CS2312 Problem Solving and Programming

2020-2021 Semester B

Department of Computer Science, City University of Hong Kong

Instructor: Dr. Helena WONG

Your Attention, Please!

- Not a pure programming course.
- Your first programming course and your coming software design coursehas a large gap in terms of the level of abstraction required.
- This course is to help you to raise the level of abstraction from pure programming to a logical organization of software code based on the requirements of the targeted applications to be developed.

[Borrowed from Dr. Ricky CHAN's notes in Spring 2013]

Related Courses:
CS2310 Computer Programming
CS2312 Problem Solving and Programming
CS3342 Software Design
CS3343 Software Engineering Practice

Java Programming and OO

- [Teaching Focus #1] Java Programming
  - Crash introduction of basics

    you have learnt C++ already, we can move fast onto java
  - Intensive study of key and advanced techniques

    target: pave the way for Part 2
- [Teaching Focus #2] Doing the OO
  - Object Oriented - concepts/design/principles/practices

Intended Learning Outcomes - Briefly:

1. [OO] Understand OO concepts
2. [OOD] Design OO solutions
3. [OOP] Implement the OO solutions in Java
4. [Practices] Apply the best practices in Java programming
5. [Review] Evaluate and review OO design and code

Python and Functional Programming

- [Teaching Focus #3]

  Python and Functional Programming

  - Given in week 12-13

Textbook and Materials

- [Focus #1] Java Programming

  Textbook: C.S. Horstmann, and G. Cornell, **Core Java™ Volume I**, Prentice Hall.

  Other books on my desk:

  - Walter Savitch, **Absolute Java**, Addison-Wesley.
  - Y. D. Liang, **Intro. to Java™ Programming Comprehensive Version**, Pearson.

  Official site of Java, tutorial: http://docs.oracle.com/javase/tutorial/index.html

- [Focus #2] OO concepts/design/principles/practices
  - Materials from Dr. Sam NG for his teaching of a previous course: CS2332 OOP in C++
    Sam is also the author of the current syllabus of CS2312.
  - Materials from Dr. Ricky CHAN [CS2312 / Spring 2013, CS3342], Dr. Jacky KEUNG [CS3342]
  - More.. [Check out at our courseweb]

- Acknowledgments:

  "Some of the material for this course was influenced by and, in some cases, directly borrowed from, materials available on the web for similar courses at other universities. I thank the instructors who posted their materials on the web." [Borrowed from http://www.cse.ohio-state.edu/~neelam/courses/45923/]

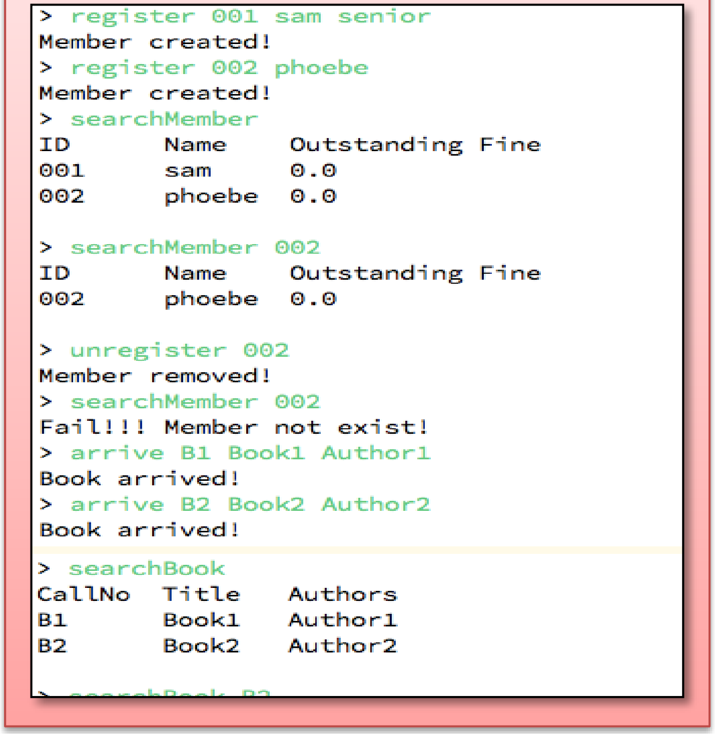
Sample OO Program

Consider a *Library System* which allows:

- Register a new member. A member may be a child, adult or senior.
- Cancel, search for an existing member.
- Add a new book.
- Remove the record of a book.
- Search for the details of a book.
- A member borrows / returns a book.
- A member pays fine. Fine rate is $3/day for children, $10/day for adult and $5/day for senior.
- Undo the last action performed by the user.

Procedural approach and **OO approach** are very different!!
Which would be our approach for even *larger* problems?

Sample rundown:



```
> register 001 sam senior
Member created!
> register 002 phoebe
Member created!
> searchMember
ID        Name      Outstanding Fine
001       sam       0.0
002       phoebe    0.0

> searchMember 002
ID        Name      Outstanding Fine
002       phoebe    0.0

> unregister 002
Member removed!
> searchMember 002
Fail!!! Member not exist!
> arrive B1 Book1 Author1
Book arrived!
> arrive B2 Book2 Author2
Book arrived!

> searchBook
CallNo    Title     Authors
B1        Book1     Author1
B2        Book2     Author2

> searchBook B2
```

CS2310 [**Procedural approach**]: Specify **what** tasks to do in each step
CS2312 [**Object-oriented approach**]: Specify **who** performs **what tasks** in each step.

> "Object-oriented design has been widely adopted by businesses around the world. When done properly, the approach leads to simpler, concrete, robust, flexible and modular software. " -- Robert C. Martin *(Uncle Bob)*

Contents **Topic 01 - Java Fundamentals**

I. Introducing JAVA

- The White Paper for Java was announced in May 1996
  James Gosling , Henry McGilton - Sun engineers

https://www.oracle.com/java/moved-by-java/

- Java is designed to achieve:
  - Simple
    Java is partially modeled on C++, but simplified and improved.
  - Object oriented
    Java was designed from the start to be object-oriented.
  - Distributed
    Java is designed to make distributed computing easy with networking capability. Writing network programs is like sending and receiving data to and from a file.
  - Multithreaded
    Multithread programming is smoothly integrated.
  - Dynamic
    Designed to adapt to an evolving environment. Libraries can freely add new methods and instance variables without effecting clients. Straightforward to find out runtime type information.
  - Architecture neutral, Portable
    With a Java Virtual Machine (JVM), one program can run on any platform without being recompiled.
  - High performance
    High performance of interpreted bytecodes, efficient translation of bytecodes to machine code.
  - Robust
    Java compiler, modified program constructs, runtime exception-handling
  - Secure
    Security mechanisms to protect against harm caused by stray programs.

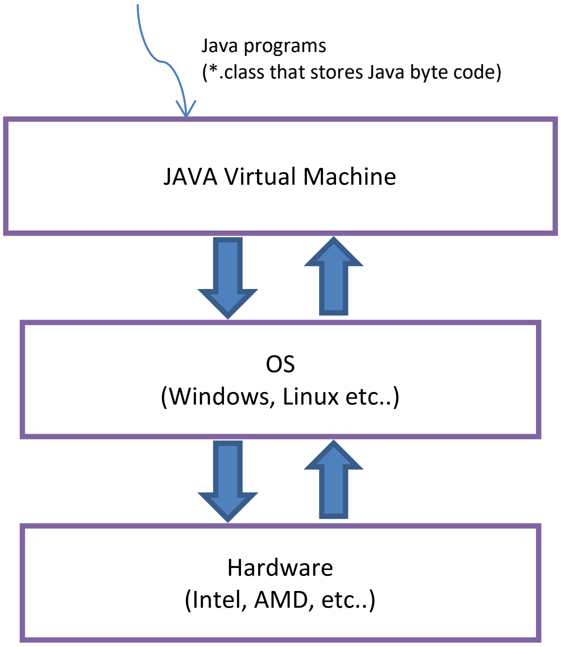
- The Java platform is available as different packages:

https://www.oracle.com/java/technologies/language-environment.html

  - JRE (Java Runtime Environment) – For consumers to run Java programs.
  - JDK (Java Development Kit) – For programmers to write Java programs.
    Includes JRE plus tools for developing, debugging, and monitoring Java applications.

- Once installed, the Java Virtual Machine (Java VM) is launched in the computer.

- During runtime, the Java VM interprets Java byte code and translates into OS calls.



- Java Versions:

  Version 1.0 (1995)
  Version 1.1 (1996)
  Version 1.2 (1998)
  Version 1.3 (2000)
  Version 1.4 (2002)

  Version 1.5 (2004) a. k. a. Java 5
  Version 1.6 (2006) a. k. a. Java 6
  Version 1.7 (2011) a. k. a. Java 7
  ..
  Version ??
  https://www.oracle.com/java/
  https://www.oracle.com/technetwork/java/java-se-support-roadmap.html

- Editions for different development purposes:

  - Java Standard Edition (J2SE)
    J2SE can be used to develop client-side standalone applications or applets.

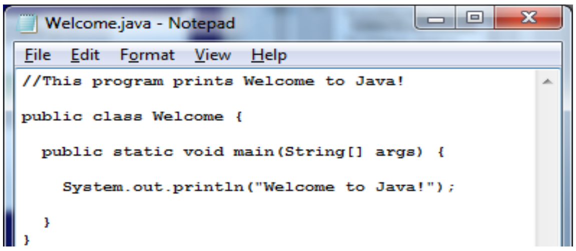
  - Java Enterprise Edition (J2EE)
    Server-side applications such as Java servlets, Java ServerPages, and Java ServerFaces.

  - Java Micro Edition (J2ME)
    Applications for mobile devices such as cell phones.

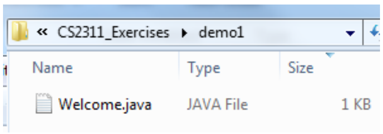
II. Compiling and Launching from Command-Line, IDE, A Simple JAVA Program

With JDK installed, you can compile and run Java programs in this way:

1. Create the source file: `Welcome.java`

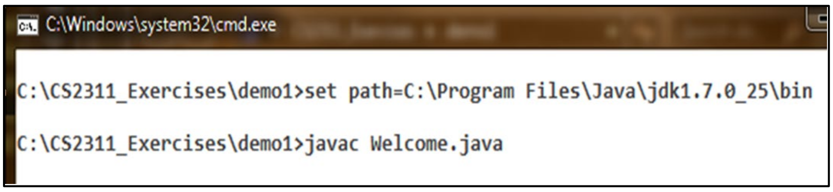


```
//This program prints Welcome to Java!

public class Welcome {

  public static void main(String[] args) {

    System.out.println("Welcome to Java!");

  }
}
```

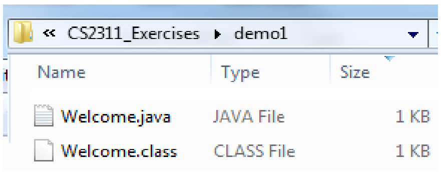


Name	Type	Size
Welcome.java	JAVA File	1 KB

2. At the command prompt, set path to JDK and then compile to give `Welcome.class`

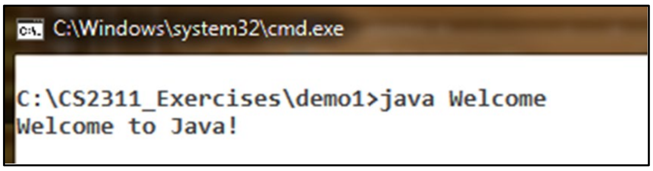


```
C:\CS2311_Exercises\demo1>set path=C:\Program Files\Java\jdk1.7.0_25\bin

C:\CS2311_Exercises\demo1>javac Welcome.java
```



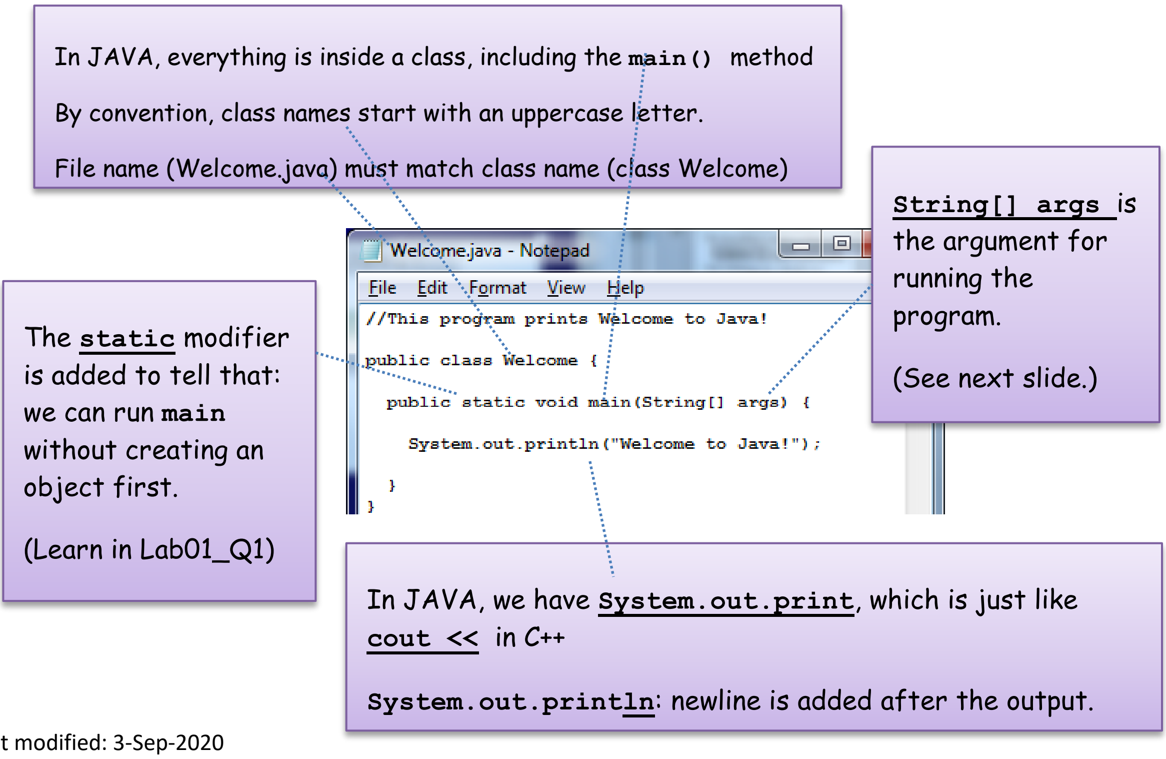
Name	Type	Size
Welcome.java	JAVA File	1 KB
Welcome.class	CLASS File	1 KB

3. Run it:



```
C:\CS2311_Exercises\demo1>java Welcome
Welcome to Java!
```

Explanation of the program:



In JAVA, everything is inside a class, including the `main()` method

By convention, class names start with an uppercase letter.

File name (Welcome.java) must match class name (class Welcome)

```
//This program prints Welcome to Java!

public class Welcome {

  public static void main(String[] args) {

    System.out.println("Welcome to Java!");

  }
}
```

<u>`String[] args`</u> is the argument for running the program.

(See next slide.)

The <u>`static`</u> modifier is added to tell that: we can run **`main`** without creating an object first.

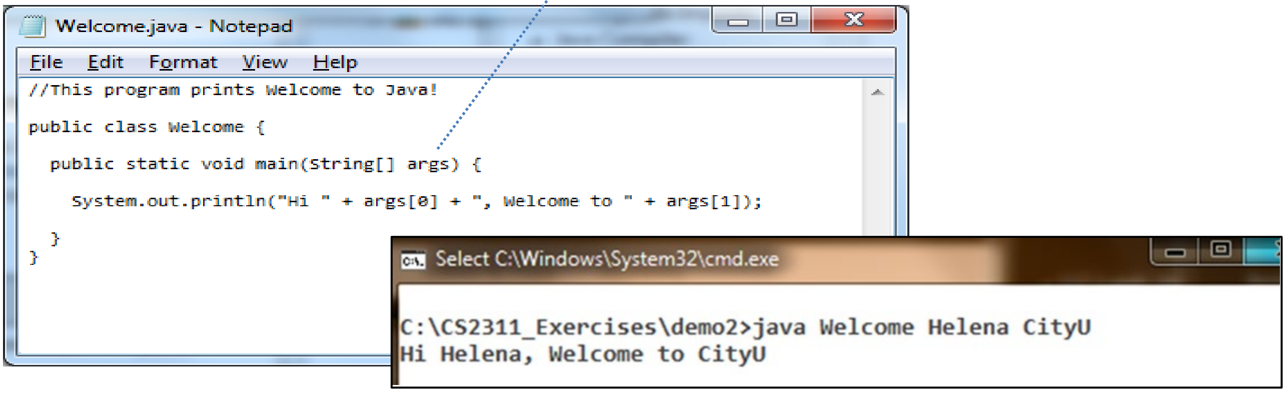
(Learn in Lab01_Q1)

In JAVA, we have <u>`System.out.print`</u>, which is just like <u>`cout <<`</u> in C++

`System.out.println`: newline is added after the output.

Arguments can be supplied to `main()` as an array of strings:

Example:

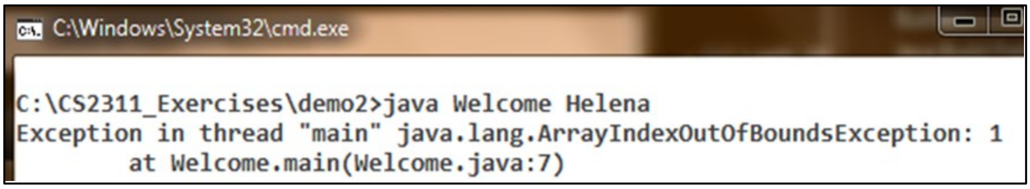

Welcome.java - Notepad
File Edit Format View Help
Select C:\Windows\System32\cmd.exe

```
//This program prints Welcome to Java!

public class Welcome {

  public static void main(String[] args) {

    System.out.println("Hi " + args[0] + ", Welcome to " + args[1]);

  }
}
```

```
C:\CS2311_Exercises\demo2>java Welcome Helena CityU
Hi Helena, Welcome to CityU
```

Run-time exception:

The program code expects 2 arguments. But the only one is given.



```
C:\CS2311_Exercises\demo2>java Welcome Helena
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
        at Welcome.main(Welcome.java:7)
```

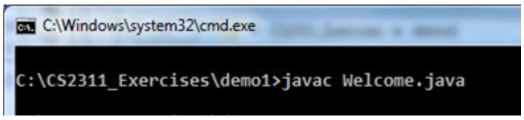
- Integrated Development Environments (IDE):

- NetBeans
- Eclipse
- **repl.it**
- **Vs Code**

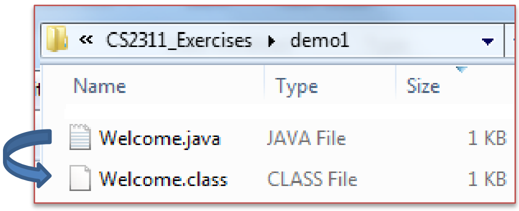
III. How does JAVA work

Compiling and Running Programs

- Source files (.java) are compiled into .class files by the javac compiler.

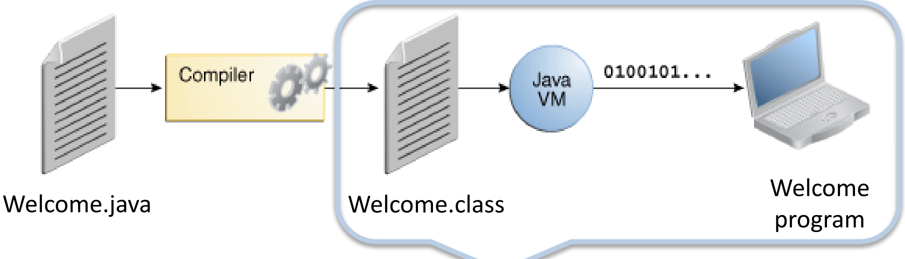

C:\Windows\system32\cmd.exe

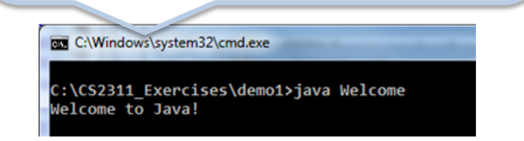
```
C:\CS2311_Exercises\demo1>javac Welcome.java
```


« CS2311_Exercises ▸ demo1

Name	Type	Size
Welcome.java	JAVA File	1 KB
Welcome.class	CLASS File	1 KB

- A .class file does not contain code that is native to the computer; It contains **bytecodes** — in machine language of Java Virtual Machine (Java VM).
- The *JRE* runs .class with an instance of the Java Virtual Machine.



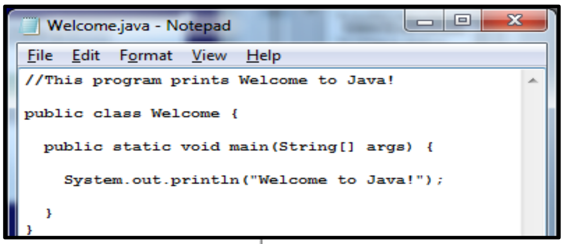

C:\Windows\system32\cmd.exe

```
C:\CS2311_Exercises\demo1>java Welcome
Welcome to Java!
```

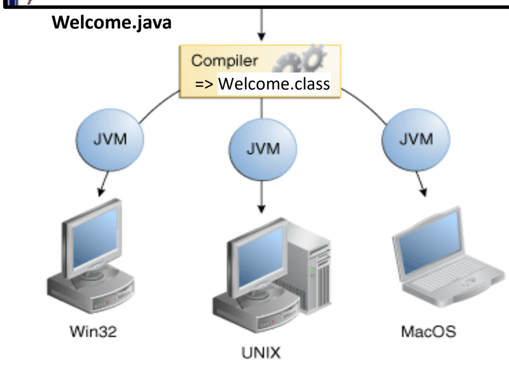
How are JAVA Programs "Architecture neutral", "Portable" ?

