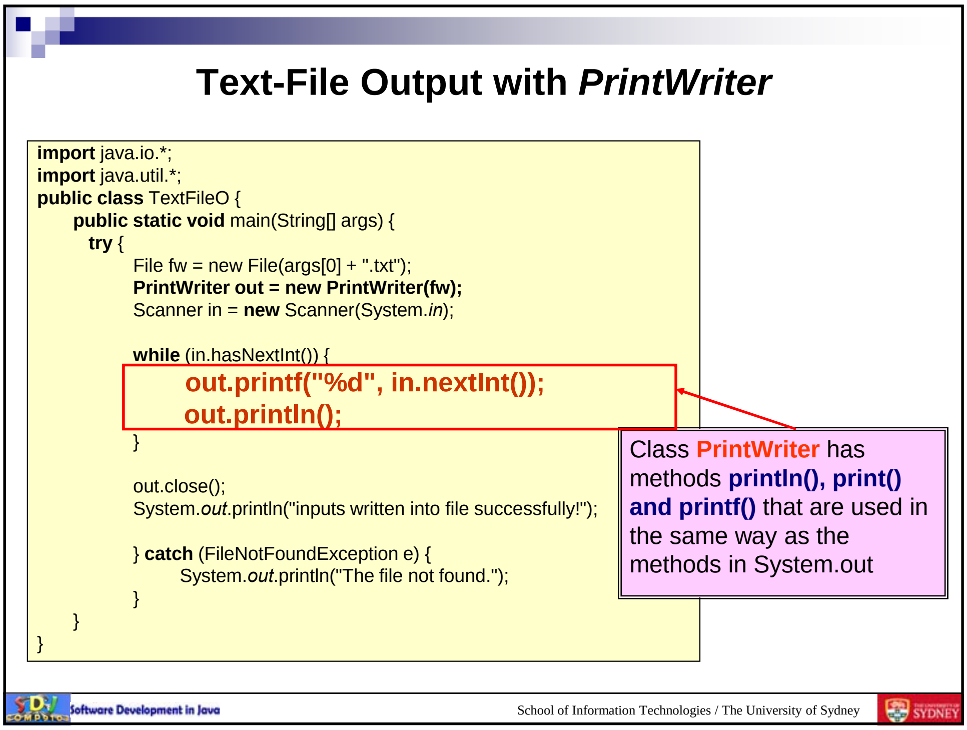
  PrintWriter out = new PrintWriter(fw);

  □ If the named file (eg, fileout.txt) exists already, its old contents are lost.

  □ If the named file does not exist, a new empty file is created (and eg, named fileout.txt).

PrintWriter out = new PrintWriter(new File(file_name));

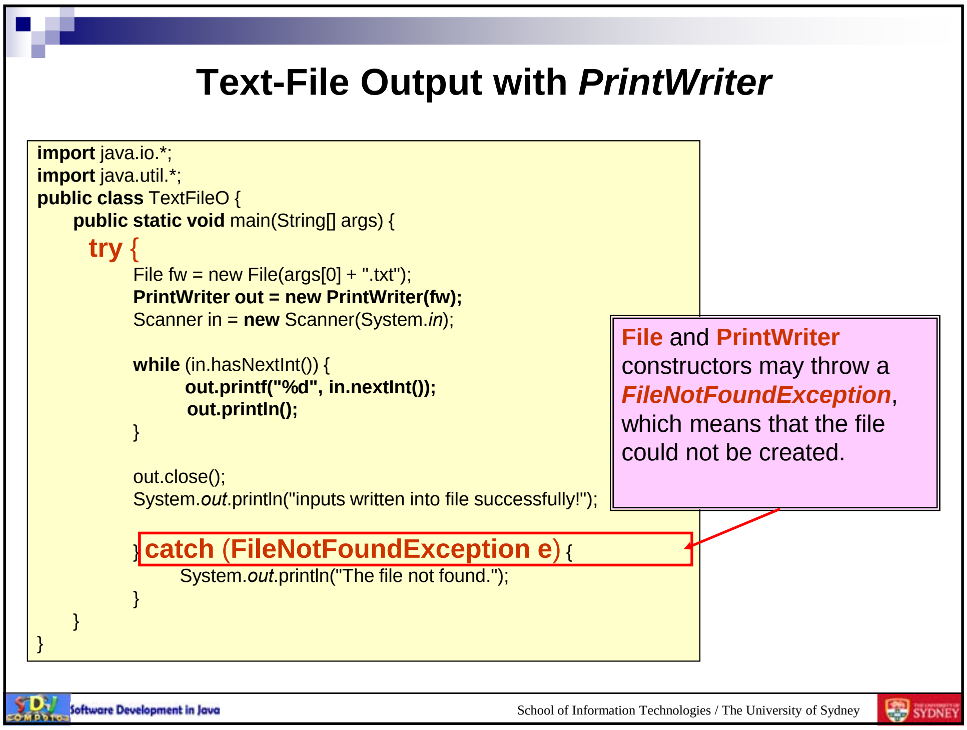
PrintWriter out = new PrintWriter(new FileWriter(fw, true)); /*append the new contents to the end of the file*/