The role of Java VM

- Java VM is available on many different operating systems.
- Once you install JRE or JDK, Java VM is ready in your computer.
- The **same .class file** is capable of running on Microsoft Windows, the Solaris™, Linux, or Mac OS.



```
//This program prints Welcome to Java!

public class Welcome {

  public static void main(String[] args) {

    System.out.println("Welcome to Java!");

  }
}
```



IV. Review - Programming Style, Documentation, Syntax error / Runtime error / Logic error

Programming Style and Documentation

- Appropriate Comments
- Naming Conventions
  - Choose meaningful and descriptive names.
- Proper Indentation and Spacing Lines
  - Tabs, tidy spacing
  - Use blank line to separate segments of the code.
- Block Styles

```
public class Day {
  private int year;
  private int month;
  private int day;
  public Day(int y, int m, int d) {
    this.year = y;
    this.month = m;
    this.day = d;
  }
  public String toString() {
    return day + "-" + month + "-" + year;
  }
}
```

Poor! Hard to read!
Please add line breaks before methods

Next-line style
(OK)

```
public class Test
{
  public static void main(String[] args)
  {
    System.out.println("Block Styles");
  }
}
```

End-of-line style
(OK)

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Block Styles");
  }
}
```

Three types of programming errors

- Syntax Errors
  - Detected by the compiler

```
public class ShowSyntaxErrors {
  public static main(String[] args) {
    System.out.println("Welcome to Java);
  }
}
```

- Runtime Errors
  - Causes the program to abort

```
public class ShowRuntimeErrors {
  public static void main(String[] args) {
    System.out.println(1 / 0);
  }
}
```

- Logic Errors
  - Produces incorrect result

```
public class ShowLogicErrors {
  public static void main(String[] args) {
     System.out.print("Five plus six is ");
     System.out.println("5"+"6");
  }
}
```

Debugging

(1) A video on Canvas => CS2312 => https://www.cs.cityu.edu.hk/~helena/cs231220... :

debugger in VS Code (Tracing Lec01 Q12 Fib and Lab01 Q02 Day)

(2) https://code.visualstudio.com/docs/java/java-debugging

Contents Topic 02 - Java Fundamentals

I. A Simple Java Program

FirstSample.java

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

Access Modifier: **public**

- The keyword ***public*** is one of the **access modifiers**, that states how other parts can get the access. For example, here *public* is applied to *class FirstSample* meaning that outsiders can use the class.
- Except *public*, we also have other modifiers: *private*, *protected*, etc., We will cover them in next topic.

JAVA classes and .java files

- **Classes** are the building blocks for Java programs.
  By convention: start with an uppercase letter
- A .java file cannot have 2 or more public classes.
  The name of a public class has to be the same as the file name
- Case sensitive. `FirstSample.java` matches with class name `FirstSample`

The **main** method in Java

- **main()** does not return anything, thus **void**
- **main()** has to be inside a class; as a **static** method (ie. A method that does not operate on objects.).

String[] args

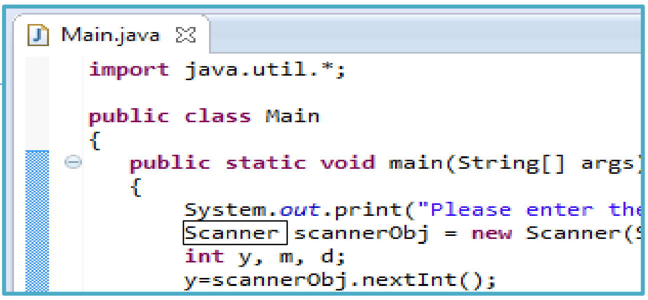
- Arguments can be supplied to **main()** as an array of strings. (Ref. Topic01)

`print(..), println(..)`

- General syntax to invoke a method: `object.method(parameters)`
- Use the `System.out` object and call its `println` method.
  - **`System.out.println("Hello!");`**
    prints "Hello!" + terminate the line (newline character '\n')
- Can be without argument: simply a blank line
  - **`System.out.println();`**
- `.print()` method
  - **`System.out.print("Hello!");`** ← not terminating the line (✗ "\n")

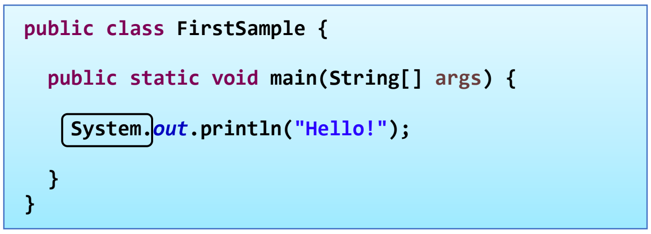
II. Packages and the import statement

- **`Packages`** are groups of classes.
- The standard Java library is distributed over a number of packages
  e.g. **`java.lang, java.util, ..`**
- **`java.util`**
  - **`java.util`** contains the **`Scanner`** class (and many other classes)
  - We can use it like: **`java.util.Scanner`**
  - If we import **`java.util.*`**, then we can simply write: **`Scanner`**
- **The `import` statement:**
  - `import java.util.*;`
    We can use all classes in `java.util`
  - `import java.util.Scanner;`
    We can use Scanner



```
import java.util.*;

public class Main
{
    public static void main(String[] args)
    {
        System.out.print("Please enter the
        Scanner scannerObj = new Scanner(S
        int y, m, d;
        y=scannerObj.nextInt();
```

- **`java.lang` - Java fundamental classes**
  - No need to import `java.lang` (assumed already for all programs)
  - `java.lang` provides the `System` class , `Math` class, and many more..
- **Use of the `java.lang.System` class:**

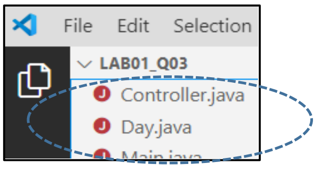


```
public class FirstSample {

  public static void main(String[] args) {

    System.out.println("Hello!");

  }
}
```

Ref: JAVA API Documentation https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html

III. Creating Packages and "Default Package"

- We can group our source files into **packages**
  - A class not grouped into package is in "default package"

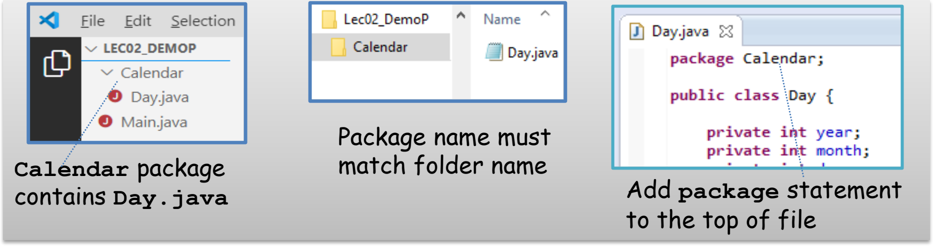


  - A class grouped into package is in the package folder and has the package statement:

Project folder in VS Code

File explorer

Source code



```
package Calendar;

public class Day {

    private int year;
    private int month;
```

`Calendar` package contains `Day.java`

Package name must match folder name

Add `package` statement to the top of file

To use the class, type `Calendar.Day` or add `import Calendar.*`

- Unluckily PASS doesn't compile packages at this moment.

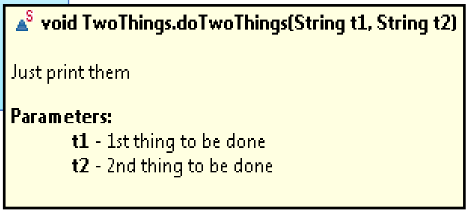
IV. Comments

Three ways of marking Comments

- Like **C++: //, /* . . . */**
- **For automatic documentation generation: /** . . . */**

```
/**
   Just print them
   @param t1 - 1st thing to be done
   @param t2 - 2nd thing to be done
 */
static void doTwoThings(String t1, String t2)
{
    System.out.println(t1);
    System.out.println(t2);
}
```

automatic documentation generation (by the javadoc program from JDK)



void TwoThings.doTwoThings(String t1, String t2)

Just print them

Parameters:
  t1 - 1st thing to be done
  t2 - 2nd thing to be done

Some IDEs add * in front of every line, for visual style only.

```
/**
 * Just print them
 * @param t1 - 1st thing to be done
 * @param t2 - 2nd thing to be done
 */
```

V. Data Types and Variables

Overview of Data types

There are two kinds of types in the Java: **Primitive types** and **Reference types**

I. **Primitive types**: Java has 8 primitive types: `boolean, byte, short, int, long, float, double, char`

II. **Reference types** (~ pointers in C++) :

- The values of a reference type are references to objects.
  An *object* is an *instance* of a class.
  Note: The word "object" is often used interchangeably with "instance".
- Examples of built-in Java classes: `String, Math, Scanner`
  Examples of user-defined classes (Lab01): `Main, Day, Model, View, Controller`

  We can create objects of these classes, using the **new** operator. After creation, we get the **object reference**.

  We often use a variable to hold the object reference;
  Then we can use the variable to access the object.

  e.g. Day dayObj = new Day(2013, 12, 31);
  System.out.println(dayObj.toString());
- An **array** is also a special kind of object

Variables

We have seen that there are 2 types of data: Primitive and Reference types.

These data can be stored in variables: primitive values and reference values:

Types of variables in JAVA	Information stored	Examples

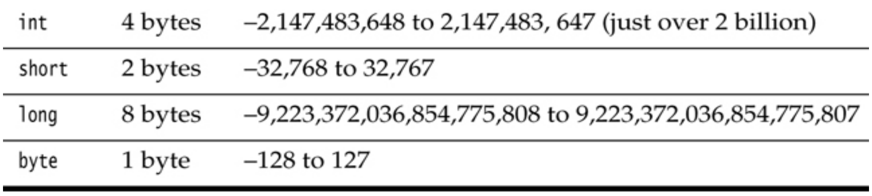
(i) Variables of Primitive Types	The variables hold the exact values	`int x;`
		`x = 689;`

(ii) Variables of Reference Types	The variables hold the references to **objects** (Like pointers in C++)	`Day d;` `d = new Day(2016,1,20);`[#1]

#1: Use of a class: We use a class name as variable type, and use the class to create ("new") an object of its kind.

Data Types – Integers

- Integer types



int	4 bytes	–2,147,483,648 to 2,147,483, 647 (just over 2 billion)
short	2 bytes	–32,768 to 32,767
long	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
byte	1 byte	–128 to 127

- Java has no unsigned types
- Long integer numbers have a suffix L

```
//suffix 'L' is required for 2,223,123,123
System.out.println("Testing: " + 2223123123L); //Testing: 2223123123
long x;
x = 2223123123L;
System.out.println("x is: " + x); //x is: 2223123123
```

- To provide numbers in Binary, we need prefix: 0b
- To provide numbers in Hexadecimal, we need prefix: 0x

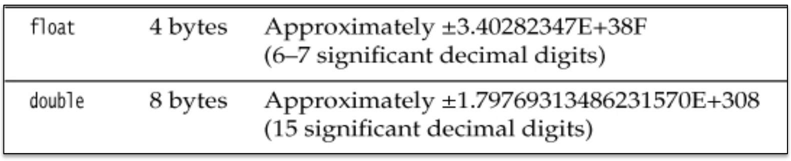
```
int x;
x = 0b11111111;
System.out.print(x); //shows 255

x = 0xFF;
System.out.print(x); //shows 255
```

Data Types – Floating Point Types

- Floating Point Types
  - For numbers with fractional parts, (ie. not whole numbers, e.g. 1.1), and
  - For very large numbers (e.g. 5 x 10^{23})

- The precision is limited (ie. keep a few significant digits).

- 2 Types:



float	4 bytes	Approximately ±3.40282347E+38F (6–7 significant decimal digits)
double	8 bytes	Approximately ±1.79769313486231570E+308 (15 significant decimal digits)

- Suffix:
  - float - F
  - double – D (optional)

```
float x1 = 0.98765987659876598765F;
System.out.println(x1); //Shows 0.9876599

double x2=0.98765987659876598765D; //'D' is optional
System.out.println(x2); //Shows 0.987659876598766

double x3=0.98765987659876598765; //'D' is optional
System.out.println(x3); //Shows 0.987659876598766

double x4=98765987659876598765D; //'D' is optional
System.out.println(x4); //Shows 9.876598765987666E19
```

- Roundoff errors

```
System.out.println(2.0 - 1.1);   //0.8999999999999999
```

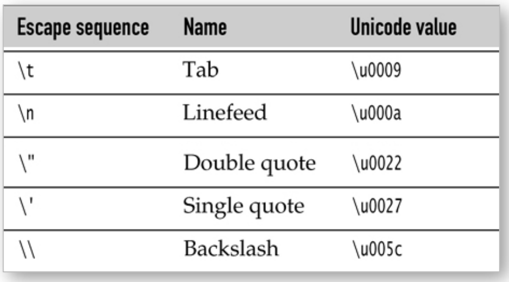
Reason of *Roundoff errors*:
computer uses binary number system; and there is no precise binary representation of a lot of fractions (e.g. 1/10)

Solution: use the `BigDecimal` class (Example is in the given code of this topic)

Data Types – char and Escape Sequence

- Primitive type for characters: `char`

- Escape Sequence for special char values:



Escape sequence	Name	Unicode value
\t	Tab	\u0009
\n	Linefeed	\u000a
\"	Double quote	\u0022
\'	Single quote	\u0027
\\	Backslash	\u005c

Data Types – Boolean

- Boolean type: `true, false`
  - We cannot convert between integers and Boolean values

```
boolean b=true;
int i = 3;
b = i; //Error!! Type mismatch - cannot convert from int to boolean
i = b; //Error!! Type mismatch - cannot convert from boolean to int
```

Declaration of Variables

- Every variable has a type:
  - `double salary;`
  - `int vacationDays;`
  - `long earthPopulation;`
  - `boolean done;`

- Common ways to name: start with lowercase letter
  - `Box box; //Box is a class type and box is the variable.`
  - `Box aBox; //using "a" as prefix`
  - `Box bxJewels, bxCoins;`

- Must explicit initialize before use

```
int x;
System.out.println(x); // ERROR--variable not initialized
```

```
int x = 12;
System.out.println(x);  //12
```

```
int x;
x= 28;
System.out.println(x);  //28
```

VI. Constants

- Constants: Java keyword is `final`
- **`final`:** the value is set once and for all.
- Common ways to name a constant: all in uppercase
- Often given as **Method constants** or **Class constants**

```
public class MyApp
{
   public static void main(String[] args)
   {
      final double CM_PER_INCH = 2.54;
      ..
   }
}
```

```
public class MyApp
{
    public static final double CM_PER_INCH = 2.54;
    public static void main(String[] args)
    {
       ..
    }
}
```

VII. Operators: Arithmetic Operators, Relational Operators, Boolean (logical) Operators, Bitwise Operators

(I) Arithmetic Operators: +, -, *, /, %, +=, -=, *=, /=, %=

x/y

- If both x and y are integers, denotes **integer division**, e.g. `3/4` is `0`
- Otherwise **floating-point division**, eg. `3.0/4` is `0.75`

x/0

- If x is integer => **division by zero exception** (run-time error)
- Otherwise, ie. x is floating point => `NaN` Not a number or `Infinity`

```
System.out.println(0/0);    ➔ java.lang.ArithmeticException: / by zero
System.out.println(0%0);    ➔ java.lang.ArithmeticException: / by zero
System.out.println(12.0/0); ➔ gives Infinity
System.out.println(0.0/0);  ➔ gives NaN
System.out.println(12.0%0); ➔ gives NaN
System.out.println(0.0%0);  ➔ gives NaN
```

(II) Relational Operators: ==, !=, >, <, >=, <=

(III) Boolean (logical) operators: &&AND, ||OR

- Evaluated in "Short circuit" fashion

  I.e., The second argument is not evaluated if first argument already determines the value.

Example 1:

```
if ((isMember==true) || (calculateAge(..)>=65))
    System.out.print("Gift");
```

if `isMember` is true, then `calculateAge` doesn't need to (will not) run.

Example 2:

```
if (totCourses>0 && totalMarks/totCourses >=90)
    System.out.print("Well done!!");
```

if `totCourse` is `0`, then `totalMark/totCourses` will not be calculated (avoid run-time error "Division-by-zero")

(IV) Bitwise Operators:

`& ("AND")` `| ("OR")` `^ ("XOR")` `~ ("NOT")` `<< (left-shift)` `>> (right-shift)`

```
Example:
  int n1, n2, n3;
  n1 = 0xFE; (ie. 0b11111110)
  n2 = n1 ^ 0xFF; //set n2 to 0b00000001 (0b11111110 XOR 0b11111111)
  n3 = n2 << 4; //set n3 to 0b00010000 (left-shift 0b00000001 by 4 bits)
```

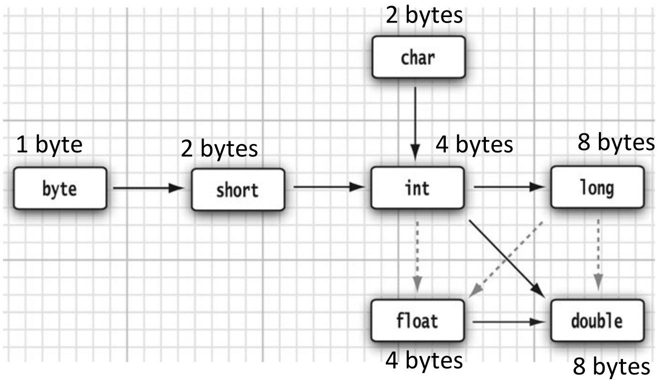
VIII. Conversion between numbers, type casting

Legal conversions:

Case 1: Without information loss, or →

Case 2: Just to lose precision ⇢

```
double d; float f; int i=2147483647;
d=i;
f=i;
System.out.println(f); //output: 2.14748365E9
System.out.println(d); //output: 2.147483647E9
System.out.println(i); //output: 2147483647
```



Case 3: Though legal, but may lose information. For these cases explicit **type casting** is needed:

```
int i=97; char c;
c=(char)i;
System.out.println(i); //output: 97
System.out.println(c); //output: a
```

Q: Why int->char may cause information lost?
A: int is 4 bytes, can hold big range of values
But char is 2 bytes only

IX. Parentheses and Operator Hierarchy

When one expression contains 2 or more operators, then

The order of Evaluation depends on precedence level and associativity.

E.g.

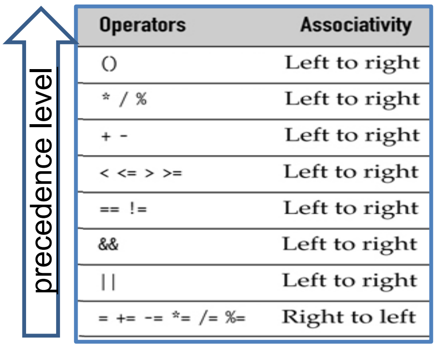
```
int x = 12345 / (4 + 5 * 7 - 2);
              ④    ②   ①   ③
```

- The precedence level of / and * are higher than the precedence level of + and -

- To override the above ordering, we add () for grouping.

↑

the precedence level of () is high


precedence level

Operators	Associativity
()	Left to right
* / %	Left to right
+ -	Left to right
< <= > >=	Left to right
== !=	Left to right
&&	Left to right
||	Left to right
= += -= *= /= %=	Right to left

Exercise:

Q1. For each underlined expression below, mark the steps with ①, ②:

(i). `System.out.println(1234 / 100 / 10) ;`

(ii) `System.out.println(1234 * 60 % 24);` (note: * and % have the same precedence)

(iii). `int a,b; a = b = 10;`

Q2. Delete the wrong items below (*):

In * (i) / (ii) / (iii), we say that the associativity is left-to-right.

In * (i) / (ii) / (iii), we say that the associativity is right-to-left.

Note:
When operators have the same **precedence level**, then **the associativity rules** of the operators decide the order of evaluation.

X. Strings & StringBuilder

Strings (`java.lang.String`)

- A `String` object contains a sequence of Unicode characters (code units in UTF-16 encoding)

- String variables are references to string objects

```
String s = new String("Hello");
```

or

```
String s = "Hello";
```

← Shorthand

- **`.length`** method yields number of characters

- **`""`** is the empty string of length 0, different from `null`

```
if (s!= null && s.length() != 0)
```

← Check for non-null and non-empty

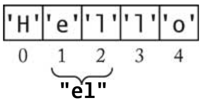
- **`.charAt`** method yields `char`:

```
char c = s.charAt(i);
```

- **`.substring`** method yields substrings:

```
String greeting = "Hello";
String s = greeting.substring(1,3);
```

It means from position 1 inclusive to position 3 **exclusive**



- +

```
String greeting = "Hello"; String s;
s = 1000 + " " + greeting; // "1000 Hello"
s = 1000 + ' ' + greeting; // ___________
```

- Use **`.equals`** to compare **string contents**:

```
String greeting = "Hello";
String part = greeting.substring(1, 3);
if (part == "el") {..} //NO!
if (part.equals("el")) {..} //OK
```

← == tests whether both refers to the same object (not compare contents)

- Converting Strings to Numbers: **Integer.parseInt**

```
String input = "7";
int n = Integer.parseInt(input); // n gets 7
```

- Strings are **immutable**:
  - No method can change a character in an existing string
  - To turn greeting from "Hello" to "Help!", it is not so convenient:

```
greeting = greeting.substring(0,3)+"p!";
```

← actually a new string object

For the original string object which was previously referred by greeting, Java has the *Garbage Collection* mechanism to recycle the unused memory.

StringBuilder (java.lang.StringBuilder)

```
StringBuilder sb = new StringBuilder();
sb.append("Hello ");
sb.append(name); //suppose name is "Peter"
String result=sb.toString(); //gives "Hello Peter"
```

- If more concatenation work is needed,
using + for string concatenation is inefficient ☹
reason: it actually creates new string objects

- **`StringBuilder`** object – can manipulate characters within itself. ☺

- Other `StringBuilder` methods for handling character contents: **`setCharAt, insert, delete`**

XI. Input (Console, File, Input from String)

(1) Reading input from Console

- Construct `Scanner` from input stream (e.g. `System.in`)
  `Scanner in = new Scanner(System.in);`

- `.nextInt, .nextDouble` reads next `int` or `double`
  `int n = in.nextInt();`

- `.next` reads next string (delimited by whitespace: space, tab, newline;
  discard leading whitespace)

- `.nextline` reads until `newline` and removes `newline` from the stream.
  Sometimes we need `.nextline` to remove extra line break (Learn from Lab03)

- `.close` closes the stream

```
Scanner in = new Scanner(System.in);
String s1,s2;
s1 = in.next(); //type "   Today     is a good day."
s2 = in.nextLine();
System.out.println(s1); //"Today"
System.out.println(s2); //"     is a good day“
in.close();
```

```
   Today     is a good day.
Today
     is a good day.
```

Rundown

(2) Reading input from a file

- Construct `Scanner` from a `File` object
  - `Scanner inFile = new Scanner(new File("c:\\data\\case1.txt"));`

- `.hasNext()` checks whether there is still “next string” in the file.

```
Scanner inFile = new Scanner(new File(fileName));

while (inFile.hasNext()) {
       String line = inFile.nextLine();
       ..
}
inFile.close();
```

(3) Reading input from another string

```
Scanner inData = new Scanner(str); //where str is a String
//.. apply .hasNext(), .next(), .close() etc..
```

Example: Read a of words and show them line by line:

```
System.out.print("Enter a line of words: ");

Scanner scannerConsole = new Scanner(System.in);
String str = scannerConsole.nextLine();

Scanner scannerStr = new Scanner(str);
while (scannerStr.hasNext())
       System.out.println(scannerStr.next());

scannerStr.close();
scannerConsole.close();
```

Output

```
Enter a line of words: Have a good day!
Have
a
good
day!
```

XII. Output (System.out.printf, String.format, PrinterWriter)

Formatted Output – using System.out.printf()

- Using **.print, .println** for floating-point values (problem):

```
double x = 10000.0 / 3.0;
System.out.println(x); //prints 3333.3333333333335
```

- Using **.printf** – formatted output (solution)

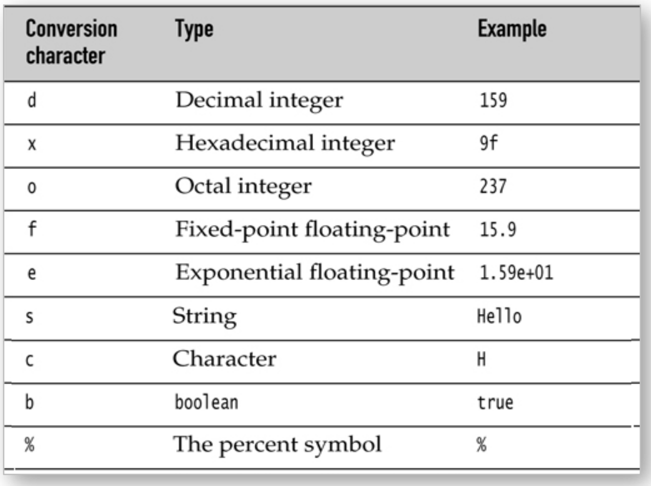
```
double x = 10000.0 / 3.0;
System.out.printf("%8.2f", x);//prints  3333.33
```

Field width of 8 characters
Precision of 2 characters
=> Result has a leading space and 7 characters

- Using **.printf** – multiple parameters

```
System.out.printf("Hi %s. Next year you'll be %d\n", name, (age+1));
```

- Conversion characters (%f, %d, %s):



Conversion character	Type	Example
d	Decimal integer	159
x	Hexadecimal integer	9f
o	Octal integer	237
f	Fixed-point floating-point	15.9
e	Exponential floating-point	1.59e+01
s	String	Hello
c	Character	H
b	boolean	true
%	The percent symbol	%

- Similar method to create a string

```
String msg = String.format(“Hi %s. Next year you'll be %d", name, age+1);
```

- Output to a file (Create a PrinterWriter object, then apply .print etc..)

```
PrintWriter out = new PrintWriter("c:\\report\\myfile.txt");
out.println("My GPA is 4.0");
out.close();
```

XIII. Control flow

- Control structures (similar to C++):
  if if..else switch-case while do-while for

- Block Scope (compound statement inside {})

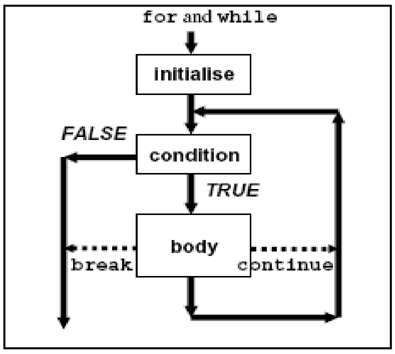
- Cannot declare identically named variables in 2 nested blocks:

```
public static void main(String[] args)
{
   int n;
   . . .
   {
     int k;
     int n; // ERROR--can't redefine n in inner block
     . . .
   }
}
```

- Declaring a variable in a for-loop:

```
for (int i = 1; i <= 10; i++)
{
  . . .
}
// i no longer defined here

for (int i = 11; i <= 20; i++) // OK to define another variable named i
{
  . . .
}
```

- Using break and continue:
  break
  means "immediately halt execution of this loop"
  Continue
  means "skip to next iteration of this loop."



We should use them only if that improves coding quality !

```
// Find x in an array A

bFound=false;
for (i=0;i<n;i++)
{
    if (A[i]==x)
    {
        bFound=true;
        break;
    }
}
```

Or

```
// No need to use break
bFound=false;
for (i=0;i<n && !bFound;i++)
{
      if (A[i]==x)
            bFound=true;
}
```

```
// read in 10 numbers
// and handle only the positive ones
for (i=0; i<10; i++)
{
    x=scannerObj.nextInt();
    if (x<0)
    {
        System.out.println("Wrong");
        continue;
    }

    .. // processing of x
}
```

XIV. Arrays

- An array is a collection of elements **of the same type**

- Index is zero-based

```
int[] arr; // int[] is the array type; arr is the array name
           // int arr[]; is also okay, but not welcome by Java fans

arr = new int[5]; //create the array;
arr[0] = 3;
arr[1] = 25;

for (int i=0;i<arr.length;i++) //use .length to tell the array size
    System.out.println(arr[i]);
```

Once created, cannot change size

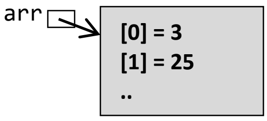
If extension is needed, make arr to refer to a new larger array and copy the old contents.

Output

```
3
25
0
0
0
```

Initialized values
For number elements : 0
For boolean elements: false;
For object elements: null

- Array variable is a reference



- Styles of array declaration:

(1)

```
int[] arr;
arr = new int[5];
arr[0] = 3;
arr[1] = 25;
```

(2)

```
int[] arr = new int[5];
arr[0] = 3;
arr[1] = 25;
```

(3)

```
int[] arr = {3,5,0,0,0};
```

Shorthand - Declaration with initializers

- Reinitializing **an array variable**

```
int[] arr = {3,5,0,0,0};
arr = new int[] {1,2,3,4,5,6,7,8};
```

← a new array

For the original array which was previously referred by arr, Java has a *Garbage Collection* mechanism to recycle the unused memory.

- **The "for each" loop:**
    - Syntax: for (variable : collection) statement
    - Example:

```
for (int x: arr)
    System.out.println(x);
```

← "for each" loop - goes through each element as **x**

Practice: Use more "for-each" loop from now on ! ☺

- **Arrays.toString**
    - Provided by the `java.util.Arrays` class
    - Returns a string representation of the array

```
int[] arr = {3,5,0,0,0};
System.out.println(Arrays.toString(arr));
```

Output

```
[3, 5, 0, 0, 0]
```

- Sorting with **Arrays.sort**

```
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
```

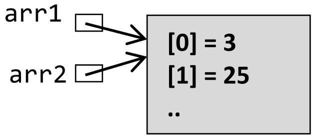
Output

```
[0, 0, 0, 3, 5]
```

- **Array copying**

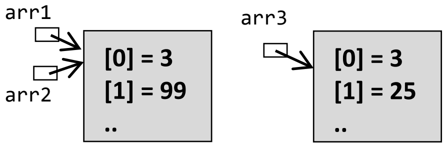
(1) Copying reference – not really creating new array

```
arr1 = new int[] {3,25,0,0,0};
arr2 = arr1;
```



(2) Copying as a new array: `Arrays.copyOf`
Syntax: `Arrays.copyOf(originalArray, newSize);`

```
int[] arr1, arr2, arr3, arr4, arr5;
arr1 = new int[] {3,25,0,0,0}; // 5 elements

arr2 = arr1;
arr3 = Arrays.copyOf(arr1, 4); // only want 4 elements

arr1[1]=99;

System.out.println(Arrays.toString(arr1));
System.out.println(Arrays.toString(arr2));
System.out.println(Arrays.toString(arr3));
```



Output

```
[3, 99, 0, 0, 0]
[3, 99, 0, 0, 0]
[3, 25, 0, 0]
```

`Arrays.copyOfRange` is another useful method. Learn it in Lab03.

- **Multidimensional array**

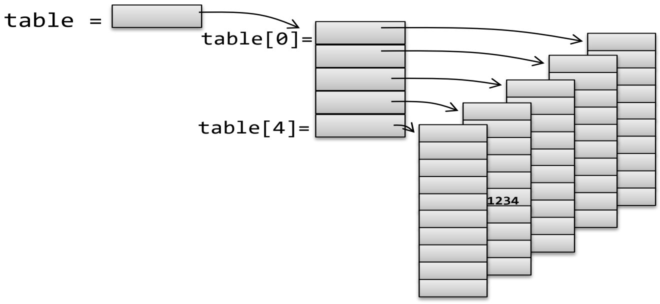
2D array:
- Is a 1D array of some 1D arrays

5 rows 10 columns

```
int[][] table = new int[5][10];
table[3][5]=1234; // set 4th row, 6th column to 1234
for (int[] arr1D: table)
   System.out.println(Arrays.toString(arr1D));
```

Output

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1234, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```



Each element in the 2D array is a 1D array !

Ragged Array:
- Different rows have different lengths
  It is easy to do so in Java.

```
String[][] helpers = {
        {"Helena", "Kit", "Jason"},
        {"Helena", "Kit", "Jason"},
        {"Kit", "Jason"},
        {"Helena", "Kit"},
        {"Helena"}
};

System.out.println("Helpers for T01-T05:");
System.out.println("====================");

for (String[] arr1D: helpers)
        System.out.println(Arrays.toString(arr1D));
```

Output

```
Helpers for T01-T05:
====================
[Helena, Kit, Jason]
[Helena, Kit, Jason]
[Kit, Jason]
[Helena, Kit]
[Helena]
```

O Contents Topic 03 - Objects and Classes

I. Classes and Objects – Casual Preview

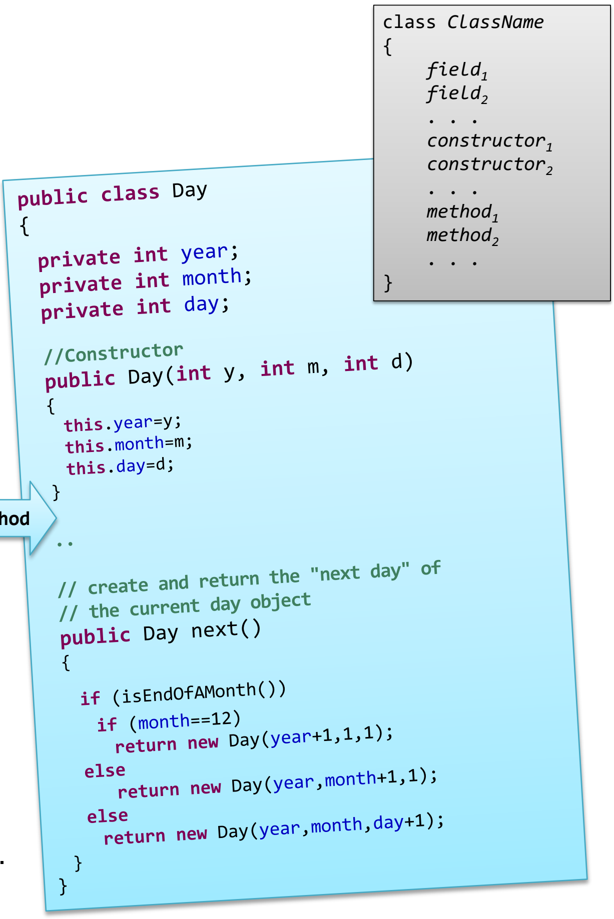
A. Simple Class Example 1 - Day (See Lab01 Q1)

Add one more method to the **Day** class:

(1) Fill in the blank according to the comment:

```
// advance the current day object by 1 day
public void advance()
{
   if (isEndOfAMonth())
   {
      if (month==12)
      {
         year=__________;
         month=_________;
         day=____________;
      }
      else
      {

      }
   }
   else
   {

   }
}
```

Add this method



```
class ClassName
{
    field1
    field2
    . . .
    constructor1
    constructor2
    . . .
    method1
    method2
    . . .
}
```

```
public class Day
{
  private int year;
  private int month;
  private int day;

  //Constructor
  public Day(int y, int m, int d)
  {
    this.year=y;
    this.month=m;
    this.day=d;
  }
  ..

  // create and return the "next day" of
  // the current day object
  public Day next()
  {
    if (isEndOfAMonth())
      if (month==12)
        return new Day(year+1,1,1);
      else
        return new Day(year,month+1,1);
    else
      return new Day(year,month,day+1);
  }
}
```

Note: some methods return a result, some do not

(2) Both **.advance()** and **.next()** calculate the next day. Complete the code below based on the comments.

Main.java

```
Day d1 = new Day(2014, 1, 28);
System.out.println(d1.toString());  //Show 28 Jan 2014
______.advance();                   //Advance one day
______.next();                      //Advance one day
System.out.println(d1.toString());  //Show 30 Jan 2014
```

B. Simple Class Example 2- Employee

- Consider a simplified Employee class used in a payroll system:

```
class Employee
{
   // instance fields
   private String name;
   private double salary;
   private Day hireDay;

   // constructor
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = new Day(year,month,day);
   }

   public String getName() {return name;}

   public double getSalary() {return salary;}

   public Day getHireDay() {return hireDay;}

   public void raiseSalary(double percent)
   {
      double raise = salary * percent /100;
      salary += raise;
   }
}
```

- Alternatively:

```
class Employee
{
    // instance fields
    private String name;
    private double salary=0;
    private Day hireDay;

    ..
}
```

Initialization can be done for instance fields.

- We can even do some method call and computation:

`private double salary = Math.random()*10000;`

- Using the Employee class in a program:

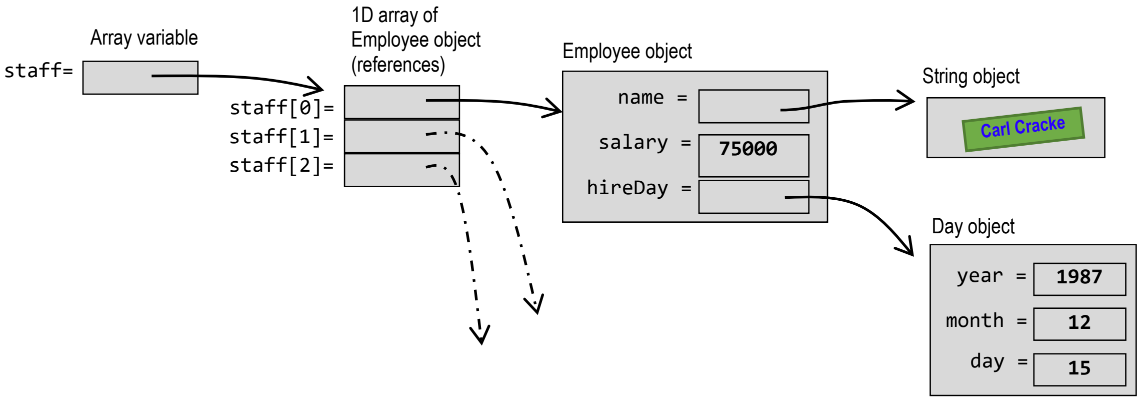
```
public class Main_EmployeeTest
{
   public static void main(String[] args)
   {
      // fill the staff array with three Employee objects
      Employee[] staff = new Employee[3];
      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      // raise everyone's salary by 5%
      for (Employee e : staff)
         e.raiseSalary(5);

      // printing
      for (Employee e : staff)
         System.out.println(
            "name=" + e.getName() +
            ",salary=" + e.getSalary() +
            ",hireDay=" + e.getHireDay());
   }
}
```

Output

```
name=Carl Cracker,salary=78750.0,hireDay=15 Dec 1987
name=Harry Hacker,salary=52500.0,hireDay=1 Oct 1989
name=Tony Tester,salary=42000.0,hireDay=15 Mar 1990
```

C. Array of objects

```
public class Main_EmployeeTest
{
    public static void main(String[] args)
    {
        // fill the staff array with three Employee objects
        Employee[] staff = new Employee[3];
        staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
        staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
        staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
        ..
    }
}
```



D. .toString()

If an object's text representation is needed, Java automatically looks for its .toString() method.

```
public class Main_EmployeeTest
{
  public static void main(String[] args)
  {
    ..
    // printing
    for (Employee e : staff)
      System.out.println(
        "name=" + e.getName() +
        ",salary=" + e.getSalary() +
        ",hireDay=" + e.getHireDay());
  }
}
```

```
class Employee
{
    // instance fields
    private String name;
    private double salary;
    private Day hireDay;
    ..
    public Day getHireDay()
    {
        return hireDay;
    }
    ..
}
```

```
public class Day
{
  private int year;
  private int month;
  private int day;
  ..

  // Return a string for the day (dd MMM yyyy)
  public String toString()
  {
    final String[] MonthNames = {
      "Jan", "Feb", "Mar", "Apr","May", "Jun",
      "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    return day+" "+
      MonthNames[month-1] +
      " "+ year;
  }
  ..
}
```

Output:

```
name=Carl Cracker,salary=78750.0,hireDay=15 Dec 1987
name=Harry Hacker,salary=52500.0,hireDay=1 Oct 1989
name=Tony Tester,salary=42000.0,hireDay=15 Mar 1990
```

Explanation:

- The reference of a Day object is returned by `e.getHireDay()`
- However, a string is needed at `",hireDay=" + e.getHireDay());`
- JAVA automatically looks for a .toString() method of the Day class and invoke it for the Day object
- .toString() returns a string. Done!

II. Introduction to OOP

OO Programs

- An O-O program is made up of objects

  Each object has
  - a specific functionality exposed to its users
  - A hidden implementation

- Basically, as long as an object satisfies your specifications, you don't care how the functionality is implemented
  - E.g., consider `Scanner` objects: `.nextInt`, `.hasNext`

- Traditional Structured Programming vs OOP
  - Structured Programming: Designing a set of procedures to solve a program

    Usually top-down (starting from `main()`)

  - OOP: Puts data first, then looks at the algorithms to operate on the data

    There is no "top".

    Begin by studying a description of the application, and

    identifying classes (often from nouns) and then add methods (often as verbs).

- OOP: More appropriate for larger problems:
  - Because of well-organized code around data
  - E.g., Easier to find bugs which causes faults on a piece of data

Classes

- A class is the template / blueprint from which objects are made.
  - Like cookie cutters from which cookies are made

  - Classes in Java program
  - Provided by the standard Java library
  - Created by ourselves for our application's problem domain

- Terminologies:

  "object" ~ "instance" ~ "object instance"

  - **Instance** of a class : an Object created from a class
    - E.g., **Scanner s; s = new Scanner(..);** // s is an instance of the `Scanner` class
    - E.g., **Day d; d = new Day(2014,1,19);** // d is an instance of the `Day` class

  - **Instance fields**: Data in an object (e.g., `day`, `month`, `year`)

  - **Object state**: The set of values in the instance fields of an object

    Or "how does the object react when its method runs"

  - **Methods**: procedures which operate on the data.

  - **Encapsulation** (sometimes called information hiding)
    - Combine data and behavior in one package, consider as implementation details
    - Implementation details are hidden away from the users of the objects
    - E.g., We use `Scanner` objects, but their implementation details are encapsulated in the `Scanner` class.

III. Objects and Object variables - The Constructors and the new operator

- Work with objects - we often:
  - First: construct them and specify initial state
    (give values for instance fields like `day`, `month`, `year`)
  - Then: apply methods to the objects.

- **Constructors:**
  - We use constructors to construct new instances
  - A constructor is a special method
    - ☆ Purpose: construct and **initialize objects**
    - ☆ Named the same as the **class name**, **no return value**
    - ☆ Always called with **the new operator**
      e.g. **`new Day(2014,1,3)`**
    - ☆ A class can have **more than one constructor**

e.g. Add the second constructor to the **Day** class:

```
public Day (int y, int m, int d) {..}
public Day (int y, int nth_dayInYear) {..}
```

Usage of such a constructor:

```
d1=new Day(2014,45); //The 45th day in 2014
System.out.println(d1); //14 Feb 2014
```

- **Examples of Constructors**

```
class Employee
{
   // instance fields
   private String name;
   private double salary;
   private Day hireDay;

   // constructor
   public Employee(String n, double s,
         int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = new Day(year,month,day);
   }
    // ... more methods
}
```

```
public class Day
{
 private int year;
 private int month;
 private int day;

 //Constructor
 public Day(int y, int m, int d)
 {
  this.year=y;
  this.month=m;
  this.day=d;
 }

 //.. more methods
}
```

- **More notes on Constructors**
  - Recall: constructors must be called with the `new` operator

    We **CANNOT** apply it solely for resetting the instance fields like:

    ☒ `birthday.Day(2014,1,27);` // Willing to change the instance fields in birthday?
    "Error: The method Day(int, int, int) is undefined for the type Day"

- **When an object is created, we may:**

1. Pass the object to a method:

   `System.out.println(new Day(2014,1,15)); //` ☆ Note: actually `.toString()` is called.

2. Apply a method to the object just constructed:

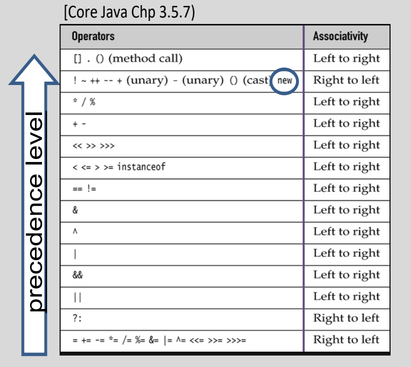
```
System.out.print((new Day(2014,1,15)).next());
```

- Or simply `System.out.print(new Day(2014,1,15).next());`

  Reason: constructor is always called with the new operator

- Which style to write?

  Suggestion: *Choose the one you feel comfortable to read.*
  But you should be able to understand both styles when you read others' code.



[Core Java Chp 3.5.7)

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

precedence level

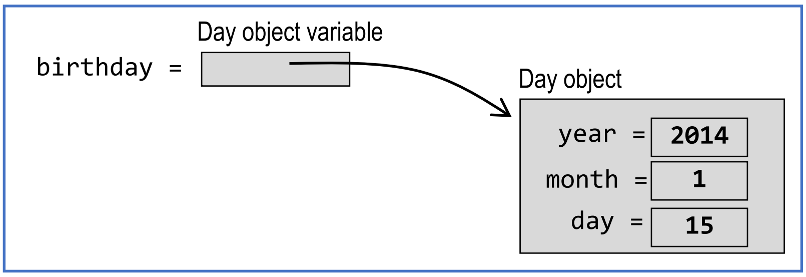
3. Hang on to the object with the use of an object variable:

   `Day birthyday;` //This variable, birthday, doesn't refer to any object yet

   `birthday = new Day(2014,1,15);` //Now, initialize the variable to refer to a new object

   Or, combined as:

```
Day birthday = new Day(2014,1,15);
```



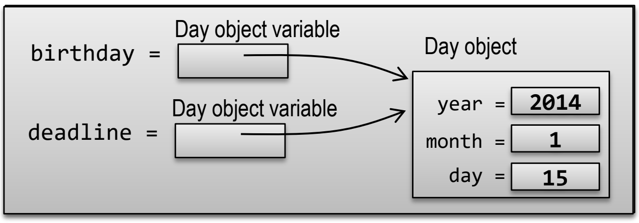
- **An object variable:**

  ☆ Doesn't actually contain an object.

  ☆ The value of any object variable is "a reference to an object that is stored separately".

  ☆ We set a variable to refer to an existing object of the matching type:

Day **birthday**, **deadline**;
birthday = new Day(2014,1,15);
deadline = birthday;



- For convenience, we often **verbally** say *"object"* instead of *"object variable"*.
  But we need to bear in mind the actual picture.

- Note: The `return` value of the `new` operator is also a reference.

- Special value: `null` (means nothing)
  We can set an object variable to `null` to mean that it refers to no object.
  E.g., `Day d1=null; //later checking: if (d1==null) ..`

- Local variables (ie. variables defined in a method) are not automatically initialized.
  To initialize an object variable: we can set it to `null`, or refer to an existing object, or use `new`

An exercise:

Consider this simplified Day class.

Your task: Match ① - ⑥ with the descriptions:

a. ***Error: d1 may not have been initialized***
[This is a compilation problem in Java.
If not fixed, we cannot run the program.]

b. ***Runtime exception: java.lang.NullPointerException***

c. ***print: 15 Jan 2014***

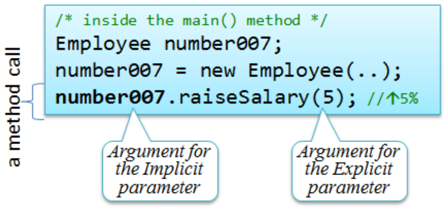
d. ***print: null***