Text-File Output with *PrintWriter*

```
import java.io.*;
import java.util.*;
public class TextFileO {
      public static void main(String[] args) {
         try {
                 File fw = new File(args[0] + ".txt");
                 PrintWriter out = new PrintWriter(fw);
                 Scanner in = new Scanner(System.in);

                 while (in.hasNextInt()) {
                        out.printf("%d", in.nextInt());
                        out.println();
                 }

                 out.close();
                 System.out.println("inputs written into file successfully!");

                 } catch (FileNotFoundException e) {
                        System.out.println("The file not found.");
                 }
      }
}
```

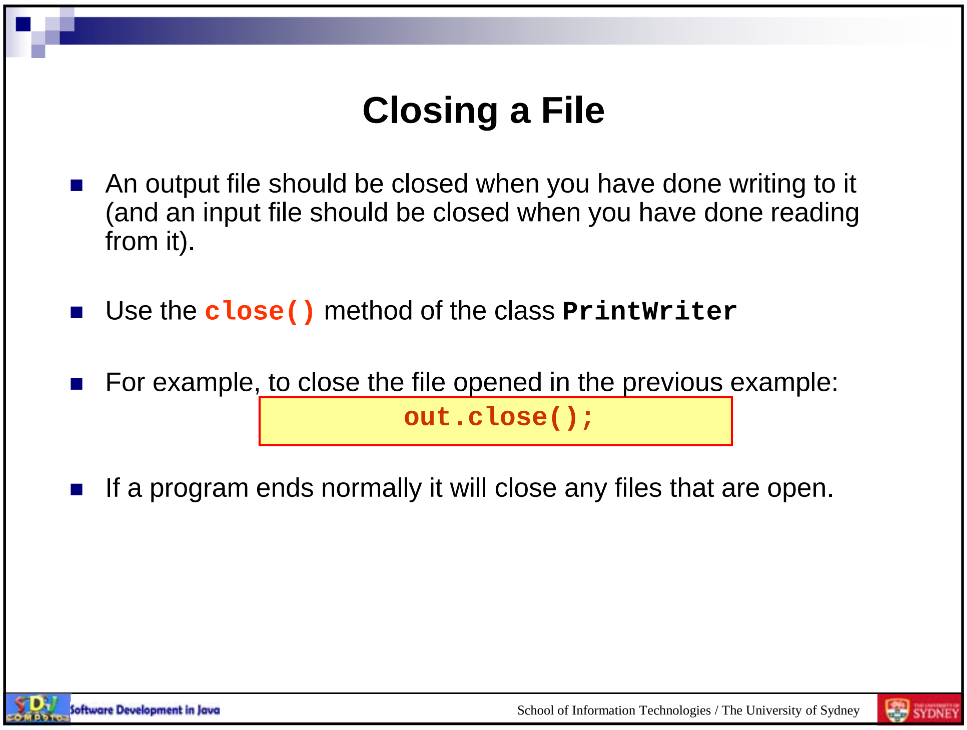
Class PrintWriter has methods println(), print() and printf() that are used in the same way as the methods in System.out



Text-File Output with *PrintWriter*

```
import java.io.*;
import java.util.*;
public class TextFileO {
      public static void main(String[] args) {
        try {
              File fw = new File(args[0] + ".txt");
              PrintWriter out = new PrintWriter(fw);
              Scanner in = new Scanner(System.in);

              while (in.hasNextInt()) {
                     out.printf("%d", in.nextInt());
                     out.println();
              }

              out.close();
              System.out.println("inputs written into file successfully!");

              } catch (FileNotFoundException e) {
                     System.out.println("The file not found.");
              }
      }
}
```

File and PrintWriter constructors may throw a *FileNotFoundException*, which means that the file could not be created.



Closing a File

- An output file should be closed when you have done writing to it (and an input file should be closed when you have done reading from it).
- Use the **`close()`** method of the class **`PrintWriter`**
- For example, to close the file opened in the previous example:

```
out.close();
```

- If a program ends normally it will close any files that are open.