```
class Day
{
   private int year;
   private int month;
   private int day;

   public Day(int y, int m, int d)
   {
       this.year=y;
       this.month=m;
       this.day=d;
   }

   public String toString()
   {
       final String[] MonthNames = {
                                     "Jan", "Feb", "Mar", "Apr",
                                     "May", "Jun", "Jul", "Aug",
                                     "Sep", "Oct", "Nov", "Dec"};

       return day+" "+ MonthNames[month-1] + " "+ year;
   }
}
public class Main
{
   private static void testing1()
   {
      Day d1;
      System.out.println(d1);                 ① __
      System.out.println(d1.toString());      ② __
   }

   private static void testing2()
   {
      Day d1 = null;
      System.out.println(d1);                 ③ __
      System.out.println(d1.toString());      ④ __
   }

   private static void testing3()
   {
      Day d1 = new Day(2014,1,15);
      System.out.println(d1);                 ⑤ __
      System.out.println(d1.toString());      ⑥ __
   }

   public static void main(String[] args)
   {
      testing_(); //Change this line to testing1(), testing2(), or testing3() for testing
   }
}
```

IV. The Implicit Parameter (calling object), and the *this* keyword

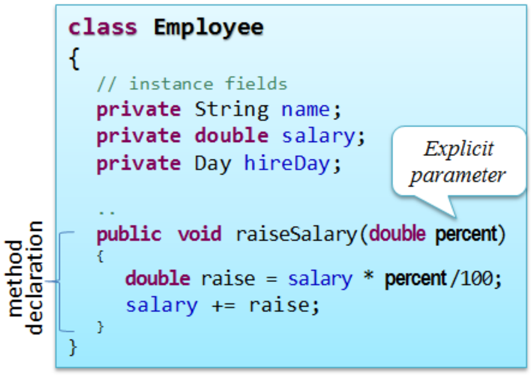
- **Implicit and Explicit Parameters**

  Suppose the **raiseSalary** method is called like this:



```
/* inside the main() method */
Employee number007;
number007 = new Employee(..);
number007.raiseSalary(5); //↑5%
```

a method call
Argument for the Implicit parameter
Argument for the Explicit parameter



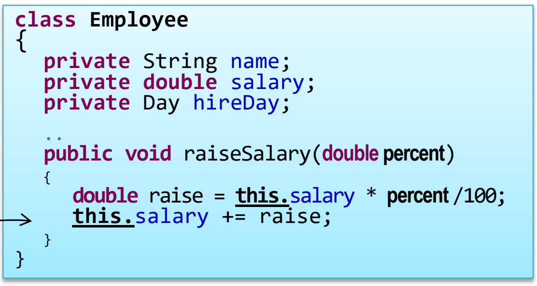
```
class Employee
{
   // instance fields
   private String name;
   private double salary;
   private Day hireDay;

   ..
   public void raiseSalary(double percent)
   {
      double raise = salary * percent/100;
      salary += raise;
   }
}
```

We say that the method has 2 parameters:

- ***Implicit parameter:*** For the object that the method is invoked on. (Here: `number007`) also called "*calling object*"
  "Implicit: not stated directly"
- ***Explicit parameter:*** The parameter listed in the method declaration. (Here: `percent`)

- **The `this` keyword**

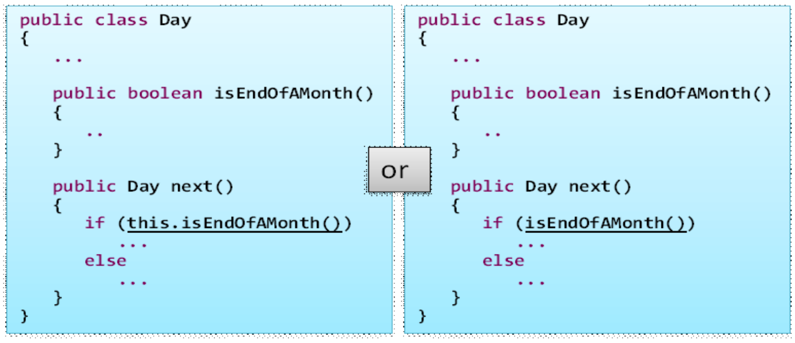
  - In every method, the keyword **`this`** refers to the implicit parameter.
  - E.g. we can rewrite `.raiseSalary`:



```
class Employee
{
   private String name;
   private double salary;
   private Day hireDay;

   ..
   public void raiseSalary(double percent)
   {
      double raise = this.salary * percent/100;
      this.salary += raise;
   }
}
```

Some programmers prefer this style (clearly distinguishes between instance fields and local variables)

  - When method A invokes method B to handle the implicit parameter (calling object), the implicit parameter is either omitted or specified using the this keyword.

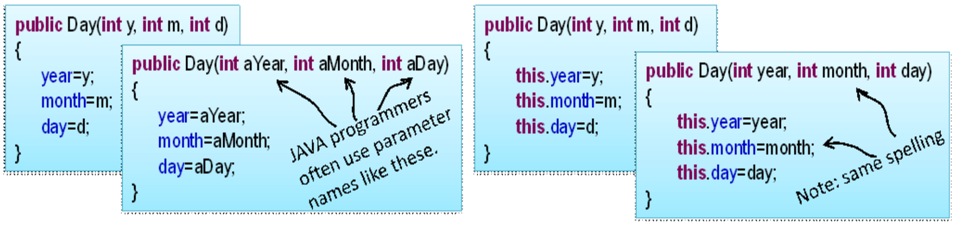


```
public class Day
{
   ...

   public boolean isEndOfAMonth()
   {
      ..
   }

   public Day next()
   {
      if (this.isEndOfAMonth())
         ...
      else
         ...
   }
}
```

or

```
public class Day
{
   ...

   public boolean isEndOfAMonth()
   {
      ..
   }

   public Day next()
   {
      if (isEndOfAMonth())
         ...
      else
         ...
   }
}
```

  - We have seen this in the constructor of the Day class. More equivalent versions:



```
public Day(int y, int m, int d)
{
   year=y;
   month=m;
   day=d;
}
```

```
public Day(int aYear, int aMonth, int aDay)
{
   year=aYear;
   month=aMonth;
   day=aDay;
}
```

JAVA programmers often use parameter names like these.

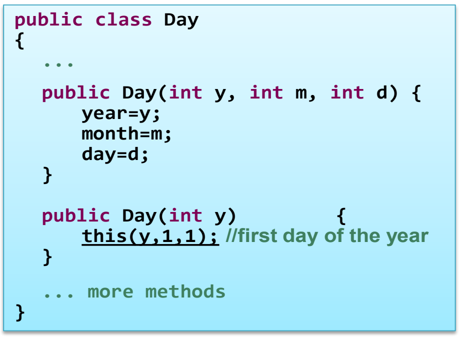
```
public Day(int y, int m, int d)
{
   this.year=y;
   this.month=m;
   this.day=d;
}
```

```
public Day(int year, int month, int day)
{
   this.year=year;
   this.month=month;
   this.day=day;
}
```

Note: same spelling

  - We can use this to call another constructor.

    Note: Must be written as the first statement in a constructor



```
public class Day
{
   ...
   public Day(int y, int m, int d) {
      year=y;
      month=m;
      day=d;
   }

   public Day(int y)          {
      this(y,1,1); //first day of the year
   }

   ... more methods
}
```

V. Day – Change of implementation -Using integer data int yyyymmdd;

A new implementation of the `Day` class
- using an integer data as `int yyyymmdd;`

```
public class Day {

   private int yyyymmdd;

   //Constructor
   public Day(int y, int m, int d) {
      yyyymmdd=y*10000+m*100+d;
   }

   // Return a string for the day like dd MMM yyyy
   public String toString() {
      final String[] MonthNames = {
            "Jan", "Feb", "Mar", "Apr",
            "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec"};

      return getDay()+" "+ MonthNames[getMonth()-1] + " "+ getYear();
   }
   public int getDay() {return yyyymmdd%100;}
   public int getMonth() {return yyyymmdd%10000/100;}
   public int getYear() {return yyyymmdd/10000;}
   ../Other methods
}
```

Question:
To use this new implementation, do the users need to adjust their code?
Answer:

No.
The way to create and use Day objects (call the public methods) are the same.

VI. Mutator, Accesor Methods getXX, setXX

- **Public methods** are provided for outsiders to act on the data:
  - **Accessor methods (getters)** allow outsiders to obtain the data
    - Often named as `get`.., eg. `getDay()`
    - May not return the values of an instance field literally.
    - E.g., The `Day` class has the instance field `int yyyymmdd;`
      the asccessor methods are:

      ```
      public int getDay()    {return yyyymmdd%100;}
      public int getMonth()  {return yyyymmdd%10000/100;}
      public int getYear()   {return yyyymmdd/10000;}
      ```

  - **Mutator methods (setters)** change the current data of an object
    - Often named as set..., eg. `setDay()`
    - May apply special checking, eg. never make `salary` negative.

- **Note: It is WRONG to think that** "we should add get… and set… for most instance fields!"
  Why? – See the coming explanation later in this topic: *Avoid unnecessary accessor and mutator methods*

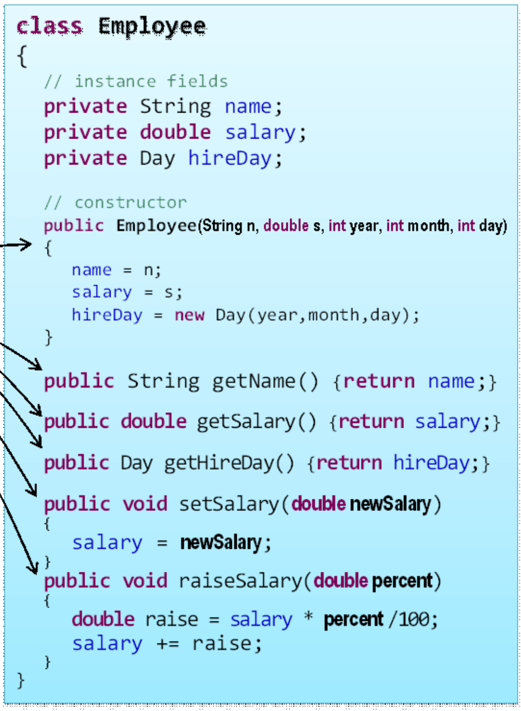
VII. public vs private, Encapsulation issue

Public vs Private

- **Public methods**

  Any method in any class can call the method.

  - The followings methods are often public:
    - Constructors
    - Accessor methods
    - Mutator methods
    - Other methods which outsiders might need



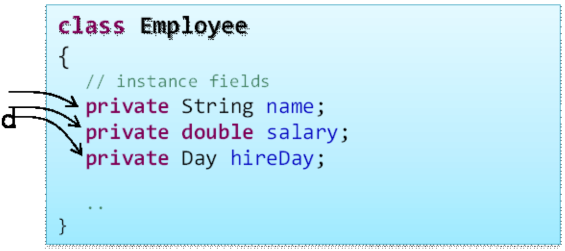
```
class Employee
{
   // instance fields
   private String name;
   private double salary;
   private Day hireDay;

   // constructor
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = new Day(year,month,day);
   }
   public String getName() {return name;}
   public double getSalary() {return salary;}
   public Day getHireDay() {return hireDay;}
   public void setSalary(double newSalary)
   {
      salary = newSalary;
   }
   public void raiseSalary(double percent)
   {
      double raise = salary * percent /100;
      salary += raise;
   }
}
```

- **Public instance fields** ☠

  For proper practice of encapsulation, we seldom declare instance fields as public.

- **Private Instance fields**:

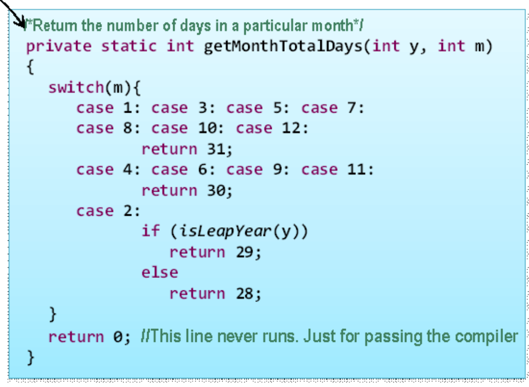
  These instance fields can be accessed only by through methods of the `Employee` class itself.



```
class Employee
{
   // instance fields
   private String name;
   private double salary;
   private Day hireDay;

   ..
}
```

- **Private methods**

  Sometimes implemented, which take the role of *helper methods*.



```
/*Return the number of days in a particular month*/
private static int getMonthTotalDays(int y, int m)
{
   switch(m){
      case 1: case 3: case 5: case 7:
      case 8: case 10: case 12:
            return 31;
      case 4: case 6: case 9: case 11:
            return 30;
      case 2:
            if (isLeapYear(y))
               return 29;
            else
               return 28;
   }
   return 0; //This line never runs. Just for passing the compiler
}
```

- A method can access the private data of all objects of its class, not only the implicit parameter.

```
class X
{
  private int data;
  public X(int d) {data=d*2;}
  public void doSomething(X r)
  {
    X s = new X(8);
    System.out.println(this.data);
    System.out.println(r.data);
    System.out.println(s.data);
  }
}
```

Avoid returning reference from accessor methods

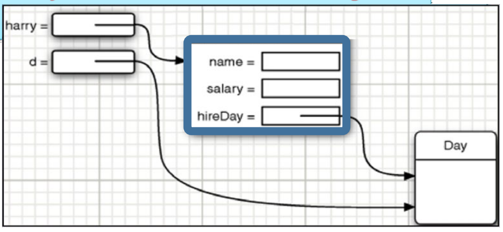
```
class Employee {
    private Day hireDay;
    ..
    public Day getHireDay() { return hireDay;}
    ..
}
```

This accessor method breaks encapsulation ☹

* Instance fields should be changed through the methods provided by the class itself.

Now problem happens - The user can write:

```
Employee harry;
harry = new Employee("harry", 75000, 1987, 12, 15);
Day d = harry.getHireDay(); //d and harry.hireDay refer to the same object
d.advance(); //changes it to 1987-12-16 !!
```



Solution / Rule of thumb:

If you need to return a *multable* data field,
you should return a new copy of the Day object.
(learn "object cloning" later)

More on mutable , immutable:

mutable: after construction, there are still other way that the object can get changed.
immutable: after construction, there is no way that the object can get changed.

e.g. **Strings are made immutable -**

String methods may return new resultant String objects,
but String methods never amend the content of the original string object.

That is, the String class does not provide you any method to change the object.
This is known as *immutable*. [http://docs.oracle.com/javase/tutorial/java/data/strings.html]

```
public static void main(String[] args)
{
  String s1, s2;

  s1 = "Everybody gets a good grade";
  s2 = s1.replace("a good grade", "an A+");

  System.out.println(s1); //Output: Everybody gets a good grade
  System.out.println(s2); //Output: Everybody gets an A+
}
```

The **replace()** method of **String**.
s1 itself does not change!

Avoid unnecessary accessor and mutator methods

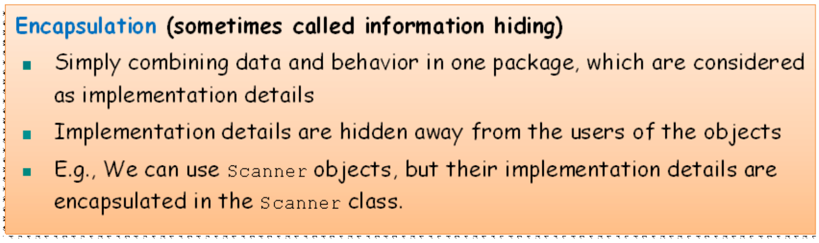
Some beginners think that *"we should add accessor and mutator methods for all instance fields"*.

This is BOTH WRONG! ☠

- Getters and setters in this way actually break encapsulation (See next section)
- We should add them only if really needed.
- We can often find replacements:
  Example 1: Inside a Rectangle class we remove .getX() and .getY(),
  but add the useful method .draw()
  Example 2: In the Day class, it is bad to provide setYear(..), setMonth(..), setDay(..)
  Reason: easily misused, e.g., change 2016-02-29 to 2016-02-**30**
- *"Don't ask for the information you need to do the work;*
  ask the object that has the information to do the work for you. "
  - http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html

VIII. Benefits of Encapsulation

Recall:



Encapsulation (sometimes called information hiding)

- Simply combining data and behavior in one package, which are considered as implementation details
- Implementation details are hidden away from the users of the objects
- E.g., We can use `Scanner` objects, but their implementation details are encapsulated in the `Scanner` class.

Benefits of Encapsulation

It is a good practice to encapsulate data as **private** instance fields.

1. **Protect the data from corruption by mistake.**
   Outsiders must access the data through the provided public methods
2. **Easier to find the cause of bug**
   Only methods of the class may cause the trouble
3. **Easier to change the implementation**
   - e.g. change from using 3 integers for `year,month,day` to using 1 integer `yyyymmdd`
   - We only need to change the code in the class for that data type
   - The change is invisible to outsiders, hence not affect users.

IX. The Final keyword - Method constant, Class constant, Final Instance Fields

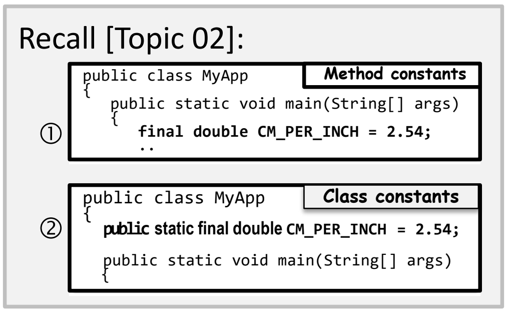
The `Final` keyword

Can be used to:

① define Method constants (see topic 02)

② define Class constants (see topic 02)

③ declare instance fields as "**Final**" (see below)



Recall [Topic 02]:

① Method constants

```
public class MyApp
{
    public static void main(String[] args)
    {
        final double CM_PER_INCH = 2.54;
        ..
```

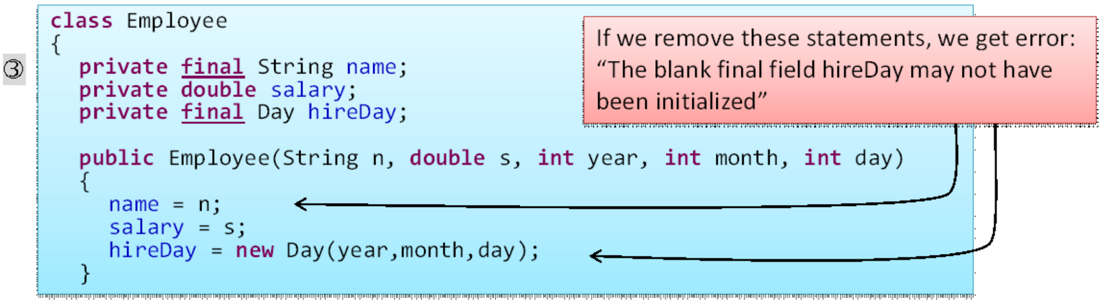
② Class constants

```
public class MyApp
{
  public static final double CM_PER_INCH = 2.54;

  public static void main(String[] args)
  {
```

Final Instance Fields

When we declare an instance field as `final`

- It is to be initialized once latest when the object is constructed.
- It cannot be modified again.



③

```
class Employee
{
   private final String name;
   private double salary;
   private final Day hireDay;

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = new Day(year,month,day);
   }
```

If we remove these statements, we get error: "The blank final field hireDay may not have been initialized"

X. The Static keyword - Class constant (used with final), Static Method, Static Fields

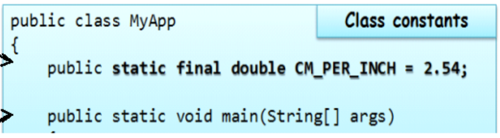
The static keyword

- Used to denote fields and methods that belong to a class (but not to any particular object):

1. **Class constant** (used with final)

   Example 1: MyApp.CM_PER_INCH

   Example 2: System.out



```
public class MyApp
{
    public static final double CM_PER_INCH = 2.54;

    public static void main(String[] args)
```

2. **Static method** (*class-level method*)

   Example 1: Day.isLeapYear(y)

   Example 2: main()

   Note: Static methods do not have implicit parameter (ie. no calling object, no this)

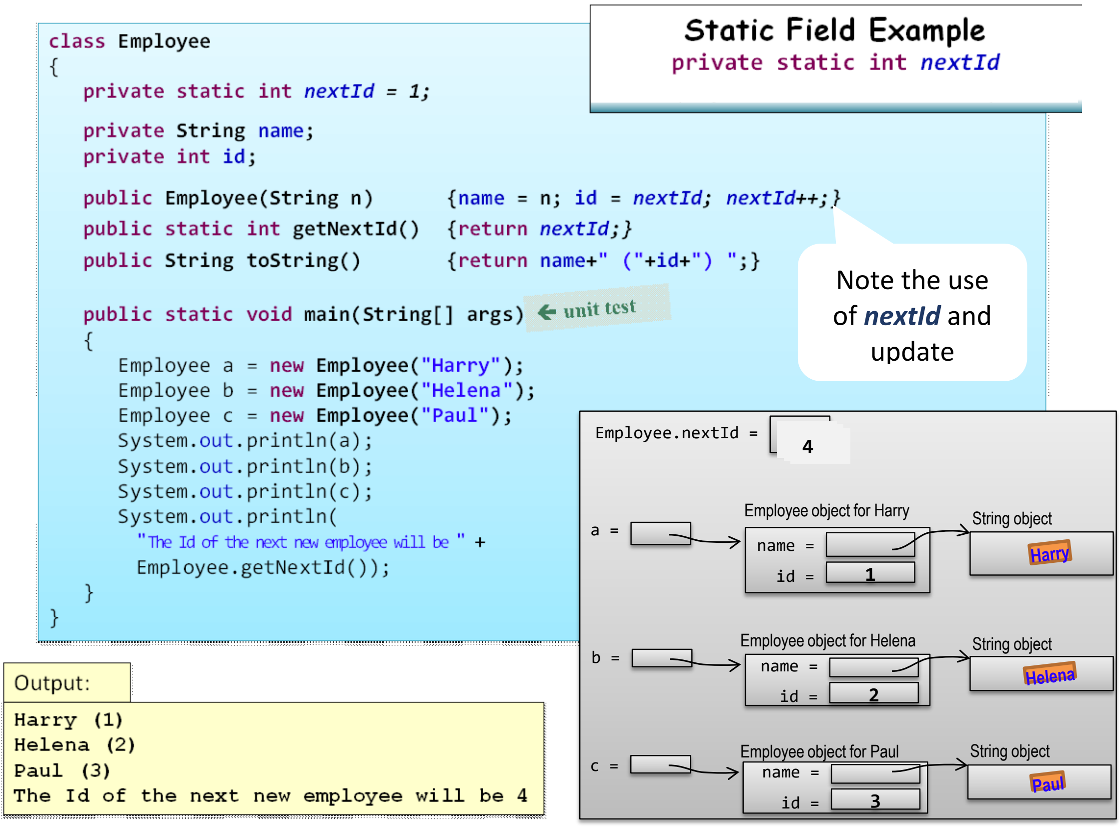
   When to use Static method?

   Answer: 1. when we don't need to access the object state because all needed parameters are supplied as explicit parameters (eg. Day.isLeapYear(y))

   2. when the method only need to access static fields of the class (eg. Employee.getNextId())

3. **Static Fields**

   A static field is a single field owned by the whole class
   i.e., Not a field per object instance. (taught in next slide)



Static Field Example
private static int nextId

```
class Employee
{
   private static int nextId = 1;

   private String name;
   private int id;

   public Employee(String n)       {name = n; id = nextId; nextId++;}
   public static int getNextId()   {return nextId;}
   public String toString()        {return name+" ("+id+") ";}

   public static void main(String[] args)
   {
      Employee a = new Employee("Harry");
      Employee b = new Employee("Helena");
      Employee c = new Employee("Paul");
      System.out.println(a);
      System.out.println(b);
      System.out.println(c);
      System.out.println(
        "The Id of the next new employee will be " +
        Employee.getNextId());
   }
}
```

← unit test

Note the use of *nextId* and update

Output:

```
Harry (1)
Helena (2)
Paul (3)
The Id of the next new employee will be 4
```

☆ If a field is non-static, then each object has its own copy of the field. (name, id)

☆ If a field is static, then there is only one such field per class. (nextId)

☆ **Static method** for accessing static fields:
To provide a public method for outsiders to access a static field, we usually make the method static *i.e. class-level method* (getNextId)

XI. Method Parameters - Java uses "call by value"

Method Parameters

Java uses "call by value":

- Methods receive copy of parameter values
- The original arguments given by the caller do not change (Employee a, b in the following example)
- Copy of object reference lets method modify object (.salary✓)

```
class Employee
{
   private String name;
   private double salary;

   public Employee(String n, double s)  {name = n; salary=s;}
   public String toString() {return name+" ($"+salary+") ";}

   private static void swapEmployee(Employee e1,Employee e2)
   {
       Employee temp=e1;       //Cannot really swap
       e1=e2;                  // caller’s arguments
       e2=temp;
   }

   private static void swapSalary(Employee e1,Employee e2)
  {
       double temp=e1.salary; //OK - swapping the
       e1.salary=e2.salary;    //  instance fields can
       e2.salary=temp;         //  be done.
   }
}
```

See the given example:

a and b are object variables, ie. references to objects.

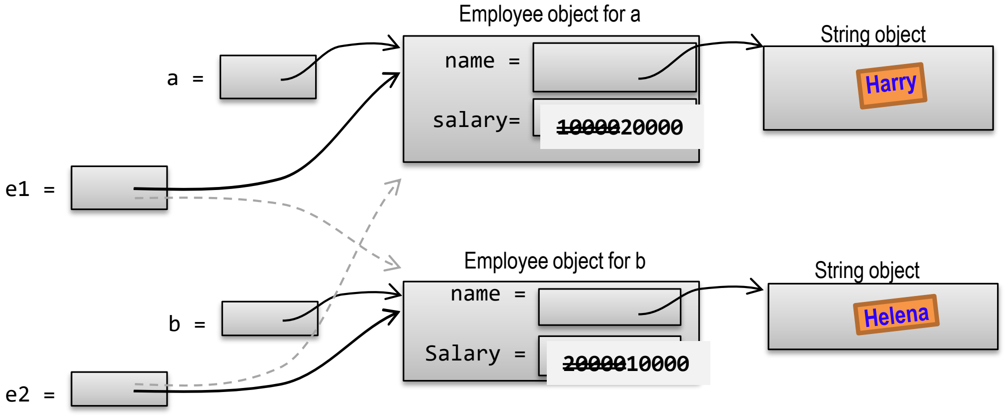
e1 and e2 are copies of a and b.

- Changing e1, e2 do not affect a, b
- But using e1 and e2 to refer to the objects and access the salary fields, it does really mean the salaries in the objects pointed by a and b.

```
public static void main(String[] args)
{
   Employee a = new Employee("Harry",10000);
   Employee b = new Employee("Helena",20000);
   Employee.swapEmployee(a,b); //No change to a and b
   Employee.swapSalary(a,b); //Salary swapped!
   System.out.println(a);
   System.out.println(b);
}
```

Output:

```
Harry ($20000.0)
Helena ($10000.0)
```



XII. Default field initialization (not for local variables)

Default field initialization (Note: not for variables!)

- If we don't set a field explicitly in a constructor,
  - * It is set to a default value:

    Numbers: 0

    Boolean values: `false`

    Object references: `null`

- However, local variables are NOT initialized

```
class Employee
{
   private String name;
   private double salary;
   public Employee(String n, double s)  { /* do nothing */}

   public String toString() {return name+" ($"+salary+") ";}

   public static void main(String[] args) // unit test for Employee class
   {
      Employee a = new Employee("Harry",10000);
      System.out.println(a); //show: null ($0.0) 🡸 no problem!
      String s;
      int x;
      System.out.println(s); // Error: The local variable s may not have been initialized
      System.out.println(x); // Error: The local variable x may not have been initialized
   }
}
```

XIII. Overloading methods, Signature

Overloading methods

- ○ ie. "use the same names for different methods"
- ○ But the parameters must be different
- ○ So that the compiler can determine which to invoke.
  E.g. `Day v1,v2; ...; swap(v1,v2);` //invoke a method called "swap" which accepts 2 Day objects.
- ○ Terminology: Name + parameters = **Signature**

Example 1:

```
void swap(Employee e1, Employee e2) {..}
void swap(Day d1, Day d2) {..} //swap all year,day,month fields
```

Example 2: [Overloaded Constructors]

```
public class Day
{
  ...
  public Day(int y, int m, int d) {year=y;month=m;day=d;}
  public Day(int y) {this(y,1,1);} //first day of the year

  ... more methods
}
```

XIV. Default Constructor - Constructors with zero argument - designed and automatic generated

Default Constructor

- Programmers sometimes create a constructor with no argument, like:

```
public Employee()
{
      name = "";
      salary = 0;
      hireDay = new Day(2014,1,1);
}
```

It is called when we create an object like: `Employee e = new Employee();`

- If we don't provide constructor, then a default constructor is automatically generated. Like this:

```
public Employee()
{
      name = null;
      salary = 0;
      hireDay = null;
}
```

Recall:
Actually, Fields are already by default null or zero or false. ☺

- If we provide 1 or more constructors, then the above will not be generated for us.

XV. Class Design Hints

Hints for Class Design

1. Always keep data private.
2. Always initialize data
3. Group instance fields as objects when appropriate

```
public class Customer
{
   private String street;
   private String city;
   private String state;
   private String zip;
   ..
}
```

→

```
public class Customer
{
   private Address address;
   ..
}
```

Create this class

4. Not all fields need individual field accessors and multators
   e.g. once hired, we won't change `hireDay`, therefore `setHireDay(..)` is not needed.

5. Break up classes that have too many responsibilities

```
public class CardDeck // bad design
{
   private int[] value;
   private int[] suit;

   public CardDeck() {...}
   public int getTopValue() {...}
   public int getTopSuit() {...}
}
```

→

```
public class CardDeck
{
   private Card[] cards;

   public CardDeck() {...}
   public Card getTop() {...}
}
```

```
public class Card
{
   private int value;
   private int suit;

   public Card(int v, int s) {...}
   public int getValue() {...}
   public int getSuit() {...}
}
```

6. Proper naming. For classes, use noun, or adjective+noun, or gerund (-ing).
   E.g. `Order`, `RushOrder`, `BillingAddress`

--- end ---

Contents Topic 04 - Inheritance

I. Classes, Superclasses, and Subclasses

Inheritance

- **Inheritance**: A fundamental concept of OO Programming
- Idea: create new classes that are built on existing classes
- Terminologies:

❑ **subclass / superclass**
The new class is called a *subclass* (*derived class*, or *child class*).
The existing class is called a *superclass* (*base class*, or *parent class*).

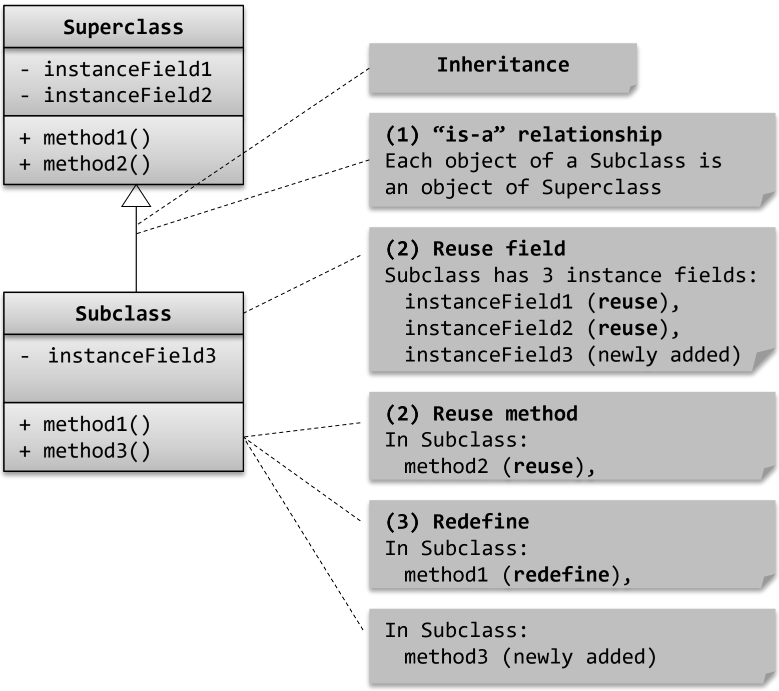
❑ **"is-a" relationship**
An object of the subclass is also an object of the superclass.
E.g. Cats are animals. Tom is a cat. Tom is also an animal.

❑ **Reuse**
All attributes/methods of the superclass can be reused (or inherited) in the subclass.
However, constructors are not inherited.

❑ **Redefine** (or called **Override**)
The methods of the superclass can be redefined in the subclass.

UML notation for Inheritance:

- Illustrating (1) “is-a”, (2) Reuse, (3) Redefine



Inheritance

(1) “is-a” relationship
Each object of a Subclass is an object of Superclass

(2) Reuse field
Subclass has 3 instance fields:
instanceField1 (**reuse**),
instanceField2 (**reuse**),
instanceField3 (newly added)

(2) Reuse method
In Subclass:
method2 (**reuse**),

(3) Redefine
In Subclass:
method1 (**redefine**),

In Subclass:
method3 (newly added)

We can add new methods and fields to adapt the sub class to new situations (instanceField3, method3)

Inheritance in JAVA:

- E.g. In a university, each university staff is also a library member ie. an obvious “is-a” relationship.

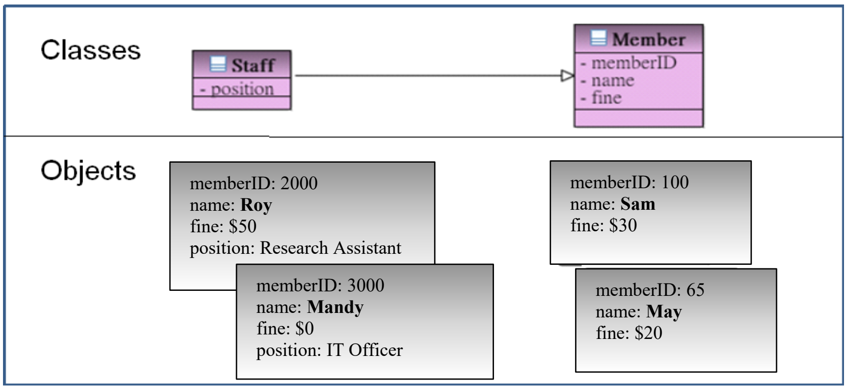
- Java keyword **extends**:

```
class Staff extends Member
{
    added methods and fields
}
```

```
class Manager extends Employee
{
    added methods and fields
}
```

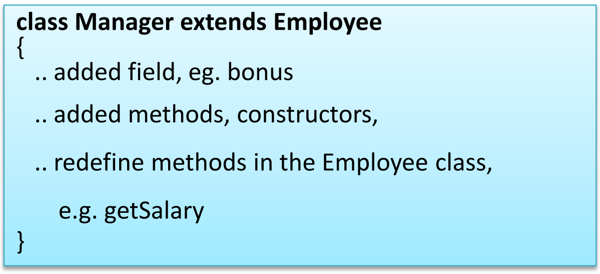
[Lab04]

- Class vs Object



* Note: in actual java implementation, the name fields store references to string objects, and the position fields store the codes of the positions

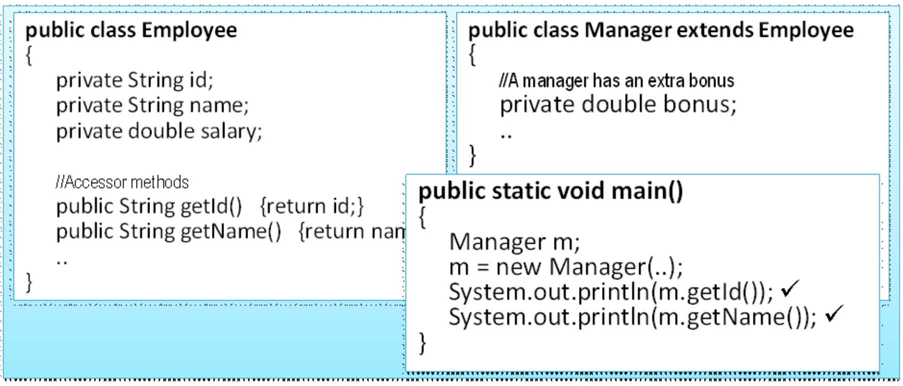
- **Subclasses -** subclasses have more data and functionality than their parent classes.



```
class Manager extends Employee
{
  .. added field, eg. bonus
  .. added methods, constructors,
  .. redefine methods in the Employee class,
     e.g. getSalary
}
```

- **Examples of Reuse:**

  [Recall]
  Reuse:
  All attributes/methods of the superclass can be reused (or inherited) in the subclass.
  However, constructors are not inherited.



```
public class Employee
{
    private String id;
    private String name;
    private double salary;

    //Accessor methods
    public String getId()   {return id;}
    public String getName()   {return nan
    ..
}
```

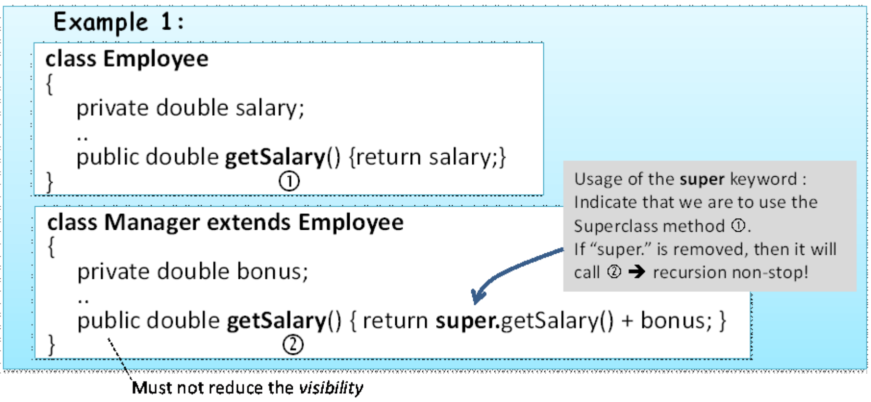
```
public class Manager extends Employee
{
    //A manager has an extra bonus
    private double bonus;
    ..
}
```

```
public static void main()
{
    Manager m;
    m = new Manager(..);
    System.out.println(m.getId()); ✓
    System.out.println(m.getName()); ✓
}
```

- **Examples of Redefine:**

  [Recall]
  Redefine:
  (or called Override)

  The methods of the superclass can be redefined in the subclass.



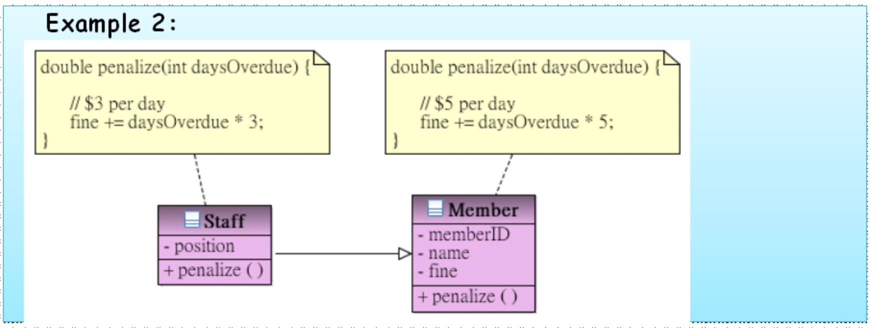
Example 1:

```
class Employee
{
    private double salary;
    ..
    public double getSalary() {return salary;}
}                       ①
```

```
class Manager extends Employee
{
    private double bonus;
    ..
    public double getSalary() { return super.getSalary() + bonus; }
}                       ②
```

Usage of the **super** keyword :
Indicate that we are to use the Superclass method ①.
If "super." is removed, then it will call ② ➔ recursion non-stop!

Must not reduce the *visibility*



Example 2:

```
double penalize(int daysOverdue) {
    // $3 per day
    fine += daysOverdue * 3;
}
```

```
double penalize(int daysOverdue) {
    // $5 per day
    fine += daysOverdue * 5;
}
```

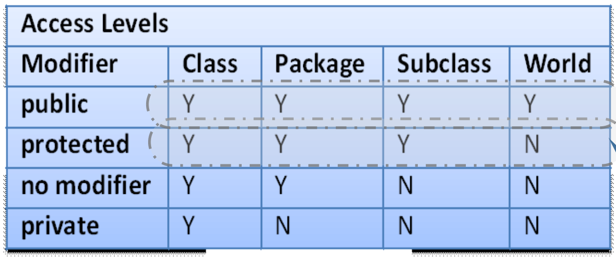
Access Level Modifiers

[http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html]

- Determine whether other classes can use a particular field or method.
  Or we say, "affect the visibility"

  ❑ At the class level, the class can be
  1. ***public***: visible to all classes everywhere.
  2. **no modifier**: visible only within its own package (***package-private***)

  ❑ At the member (field / method) level, it can be
  1. **public**
  2. **protected** – Visible to the package and all subclasses
  3. **no modifier** (package-private)
  4. **private** – Visible to the class only

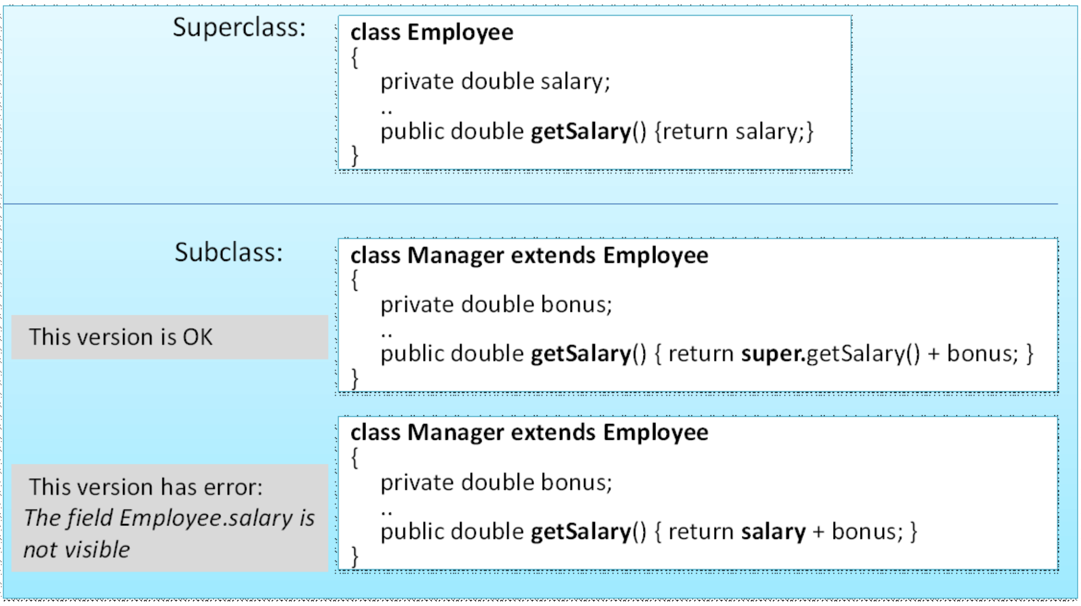


Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

E.g. If a method (or a field) is **public**, then it can be used by the code in the same class, the package which contains this class, the subclasses which inherit this class, and the world

E.g. If a method (or a field) is **protected**, then it can be used only by the code in the same class, the package which contains this class, the subclasses which inherit this class, but not others from the world.

- Example:



Superclass:

```
class Employee
{
    private double salary;
    ..
    public double getSalary() {return salary;}
}
```

Subclass:

This version is OK

```
class Manager extends Employee
{
    private double bonus;
    ..
    public double getSalary() { return super.getSalary() + bonus; }
}
```

This version has error: *The field Employee.salary is not visible*

```
class Manager extends Employee
{
    private double bonus;
    ..
    public double getSalary() { return salary + bonus; }
}
```

- Using "protected" for data fields is considered "against the spirit of OOP".

  Reason - It breaks data encapsulation:

  Instance fields should be treated as implementation details and encapsulated properly.
  Eg. Any change to the field (say, change the name) should not need outsiders (including subclasses) recompile.

Constructors - Review

We can provide zero or more constructors

- If we do not provide any constructor, then a default constructor is automatically generated.
  (a no-argument constructor: fields are default null or zero or false)

- 3 Examples for class Employee
  - Example 1: We write a 3-arguments constructor
  - Example 2: We do not write any constructor (Java automatically generates a no-argument constructor)
  - Example 3: We write three constructors (no argument, 1 argument, 3 arguments)

Example 1: We write a 3-arguments constructor

```
class Employee
{
    private String id;
    private String name;
    private double salary;

    public String toString() { return id + " " + name + " " + salary;}

    //Constructor with 3 arguments
    public Employee(String i, String n, double s)
    {
        id=i;
        name = n;
        salary = s;
    }
}
----------------------------------------------------------------
public static void main(String[] args)
{
    //Employee e0 = new Employee();

    Employee e3 = new Employee("001", "Helena", 1000);
    System.out.println(e3); //001 Helena 1000.0
}
```

Error:
The constructor Employee() is undefined

Example 2: We do not write any constructor

```
class Employee
{
    private String id;
    private String name;
    private double salary;

    public String toString() { return id + " " + name + " " + salary;}

    //not provide any constructor

}
--------------------------------
void main(String[] a…
{
    Employee e = new Employee();
    System.out.println(e.toString()); //null null 0.0
}
```

OK:
Since we don't write any constructor,
Java automatically generates a no-argument constructor

Example 3: We write three constructors

```
class Employee
{
    private String id;
    private String name;
    private double salary;

    public String toString() { return id + " " + name + " " + salary;}

    public Employee()
    {
        id="[-]";
        name = "[new staff]";
        salary = Math.round(Math.random()*1000);
    }

    public Employee(String i)
    {
        id=i;
    }

    public Employee(String i, String n, double s)
    {
        id=i;
        name = n;
        salary = s;
    }
}
----------------------------------------------------------------
public static void main(String[] args)
{
    ...
}
```

```
Employee e0 = new Employee();
System.out.println(e0); //  [-] [new staff] 191.0

Employee e1 = new Employee("001");
System.out.println(e1); // 001 null 0.0

Employee e3 = new Employee("001", "Helena", 1000);
System.out.println(e3); // 001 Helena 1000.0
```

Question:
What if we add the 4th constructor as:

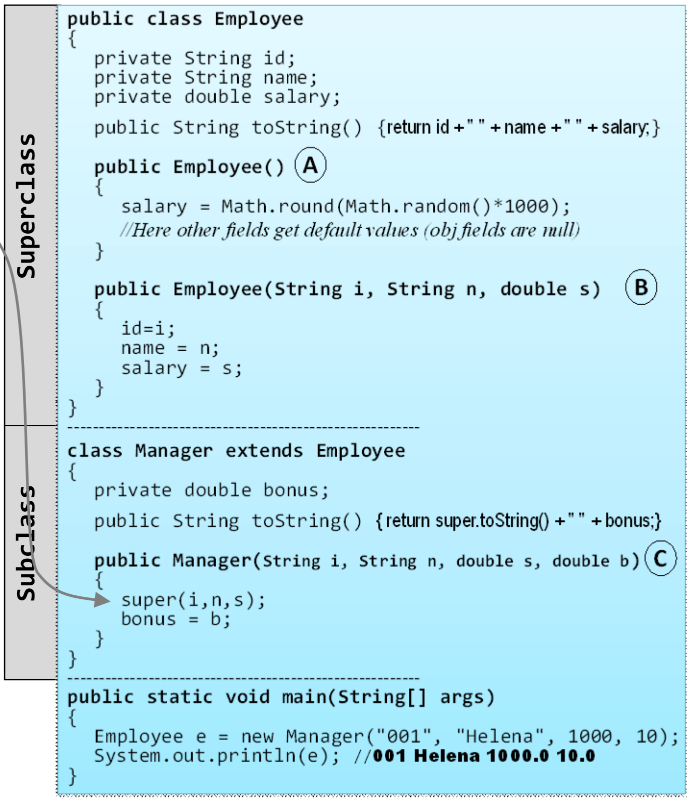
```
public Employee(String nm)
{
    name = nm;
}
```

Anwer:
Error : Duplicate method Employee(String)

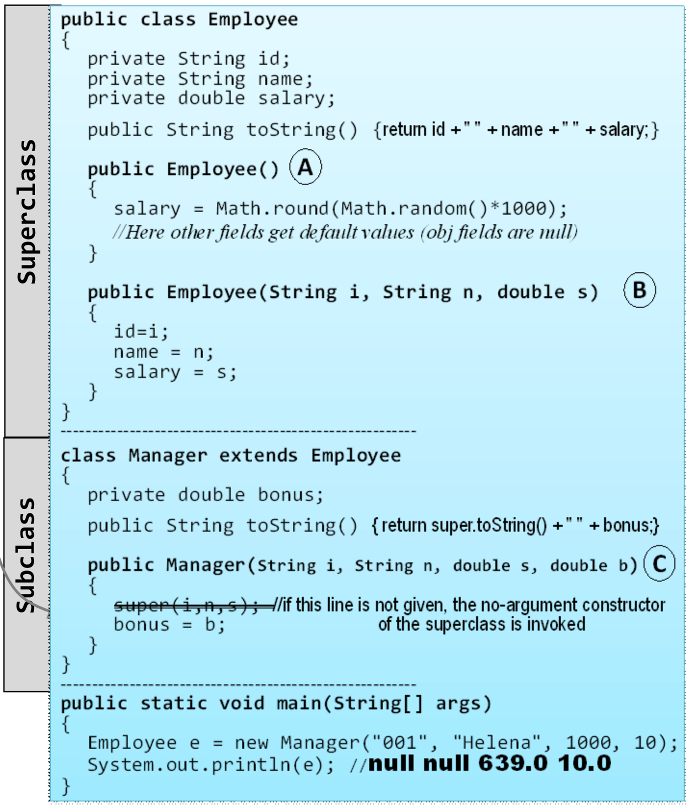
Compiler:
When I see `new Employee("abc")`, I cannot decide which constructor to run.

Constructors - Calling Constructors of superclass

- Constructors are not inherited
- But inside a subclass constructor
  1. We can explicitly call a superclass constructor
     - must be the first statement.
  2. Otherwise the no-argument constructor of the superclass is invoked.



```
public class Employee
{
   private String id;
   private String name;
   private double salary;

   public String toString() {return id + " " + name + " " + salary;}

   public Employee() (A)
   {
      salary = Math.round(Math.random()*1000);
      //Here other fields get default values (obj fields are null)
   }

   public Employee(String i, String n, double s)  (B)
   {
      id=i;
      name = n;
      salary = s;
   }
}
-----------------------------------------------------
class Manager extends Employee
{
   private double bonus;

   public String toString() {return super.toString() + " " + bonus;}

   public Manager(String i, String n, double s, double b) (C)
   {
      super(i,n,s);
      bonus = b;
   }
}
-----------------------------------------------------
public static void main(String[] args)
{
   Employee e = new Manager("001", "Helena", 1000, 10);
   System.out.println(e); //001 Helena 1000.0 10.0
}
```



```
public class Employee
{
   private String id;
   private String name;
   private double salary;

   public String toString() {return id + " " + name + " " + salary;}

   public Employee() (A)
   {
      salary = Math.round(Math.random()*1000);
      //Here other fields get default values (obj fields are null)
   }

   public Employee(String i, String n, double s)  (B)
   {
      id=i;
      name = n;
      salary = s;
   }
}
-----------------------------------------------------
class Manager extends Employee
{
   private double bonus;

   public String toString() {return super.toString() + " " + bonus;}

   public Manager(String i, String n, double s, double b) (C)
   {
      ~~super(i,n,s);~~ //if this line is not given, the no-argument constructor
      bonus = b;                    of the superclass is invoked
   }
}
-----------------------------------------------------
public static void main(String[] args)
{
   Employee e = new Manager("001", "Helena", 1000, 10);
   System.out.println(e); //null null 639.0 10.0
}
```

Question:
What if we now
i. Remove B only?
ii. Remove both A and B?
iii. Remove A only?

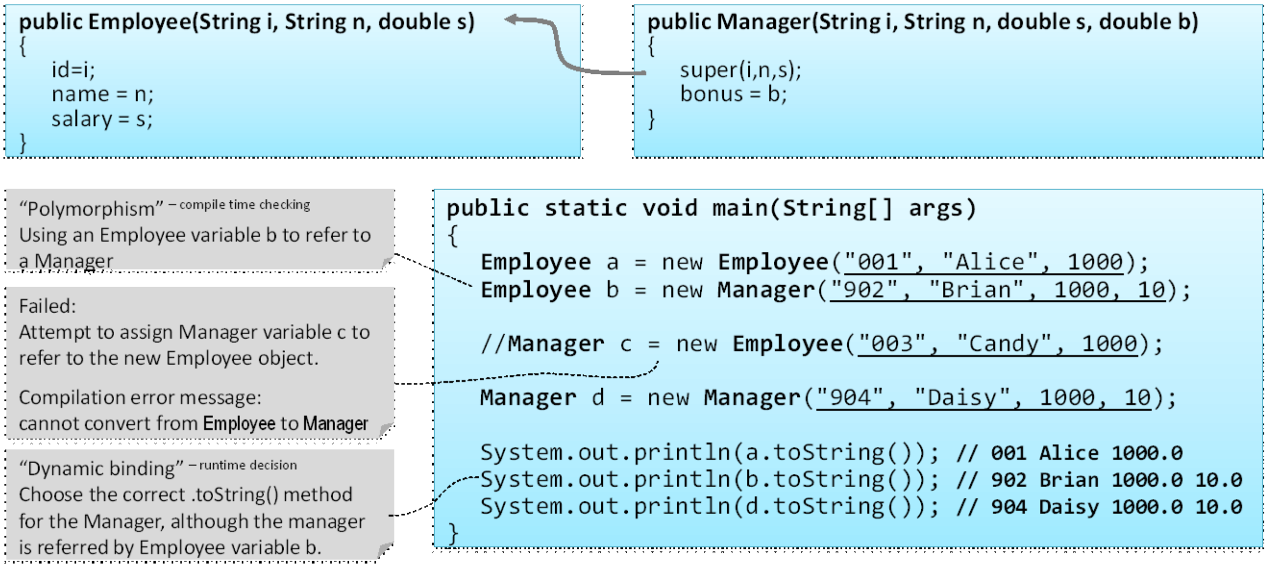
Anwer:
(i) Remove B only: OK
(ii) Remove both A and B: OK
(A no-argument constructor is given automatically for Employee. The salary will be zero)
(iii) Remove A only: - *Implicit super constructor Employee() is undefined. Must explicitly invoke another constructor*

Polymorphism and Dynamic Binding

- **Polymorphism** – An object variable can refer to different actual types. [Compile time checking]
  Superclass Superclass and subclass

- **Dynamic Binding** – Automatically select the appropriate non-static method. [Runtime decision]
  not field

Example: Suppose we have proper constructors:



```
public Employee(String i, String n, double s)
{
    id=i;
    name = n;
    salary = s;
}
```

```
public Manager(String i, String n, double s, double b)
{
    super(i,n,s);
    bonus = b;
}
```

```
public static void main(String[] args)
{
  Employee a = new Employee("001", "Alice", 1000);
  Employee b = new Manager("902", "Brian", 1000, 10);

  //Manager c = new Employee("003", "Candy", 1000);

  Manager d = new Manager("904", "Daisy", 1000, 10);

  System.out.println(a.toString()); // 001 Alice 1000.0
  System.out.println(b.toString()); // 902 Brian 1000.0 10.0
  System.out.println(d.toString()); // 904 Daisy 1000.0 10.0
}
```

"Polymorphism" – compile time checking
Using an Employee variable b to refer to a Manager

Failed:
Attempt to assign Manager variable c to refer to the new Employee object.

Compilation error message:
cannot convert from **Employee** to **Manager**

"Dynamic binding" – runtime decision
Choose the correct .toString() method for the Manager, although the manager is referred by Employee variable b.

Preventing Inheritance: final Classes and Methods

- **Final class and Final method** – Avoid being inherited or redefined

  Example of final class:

```
final class Executive extends Manager
{
    ...
}
```

  Example of final method:

```
class Employee
{
    ...
    public final String getName() {return name;}
}
```

If we redefine .getName() in the Manager class, we get Error:
Cannot override the final method from Employee

- **In Java, the String class is a final class**

  E.g. the following is not allowed.

```
static class StringSubClass extends String
{
    ..
}
```

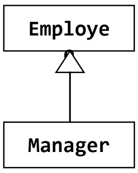
Error: *StringSubClass cannot extend the final class String*

Casting and instanceOf

- **Casting: Consider the type of an object as a different type**
  Note: you are not actually changing the object itself.

Two types of casting:

```
Manager m1 = new Manager("902", "Brian", 1000, 10);
Employee e1 = m1; //upcasting
Manager m;
m = (Manager)e1; //downcasting
System.out.println(m.getBonus());
```



upcasting: label a subclass object reference as a superclass.

- It is done automatically (implicitly): You DO NOT need to add `(Superclass)` for explicit casting.
- Always successful at run time.
- Example of use: a subclass object (like `Manager`) is added as an element in a collection of the superclass (`Employee[]`)

```
    class Employee    {
        private String id;
        private String name;
        private double salary;
        public String toString() {return id + " " + name + " " + salary;}
        .. //constructor and other methods
    }

    class Manager extends Employee {
        private double bonus;
        public String toString() {return super.toString() + " " + bonus;}
        .. //constructor and other methods
    }

    public static void main(String[] args) {
        Employee[] allEmployees;
        allEmployees = new Employee[3];
        allEmployees[0] = new Employee("001", "Alice", 1000);
        allEmployees[1] = new Manager("902", "Brian", 1000, 10); // upcasting
        allEmployees[2] = new Manager("904", "Daisy", 1000, 15); // upcasting

        for (Employee e: allEmployees)
            System.out.println(e);
    }
```

downcasting: label a superclass object reference as a subclass.

- It requires explicit casting: You need to add `(subclass)` for explicit casting.
- Example of use: To use an object 's actual features after its actual type has been temporarily forgotten.

Given an array: **`Employee[] allEmployees`**, each **`allEmployees[i]`** belongs to the **`Employee`** Type
Suppose we know that **`allEmployees[2]`** is actually a **`Manager`**. **We want to run `.getBonus()`**.
However **`allEmployees[2].getBonus()`** won't work because the type of **`allEmployees[2]`** is not **`Manager`**

```
    public static void main(String[] args) {
        Employee[] allEmployees;
        allEmployees = new Employee[3];
        allEmployees[0] = new Employee("001", "Alice", 1000);
        allEmployees[1] = new Manager("902", "Brian", 1000, 10); // upcasting
        allEmployees[2] = new Manager("904", "Daisy", 1000, 15); // upcasting

            Manager m;
            m = (Manager) allEmployees[2]; // downcasting
            System.out.println(m.getBonus());
    }
```

Be careful of casting problem during run time!!

```
class Manager extends Employee
{
      private double bonus;
      public double getBonus() {return bonus;}
      ..
}
--------------------------------------------------------------------------------
public static void main(String[] args)
{
   Employee[] allEmployees;
   allEmployees = new Employee[3];
   allEmployees[0] = new Employee("001", "Alice", 1000);
   allEmployees[1] = new Manager("902", "Brian", 1000, 10); //upcasting
   allEmployees[2] = new Manager("904", "Daisy", 1000, 15); //upcasting

   for (Employee e: allEmployees)
   {
         Manager m;
         m = (Manager)e;  //Runtime error!!!  Alice is not a manager.  (Program stops running
         System.out.println(m.getBonus());
   }
}
```

Runtime Error:
Employee cannot be cast to Manager at Main.main(Main.java:99)

Solution:
May use the instanceof operator to check the class first:

```
for (Employee e: allEmployees)
{
   if (e instanceof Manager)
   {
      Manager m;
      m = (Manager)e;
      System.out.println(m.getBonus());
   }
}
```

Output:
```
10.0
15.0
```

It works now. ☺

- Use of **instanceOf**

  - An object variable is declared with a type, eg. **Employee e;**
    Then the object variable can refer to an object of the type, or its subclass.

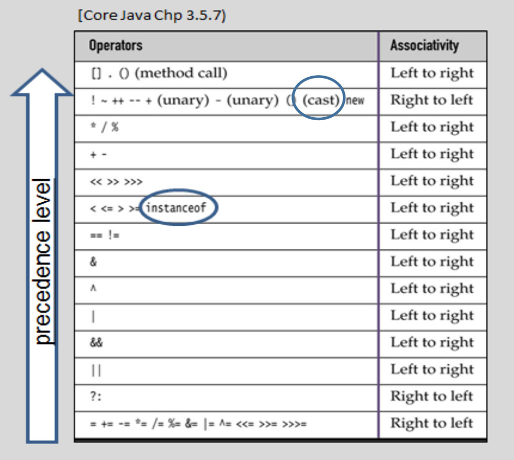
  - The **instanceof** operator compares an object to a class.
    Syntax: **<u>x instanceof C</u>**
    where **<u>x</u>** is an object and **<u>C</u>** is a class

```
for (Employee e: allEmployees)
{
   if (e instanceof Manager)
   {
      Manager m;
      m = (Manager)e;
      System.out.println(m.getBonus());
   }
}
```

Try:
```
System.out.println(e instanceof Manager);
System.out.println(e instanceof Employee);
```

Output:
```
true
true
```

- Where are `instanceof` and `cast` among other operators?


precedence level

[Core Java Chp 3.5.7)

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

- Note: `instanceof` **is often not needed**

```
for (Employee e: allEmployees)
    System.out.println(e.getSalary());
```

Beginners often use `instanceof` in an unnecessary way:
e.g. make decision about using which version of `.getSalary()`

As a proper practice, we should often let JAVA select the appropriate (redefined) method at run-time (Dynamic Binding). You will know that this follows important OO principle (e.g. OCP: Open-Close Principle / Lab06)

Abstract Classes

- **Abstract method**
  - ❑ a method with the `abstract` keyword;
  - ❑ no implementation
  - ❑ It acts as placeholders for *concrete* (i.e. nonabstract) methods that are implemented in the subclasses.

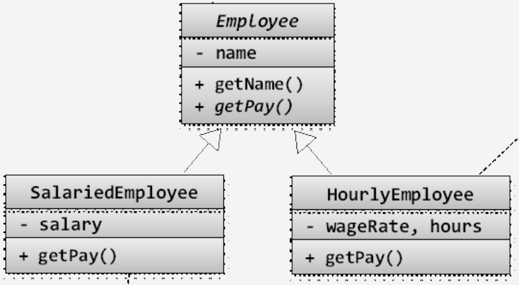
```
public abstract class Employee
{
    private String name;
    public abstract double getPay( );
    public Employee(String aName)  {name = aName;}
    public String getName( ) {return name; }
}
```

- **Abstract classes**
  - ❑ Abstract classes cannot be instantiated. i.e., we cannot create new object using an abstract class.
  - ❑ We can declare an object variable whose type is an abstract class.
  - ❑ Then the object variable can refer to an object of its *concrete* (i.e. nonabstract) subclass.
  - ❑ A class which has one or more abstract methods must be declared abstract.

An abstract superclass class

i.e. it contains abstract method(s)

```
public abstract class Employee
{
   private String name;
   public Employee(String n) {name = n;}
   public String getName( ) {return name;}
   public abstract double getPay( );
}
```



```
public class SalariedEmployee extends Employee
{
   private double salary; //annual

   public SalariedEmployee(String n, double s)
   {
      super(n); salary = s;
   }
   public double getPay( ) {return salary/12;}
}
```

A concrete subclass:

i.e. all methods (getPay(), getName() are concrete)

```
public class HourlyEmployee extends Employee
{
   private double wageRate;
   private double hours; //for the month

   public HourlyEmployee(String n, double w, double h)
   {
      super(n); wageRate = w;  hours = h;
   }
   public double getPay( ) {return wageRate*hours;}
}
```

A concrete subclass:

i.e. all methods (getPay(), getName() are concrete)

- **Abstract object variable and concrete object:**

Declare object variables whose types are abstract (here `arr[0..2]`,`e`)

Then `arr[0..2]` and `e` can refer to objects of the *concrete* subclasses (i.e. nonabstract ones).

```
public static void main(String[] args)
{
   Employee[] arr = new Employee[3];
   arr[0] = new HourlyEmployee("Helena",52.5,30);
   arr[1] = new SalariedEmployee("Kit", 15000);
   arr[2] = new HourlyEmployee("Jason", 100,60);

   for (Employee e: arr)
     System.out.println(e.getName()+
       " ($" + e.getPay()+")");
}
```

Output:
```
Helena ($1575.0)
Kit ($1250.0)
Jason ($6000.0)
```

II. Object: The Cosmic Superclass

- **The Java's `Object` class: `java.lang.object`**
  - ❑ Every class automatically "is-a" subclass of the **`Object`** class. (Inheritance)
  - ❑ Every object of every class is of type **`Object`**.

```
Object o1 = new Employee("002", "Jim", 10000);
System.out.println(o1); //002 Jim 10000.0
System.out.println(o1.getClass().toString()); //class Employee
System.out.println(o1 instanceof Object); //true
System.out.println(o1 instanceof Employee); //true
Object o2 = "Hello";
System.out.println(o2); //Hello
System.out.println(o2.getClass().toString()); //class java.lang.String
System.out.println(o2 instanceof Object); //true
System.out.println(o2 instanceof Employee); //false
```

  - ❑ Object methods: `equals, toString, getClass, clone` etc..
    - automatically inherited in every class
    - `equals, toString` : we usually need to override appropriately
    - `getClass` : returns a `Class` object which represents the class itself
    - `clone` : returns a copy of an object

- **The `equals` method of the `Object` class:**

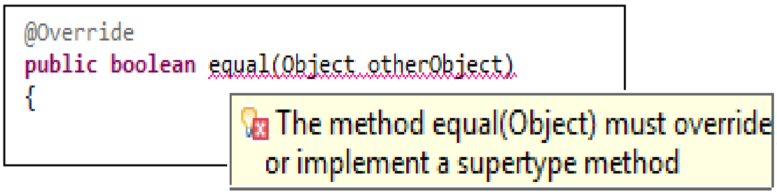
```
//java.lang.object.equals:
public boolean equals(Object obj)
```

- The Right way to **override** `equals` for a class - note the explicit parameter

```
class ClassName
{
   ..
   public boolean equals(Object obj){..}
}
```

Note: To override a method in a subclass, we must give exactly the same signature (method name + parameter list) and return type. To avoid mistake, use the **override annotation**.

- The **override annotation** :
  - ❑ Denotes that the annotated method is required to override a method in the superclass.
  - ❑ Helps us check for misspelling (eg. `equals`), or wrong parameter list etc..



```
@Override
public boolean equal(Object otherObject)
{
```

The method equal(Object) must override or implement a supertype method

But the superclasses do not have .equal(Object)

"Oh! Probably typing mistake!"

- The Right way to **override** equals for a class:

```
class SubjectResult
{
   private String name; //e.g. "Chemistry", "Geography"
   private char grade; //e.g. 'A', 'B', 'C'

   SubjectResult(String n, char g) {name=n; grade=g;}

   @Override
   public boolean equals(Object otherObject)
   {
      if (otherObject == null)
         return false;

      if (this.getClass() != otherObject.getClass())
         return false;

      SubjectResult otherSR = (SubjectResult) otherObject;

      if (!this.name.equals(otherSR.name))
         return false;
      if (this.grade!=otherSR.grade)
         return false;

      return true;
   }
}
```

Use the `@Override` annotation

Parameter: `Object`

Check against `null`

Compare the classes

Cast to our class type

Check the fields one by one
- use `.equals` for object fields
- use == for primitive fields

Call `.equals` of superclass:
`super.equals(otherObject);`

```
SubjectResult s1 = new SubjectResult("Chemistry",'A');
SubjectResult s2 = new SubjectResult("Physics",'A');
System.out.println(s1.equals(s2));//false
```

III Generic Array Lists

- **The Java's Generic `ArrayList`: `java.util.ArrayList`**

Array Sample

```
Integer[] arr;
arr = new Integer[3];

arr[0]= 100;
arr[1]= 101;
arr[2]= 109;

for (int i=0;i<arr.length;i++)
   System.out.println(arr[i]);
```

ArrayList Sample

```
ArrayList<Integer> arrlist;
arrlist = new ArrayList<Integer>();

arrlist.add(100);
arrlist.add(101);
arrlist.add(109);

for (int i=0;i<arrlist.size();i++)
   System.out.println(arrlist.get(i));
```

❑ `java.util.ArrayList` is
a very useful class which is:

- Similar to an array, for storing a collection of object elements
- Automatically adjusts its capacity
- Need a type parameter to specify the type of elements.

  Syntax: `ArrayList<Element_type>`

- Add / retrieve / remove elements: `.add(obj), .get(index), .remove(index)`
- The count of elements: `.size()`

❑ With polymorphism, an ArrayList of super-type can refer to the same type as well as sub-type objects. (e.g. Employee / Manager)

❑ Starting from Java 7, we can omit the type argument when new is used.

ie. `arrlist = new ArrayList<Element_Type>();` 🡸 OK
`arrlist = new ArrayList<>();` 🡸 Also OK. The compiler itself will check what type is needed.

<> is called the diamond syntax

IV Object Wrappers and Autoboxing

- **The Java primitive types and Wrappers**
  - Recall: Java has 8 primitive types:
    `int, long, float, double, short, byte, char, boolean`
  - Each primitive type has a class counterparts which is called a **Wrapper Class**:
    `Integer, Long, Float, Double, Short, Byte, Character, Boolean`
  - Example:

```
Integer x = 3; //Autoboxing
```

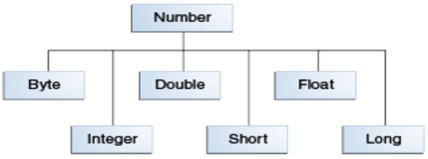
- **Java collections (like `ArrayList`) – require Wrappers**
  - Java collections (like `ArrayList`) can only store object references, not primitive types (`Integer`✓, `int`✗)

```
OK:       ArrayList<Integer>
Invalid:  ArrayList<int>
```

  - **Autoboxing** is done upon getting and setting:

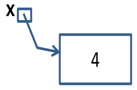
```
ArrayList<Integer> arrlist = new ArrayList<>();
arrlist.add(123); // automatically translated to arrlist.add(Integer.valueOf(123));
```

- **More about Wrapper classes:**
  - The **`Number`** class: the superclass of Integer etc..
  - Wrapper classes are **final**, so we cannot subclass them.
  - Wrapper classes are **immutable** – we cannot change a wrapped value after the wrapper has been constructed.



http://docs.oracle.com/javase/tutorial/java/data/numberclasses.html

```
Integer x = 3;     // Autoboxing
//x.setValue(4);  ✗ there is no method like this (i.e. Integer is immutable)
x = 4;             // This new value, 4, is wrapped to create another Integer object.  x is now set to refer to this new object.
```



New Integer object to wrap the int value of 4

The Number Class

- **All subclasses of** Number **provide the following methods:**
  - Conversion from this object to primitive data
  - Compare this object with another explicit object
  - Equals

Method
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
int compareTo(Byte anotherByte)
int compareTo(Double anotherDouble)
int compareTo(Float anotherFloat)
int compareTo(Integer anotherInteger)
int compareTo(Long anotherLong)
int compareTo(Short anotherShort)
boolean equals(Object obj)

- **Each also provide additional methods for conversion.**

  E.g. The Integer class provide these methods:

Method
static int parseInt(String s) // Returns an integer (decimal only).
String toString() // Returns a String object representing the value of this Integer
static Integer valueOf(int i) // Returns an Integer object holding the value of the specified primitive
static Integer valueOf(String s) // Returns an Integer object holding the value of the given string

V Design Hints for Inheritance

Inheritance - Design Hints:

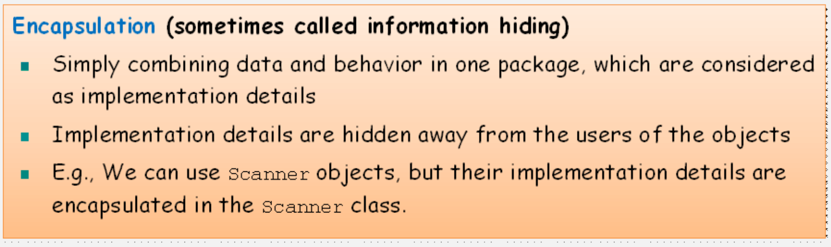
1. Place common operations and fields in the superclass.

2. Don't use protected fields
   - Using "protected" for data fields is considered "against the spirit of OOP".
   - Reason: It breaks data encapsulation:

     Instance fields should be treated as implementation details and encapsulated properly.
     Eg. Any change to the field (say, change the name) should not need outsiders (including subclasses) recompile.

[Topic03]

Recall:



Encapsulation (sometimes called information hiding)

- Simply combining data and behavior in one package, which are considered as implementation details
- Implementation details are hidden away from the users of the objects
- E.g., We can use `Scanner` objects, but their implementation details are encapsulated in the `Scanner` class.

It is a good practice to encapsulate data as **private** instance fields.

1. **Protect the data from corruption by mistake.**
   Outsiders must access the data through the provided public methods
2. **Easier to find the cause of bug**
   Only methods of the class may cause the trouble
3. **Easier to change the implementation**
   - e.g. change from using 3 integers for `year,month,day` to using 1 integer `yyyymmdd`
   - We only need to change the code in the class for that data type
   - The change is invisible to outsiders, hence not affect users.

- However, protected methods can be useful to indicate methods that are not ready for general use and should be redefined in subclasses (eg. `.clone()`)

3. Use inheritance to model the "is-a" relationship. Don't use inheritance unless all inherited fields and methods make sense.
   - Do not extend a class if your intension is just to reuse a portion (<100%) of the superclass.

4. Don't change the expected behavior when you override a method.

5. We should use polymorphism and rely on dynamic binding.
   - Do not explicitly check the type (Ref. Lab06 page 1 Approach 1)

--- end ---

Contents

Topic 05 -Interfaces and Inner Classes

I. Review

Review some key terms:

Visibility (public / protected / private):
- When we implement or redefine a method in the subclass, it must be at least as "visible" as one in the superclass.
- A subclass cannot access the private members in the superclass.

The static keyword

Used to denote fields and methods that belong to a class (but not to any particular object).

The abstract keyword

The **abstract** keyword is applied for **classes** and **nonstatic methods**:

- When applied for a nonstatic **method**: means that we intend to provide **no implementation**; and the implementation will be provided in concrete subclasses.
- When applied for a **class**: means that the class may or may not include abstract methods. Abstract classes **cannot be instantiated** (ie. cannot be used to instantiate any object), but they **can be subclassed**.

- **abstract** is NOT for fields (no matter static or nonstatic)
- **abstract** is NOT for constructors or static methods

Polymorphism – An object variable can refer to different actual types. [compile time checking]
Superclass Superclass and subclass which are concrete

E.g., An object variable (of type A) can refer to objects of various actual types, including type A and its sub-types.

Dynamic Binding – Automatically select the appropriate non-static method. [runtime checking]
Not field!

II. Interface

Java Interface

- **Interface** is a way to describe what classes should do (not how)
  - ❑ Only headings are given to methods [1] (ie. no implementation)
  - ❑ Syntax:

```
interface Interface_Name
{
        /*nonstatic methods, static final fields*/
}
```

- A class can implement an interface (or more than one interfaces)
  - ❑ An implementing class satisfies an interface by implementing the methods given in the interface.
  - ❑ Syntax:

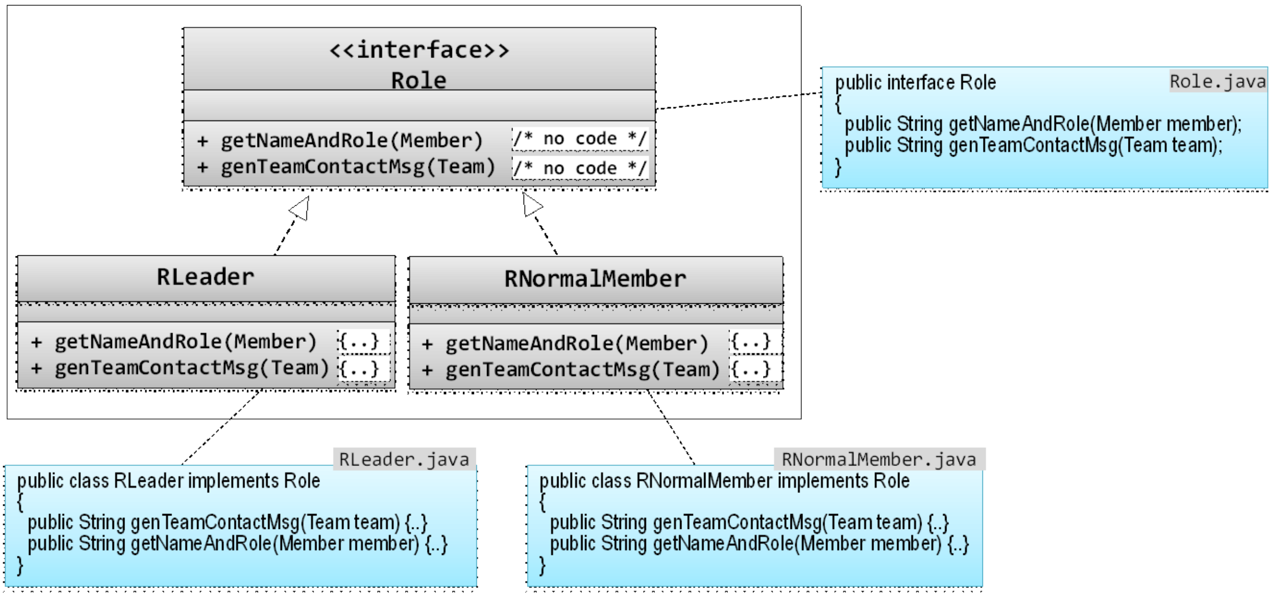
```
class Class_Name implements Interface_Name [, Interface_Name ..]
{
        ..
}
```

[1] For simplicity, here we talk about general non-static methods only.
Java 8 and onwards allow default and static methods which should come with implementation
(http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html)

Example 1 - Working with Member Roles (Lab Exercise Reviewed)

Team.java

```
Role r;
if (roleType=='l') r = new RLeader();
else /*roleType=='n'*/ r = new RNormalMember();
```



<<interface>>
Role
+ getNameAndRole(Member) /* no code */
+ genTeamContactMsg(Team) /* no code */
RLeader
+ getNameAndRole(Member) {..}
+ genTeamContactMsg(Team) {..}
RNormalMember
+ getNameAndRole(Member) {..}
+ genTeamContactMsg(Team) {..}

Role.java

```
public interface Role
{
  public String getNameAndRole(Member member);
  public String genTeamContactMsg(Team team);
}
```

RLeader.java

```
public class RLeader implements Role
{
  public String genTeamContactMsg(Team team) {..}
  public String getNameAndRole(Member member) {..}
}
```

RNormalMember.java

```
public class RNormalMember implements Role
{
  public String genTeamContactMsg(Team team) {..}
  public String getNameAndRole(Member member) {..}
}
```

Example 2 An interface can be implemented by multiple classes A class can implement multiple interfaces

```
interface A {void f1(); void f2();}

interface B {void f3(); void f4();}

class C implements A
{
   public void f1() {..}
   public void f2() {..}
}

class D implements A,B
{
   public void f1() {..}
   public void f2() {..}
   public void f3() {..}
   public void f4() {..}
}

abstract class C2 implements A
{
    public void f1() {..}
}
```

```
public static void main(String[] args)
{
   A a;
   B b;
   C c;
   D d;
   C2 c2;

   //a = new A(); // Cannot instantiate the type A
   //b = new B(); // Cannot instantiate the type B
   c = new C();
   d = new D();
   //x5 = new C2(); // Cannot instantiate the type C2

   a = new C(); // Upcasting is no problem
   b = new D();
   a = c;       // Upcasting is no problem
   b = d;
   c = (C)a; // Downcasting requires explicit cast
   d = (D)b;
}
```

We can use A, B, C, D, C2 as data types. However, only C and D can be instantiated (create object instances)

More considerations:

(1) Two interfaces can contain the same method and implemented in one class

```
interface A2 {void f1(); void f5();}
class X implements A, A2
{
    public void f1() {..}
    public void f2() {..}
    public void f5() {..}
}
```

Both A and A2 have the same void f1()

(2) An abstract class implements an interface

```
abstract class C2 implements A
{
    public void f1() {..}
}
```

Not implement all methods for A, so marked "abstract“.

(3) An interface extends one or more interfaces

```
interface T extends A,B { void f5();}
```

- A class can only extend one class.
- But an interface can extend more interfaces.

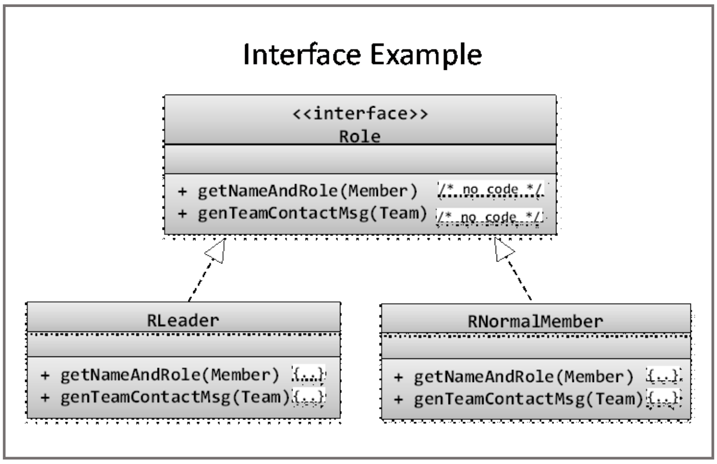
Here interface T has f1(),f2(),f3(),f4(),f5().
T is a sub-interface of both A and B.

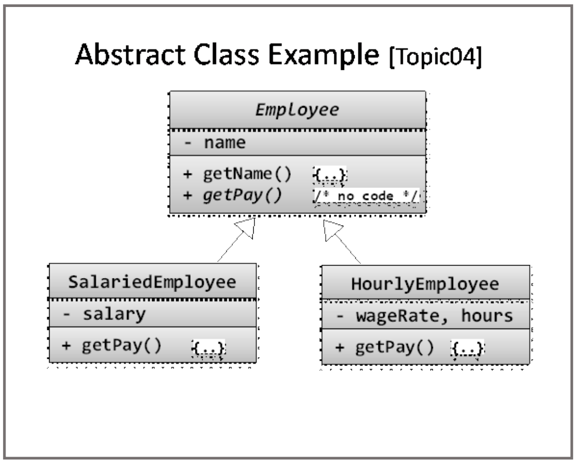
(4) A class can extend a superclass + implement interface(s)

```
class D2 extends D implements T
{
    public void f5() {..}
}
```

```
class Emloyee extends Person
implements I1, I2
{
    ..
}
```

Interface vs Abstract class


Interface Example
<<interface>>
Role
+ getNameAndRole(Member) /* no code */
+ genTeamContactMsg(Team) /* no code */
RLeader
+ getNameAndRole(Member) {...}
+ genTeamContactMsg(Team) {...}
RNormalMember
+ getNameAndRole(Member) {...}
+ genTeamContactMsg(Team) {...}



Similarities

- Cannot instantiate them (ie. cannot create object instances)
- Contain methods which are to be implemented by other classes.

Differences

Abstract class	Interface

A subclass can only inherit one abstract class
Abstract class does not support multiple inheritance.
abstract class A {}
abstract class B {}
class C extends A, B {} ✗
We cannot have multiple superclasses
	A class can implement 1 or more interfaces.
	Interface is a way to approximate multiple inheritance.
	interface A {..}
	interface B {..}
	class C implements A,B{..} ✓

Allow access modifiers (private / protected / public)	All methods are public We do not need to write the keywords *public* , *abstract*. They are implicit for Interfaces.
Can provide shared method code (default behavior) Nonstatic methods can be abstract or non-abstract	No shared method code: Nonstatic methods are abstract Generally speaking[1] - We write headers only
Has constructors	No constructors
Allow various kinds of fields: static or not, final or not	No object fields - Any field defined in an interface is actually treated as static and final

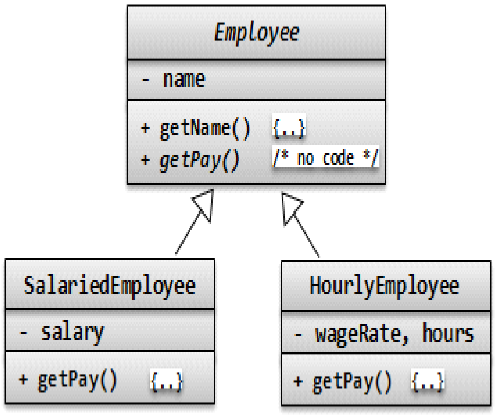
[1] For simplicity, here we talk about general non-static methods only.
Java 8 and onwards allow default and static methods which should come with implementation
(http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html)

Which should you use, abstract classes or interfaces?

[http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html]

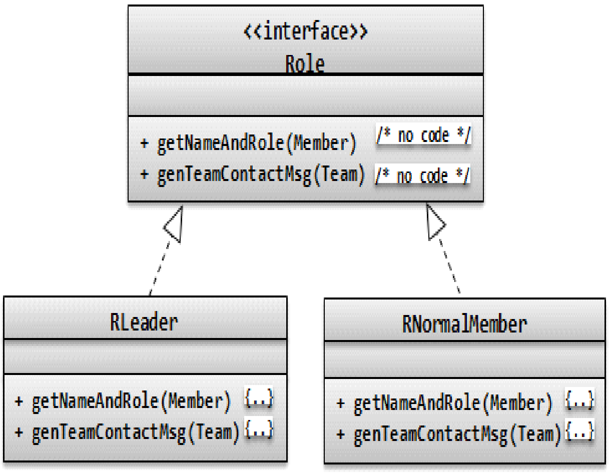
Consider using **abstract classes** for any point below:

- Want to **share code** among several closely related classes.
- Expect that subclasses have **many common methods or fields**, or require non-public access modifiers such as **protected and private**.
- Want to declare useful **object fields**. So that methods can access and modify the state of the object to which they belong.



Consider using **interfaces** for any point below:

- Expect that **unrelated classes would implement** your interface. For example, the interfaces `Comparable` and `Cloneable` are implemented by many unrelated classes.
- Want to specify the behavior of a particular data type, but **not concerned about who implements** its behavior.
- Want to take advantage of **multiple inheritance of type** (See Example 3 in next page).


<<interface>>
Role
+ getNameAndRole(Member) /* no code */
+ genTeamContactMsg(Team) /* no code */
RLeader
+ getNameAndRole(Member) {..}
+ genTeamContactMsg(Team) {..}
RNormalMember
+ getNameAndRole(Member) {..}
+ genTeamContactMsg(Team) {..}

Example 3 - Grader, Student, Exercise

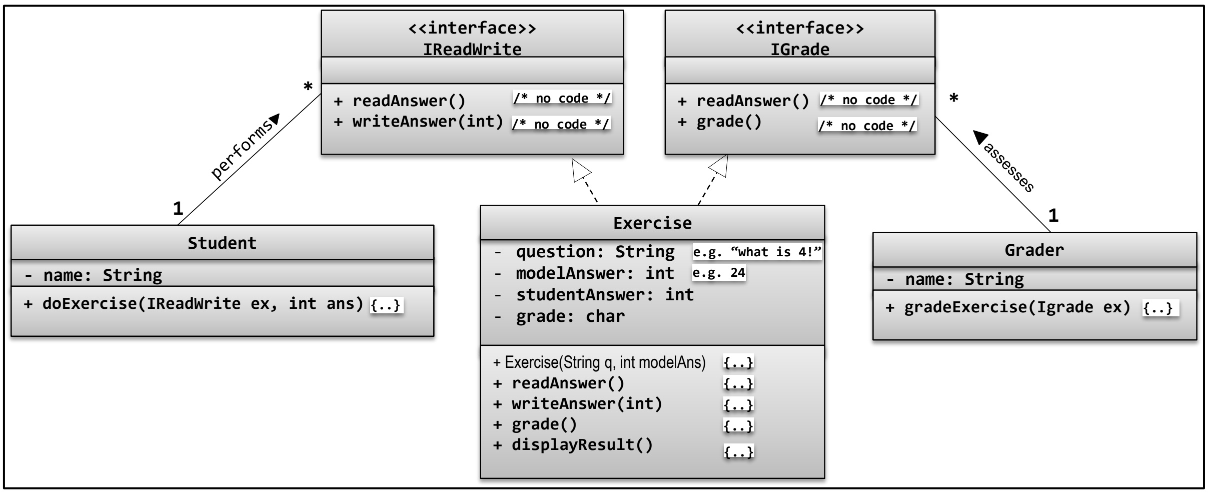
- This example illustrates the **Practical use of "one class with multiple interfaces"**

- *Storyboard: A grader can only read/grade students' exercises, while a student can only read/write the exercise.*

- You will see that the `Exercise` class implements both the `IGrade` and `IReadWrite` interfaces.

- Goal: To **filter functionalities** so that

different users (here graders and students) who receive the same object (here the exercise object) can use the dedicated functions only.



<<interface>>
IReadWrite
+ readAnswer() /* no code */
+ writeAnswer(int) /* no code */
<<interface>>
IGrade
+ readAnswer() /* no code */
+ grade() /* no code */
*
performs
1
*
assesses
1
Student
- name: String
+ doExercise(IReadWrite ex, int ans) {..}
Exercise
- question: String e.g. "what is 4!"
- modelAnswer: int e.g. 24
- studentAnswer: int
- grade: char
+ Exercise(String q, int modelAns) {..}
+ readAnswer() {..}
+ writeAnswer(int) {..}
+ grade() {..}
+ displayResult() {..}
Grader
- name: String
+ gradeExercise(Igrade ex) {..}

```
class Student
{
    private String name;
    public Student(String n) {name=n;}
    public void doExercise(IReadWrite x, int ans)
    {
        x.writeAnswer(ans);
    }
}
```

```
class Grader
{
    private String name;
    public Grader(String n) {name=n;}
    public void gradeExercise(IGrade x)
    {
        x.grade();
    }
}
```

```
interface IReadWrite
{
    void readAnswer();
    void writeAnswer(int anAnswer);
}
```

```
interface IGrade
{
    void readAnswer();
    void grade();
}
```

```
class Exercise implements IGrade, IReadWrite
{
    private int studentAnswer;
    private char grade;
    private final String question;
    private final int modelAnswer;
    public Exercise(String q, int a)
    {
        question = q; modelAnswer=a;
    }
    public void writeAnswer(int anAnswer)
    {
        studentAnswer=anAnswer;
    }
    public void readAnswer()
    {
        System.out.println(
             "Student's answer is "+ studentAnswer);
    }
    public void grade()
    {
        if (studentAnswer==modelAnswer) grade='A';
        else grade='F';
    }
    public void displayResult()
    {
        System.out.println(
             "Student's answer is "+studentAnswer+
             ", grade is: "+grade);
    }
}
```

```
public static void main(String[] args)
{
    Exercise ex = new Exercise("What is 4!", 24);
    Student m = new Student("Mary");
    Grader h = new Grader("Helena");
    m.doExercise(ex,24);
    h.gradeExercise(ex);
    ex.displayResult();
}
```

Output:

```
Student's answer is 24,
grade is: A
```

The Java Comparable Interface (sorting)

JAVA provides **sorting** methods for **comparable objects**

1) Arrays : **`Arrays.sort(`*`array`*`);`**

2) Collections (e.g. ArrayList) : **`Collections.sort(`*`array_list`*`);`**

Before using the above, we have to solve for some issues:

> ***"Nobody knows that trees and plants are to be sorted and, even if we are told, but how to sort them? ... 😐 "***

Correspondingly, in JAVA we have to

①tell that the objects are to be sorted, and ②decide how to *compare* them in sorting.

These are what we do in JAVA:

- the class should implement the interface: `java.lang.Comparable<`*`type`*`>`
- this `Comparable` interface has a method to be implemented: `int compareTo(`*`type`*` another)`

```
Return value:
0 if equal
1 if this is larger than another
-1 if this is smaller than another
```

Example: Employees ordered by salaries

```
class Employee implements Comparable<Employee>
{
    private final String name;
    private double salary;
    private final Day hireDay;
    ..
    @Override
    public int compareTo(Employee another)
    {
        if (this.salary==another.salary) return 0;
        else if (this.salary>another.salary) return 1;
        else return -1;
    }
}
```

```
class Employee implements Comparable<Employee>
{
    ..
    @Override
    public int compareTo(Employee e2) {
        .. //check this.salary and e2.salary
    }
}
```

```
public static void main(String[] args)
{
    /* sort an array of employees */
    Employee[] arr = new Employee[3];
    arr[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
    arr[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
    arr[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
    Arrays.sort(arr);
    for (Employee e : arr)
        System.out.println(e);

    /* sort an arraylist of employees */
    ArrayList<Employee> arrlist = new ArrayList<>();
    arrlist.add(arr[2]);arrlist.add(arr[0]);arrlist.add(arr[1]);
    Collections.sort(arrlist);
    for (Employee e : arrlist)
        System.out.println(e);
}
```

Output:

```
name=Tony Tester,salary=40000.0,hireDay=15 Mar 1990
name=Harry Hacker,salary=50000.0,hireDay=1 Oct 1989
name=Carl Cracker,salary=75000.0,hireDay=15 Dec 1987
name=Tony Tester,salary=40000.0,hireDay=15 Mar 1990
name=Harry Hacker,salary=50000.0,hireDay=1 Oct 1989
name=Carl Cracker,salary=75000.0,hireDay=15 Dec 1987
```

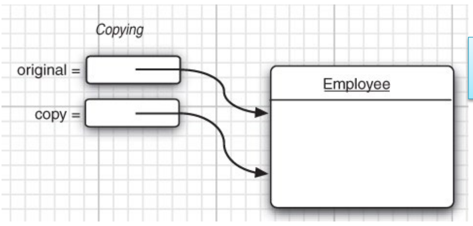
The Java Cloneable Interface (copying)

Introduction to Cloning:

To **clone** an object, it means to make a new copy of the object. **Different from copying**!!!

Copying an object variable vs **Cloning** an object

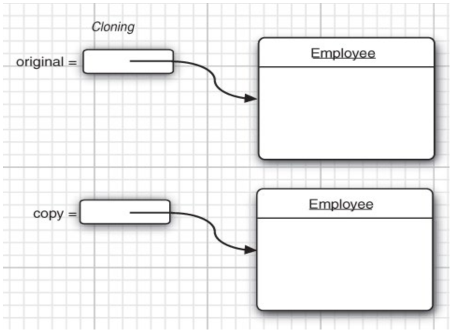
copying :



```
Employee original, copy;
original  = new Employee (..);
copy = original;
```

`= : copying a variable`

cloning:

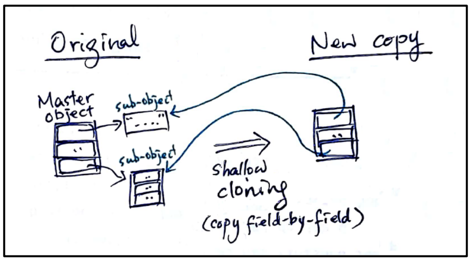


```
Employee original, copy;
original  = new Employee (..);
copy = original.clone();
```

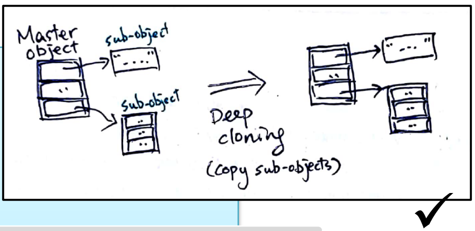
`The clone method returns a new copy`

To make an object cloneable, we need to

- Make the class **implements java.lang.Cloneable**
- Redefine the method : **`public type clone()`**
  - The **`Object` class** provides **`protected Object clone()`** which **copies field-by-field** (**Shallow-cloning**)
    If a field is a reference, it only copies the reference, that **refers to the same subobject**



  - We redefine the clone() method to handle **cloning of mutable subobjects**



✓

```
class Employee implements Comparable<Employee>, Cloneable
{
   ..
   @Override
   public Employee clone() throws CloneNotSupportedException
   {
      Employee copy = (Employee) super.clone();

      copy.hireDay = new Day(
         this.hireDay.getYear(),
         this.hireDay.getMonth(),
         this.hireDay.getDay());

      copy.name = new String(this.name);
      return copy;
   }
}
```

`Call the Object superclass's clone() method`

`Construct a copy for this.hireDay, or call .clone of this.hireDay (if Day.clone is available)`

Actually can be omitted.
Reason: Since strings are immutable, it is Okay to let both original and copy refer to the same string.

Why okay? Well, if one changes the name, the *change* is actually to create a new string object. [Ref. Lec05_Ex.pdf]

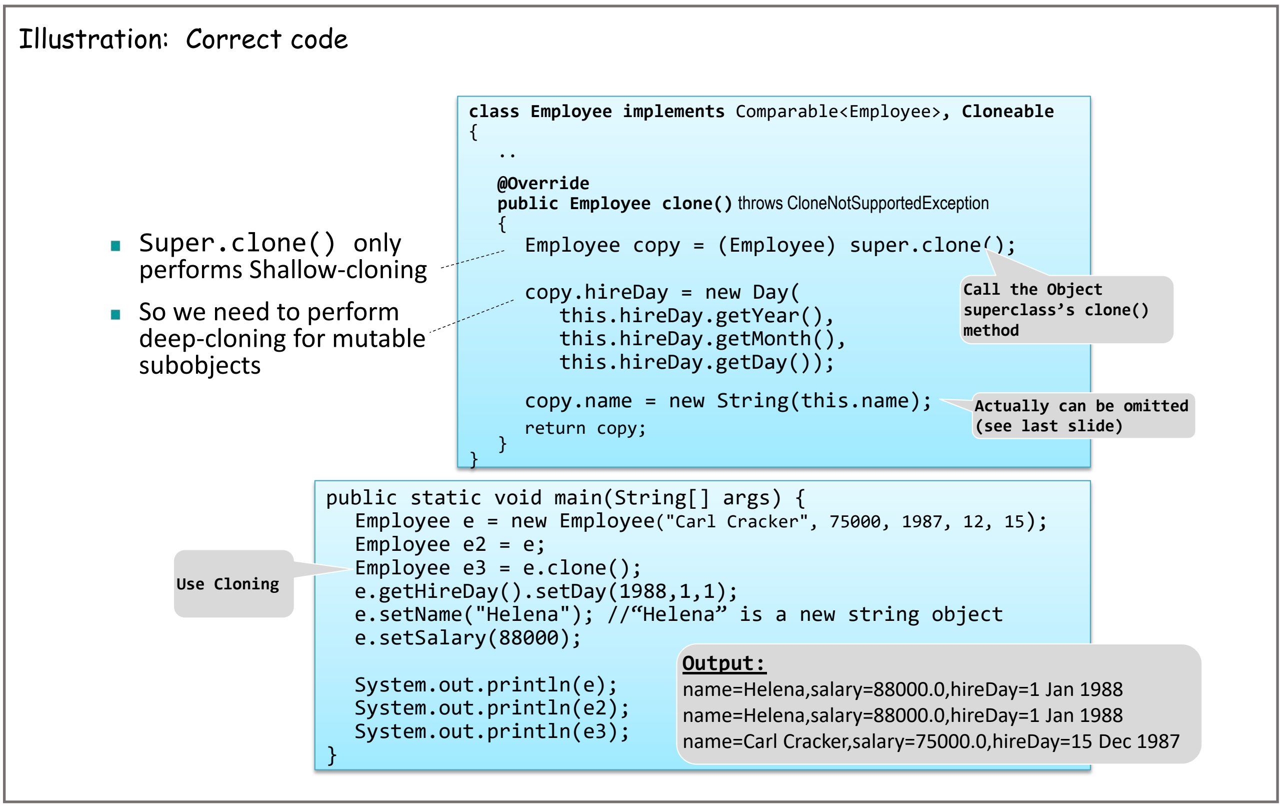


Illustration: Correct code

- `Super.clone()` only performs Shallow-cloning
- So we need to perform deep-cloning for mutable subobjects

```
class Employee implements Comparable<Employee>, Cloneable
{
    ..
   @Override
   public Employee clone() throws CloneNotSupportedException
   {
     Employee copy = (Employee) super.clone();

     copy.hireDay = new Day(
        this.hireDay.getYear(),
        this.hireDay.getMonth(),
        this.hireDay.getDay());

     copy.name = new String(this.name);
     return copy;
   }
}
```

Call the Object superclass’s clone() method

Actually can be omitted (see last slide)

Use Cloning

```
public static void main(String[] args) {
   Employee e = new Employee("Carl Cracker", 75000, 1987, 12, 15);
   Employee e2 = e;
   Employee e3 = e.clone();
   e.getHireDay().setDay(1988,1,1);
   e.setName("Helena"); //“Helena” is a new string object
   e.setSalary(88000);

   System.out.println(e);
   System.out.println(e2);
   System.out.println(e3);
}
```

Output:

```
name=Helena,salary=88000.0,hireDay=1 Jan 1988
name=Helena,salary=88000.0,hireDay=1 Jan 1988
name=Carl Cracker,salary=75000.0,hireDay=15 Dec 1987
```

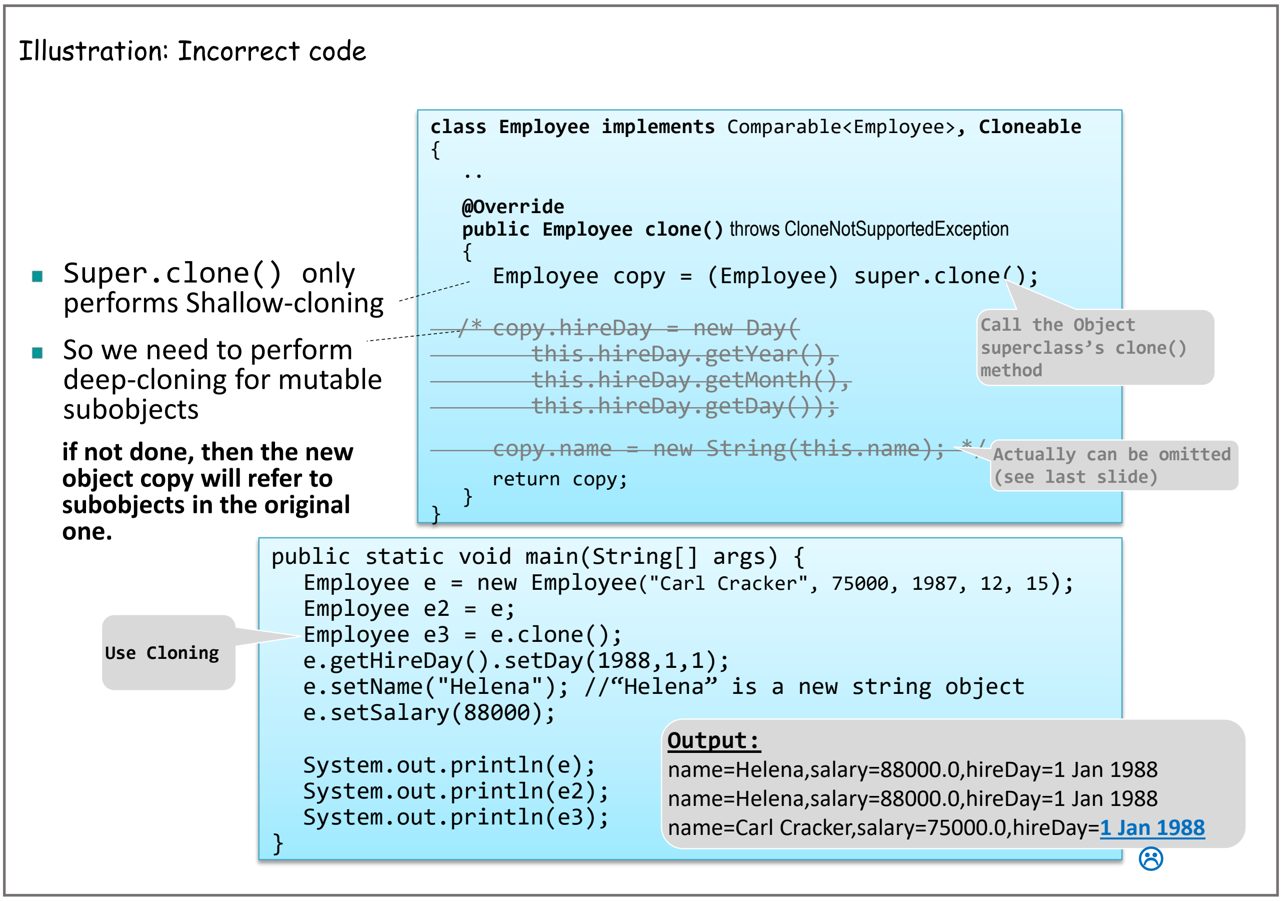


Illustration: Incorrect code

- `Super.clone()` only performs Shallow-cloning
- So we need to perform deep-cloning for mutable subobjects

if not done, then the new object copy will refer to subobjects in the original one.

```
class Employee implements Comparable<Employee>, Cloneable
{
    ..
   @Override
   public Employee clone() throws CloneNotSupportedException
   {
     Employee copy = (Employee) super.clone();

     ~~/* copy.hireDay = new Day(~~
        ~~this.hireDay.getYear(),~~
        ~~this.hireDay.getMonth(),~~
        ~~this.hireDay.getDay());~~

     ~~copy.name = new String(this.name); */~~
     return copy;
   }
}
```

Call the Object superclass’s clone() method

Actually can be omitted (see last slide)

Use Cloning

```
public static void main(String[] args) {
   Employee e = new Employee("Carl Cracker", 75000, 1987, 12, 15);
   Employee e2 = e;
   Employee e3 = e.clone();
   e.getHireDay().setDay(1988,1,1);
   e.setName("Helena"); //“Helena” is a new string object
   e.setSalary(88000);

   System.out.println(e);
   System.out.println(e2);
   System.out.println(e3);
}
```

Output:

```
name=Helena,salary=88000.0,hireDay=1 Jan 1988
name=Helena,salary=88000.0,hireDay=1 Jan 1988
name=Carl Cracker,salary=75000.0,hireDay=1 Jan 1988
```

☹

III Inner Classes

Inner Class - Introduction

- An *inner class* is a class defined within its *outer class*.
- Often used as *helping classes*
- Advantage of using Inner class:
  1) Better organization of code:

     The helper class is contained inside the outer class, rather than written separately.
  2) Access across inner/outer classes:
     (a) The outer class have access to the inner class's methods and nonstatic fields[1] (even if they are private).
     (b) The inner class has access to the outer class's methods and fields (even if they are private).

```
public class OuterClass
{
    private class InnerClass
    {
        .. Fields and methods of InnerClass
    }
    .. Fields and methods of OuterClass
}
```

----

[1] static field cannot exist in a nonstatic inner class, unless initialized with a constant expression

```
private final static int testing=3; //ok
private static int testing=3; //not allowed
```

Inner Class Example 1.

Recall:

"The outer class have access to the inner class's methods and nonstatic fields (even if they are private)"

```
class BankAccount
{
  private class Money
  {
    private String currency; //e.g. "HKD", "RMB", "NTD", "JPY", "KRW", "USD", "GBP"
    private double value;

    public Money(String c, double b) {currency=c; value=b;}

    @Override
    public String toString() {return currency+" "+value;}
  }

  private Money balance;

  public BankAccount(String currency)
  {
    balance = new Money(currency, 0.00);
  }

  public String getBalance()
  {
    return balance.toString();
  }

  public void addMoney(double incr)
  {
    balance.value += incr;
  }
}
```

Outer class

Inner class

The outer class uses the inner class to define an object field

- Use a constructor of the inner class
- Use a method of the inner class
- Access a field of the inner class

An object of the inner class

```
public static void main(String[] args)
{
   BankAccount account =
      new BankAccount("HKD");

   account.addMoney(300);

   System.out.println(
      "Account balance = "
      + account.getBalance());
}
```

Output:

```
Account balance = HKD 300.0
```

Inner Class Example 2.

Recall:

"The inner class has access to the outer class's methods and fields (even if they are private)."

```
class BankAccount
{
  private class Money
  {
    private String currency; //e.g. "HKD", "RMB", "NTD", "JPY", "KRW", "USD", "GBP"
    private double value;

    public Money(String c, double b) {currency=c; value=b;}

    @Override
    public String toString() {return currency+" "+value+" owned by "+owner;}
  }

  private Money balance; private String owner;

  public BankAccount(String currency, String ow)
  {
    balance = new Money(currency, 0.00); owner = ow;
  }

  public String getBalance()
  {
    return balance.toString();
  }

  public void addMoney(double incr)
  {
    balance.value += incr;
  }
}
```

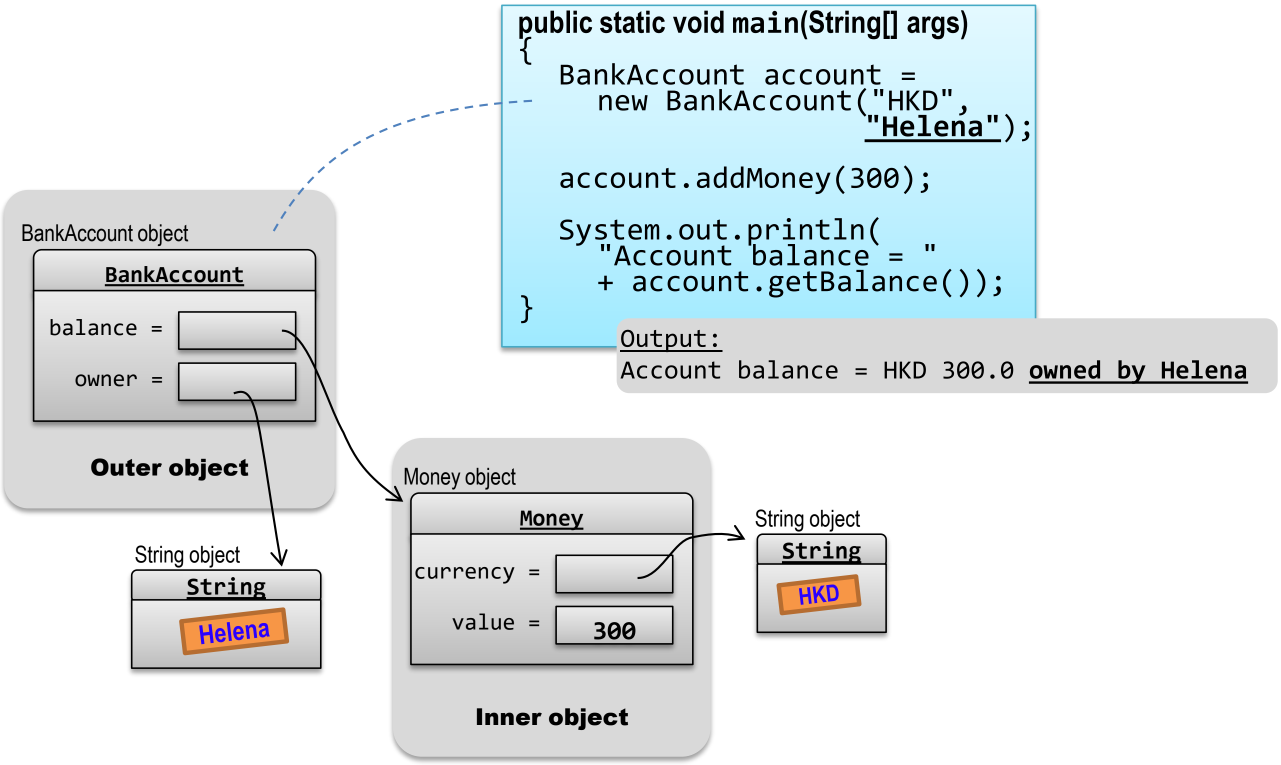
A field of the outer class

Note: we don't write the outer object like:

`outer.owner`

No need

See the **drawing** in next slide for illustration



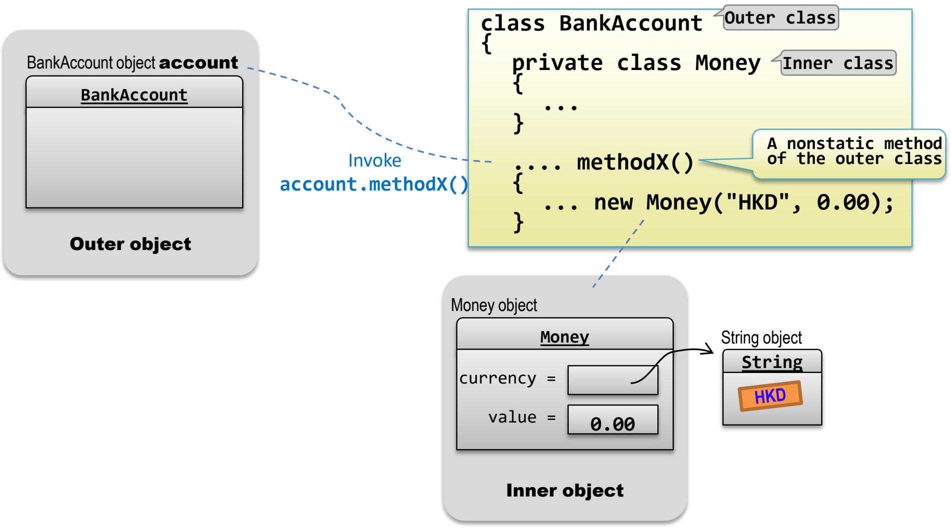
```
public static void main(String[] args)
{
   BankAccount account =
      new BankAccount("HKD",
                      "Helena");

   account.addMoney(300);

   System.out.println(
      "Account balance = "
      + account.getBalance());
}
```

```
Output:
Account balance = HKD 300.0 owned by Helena
```

Association between inner class object and outer object

E.g. In a nonstatic method (**`methodX`**) of the outer class, we create an inner object.

The *implicit parameter* (*`this`*, or known as the *calling object*) of the call to **`methodX`**, is then the outer object of the created inner object.



```
class BankAccount
{
  private class Money
  {
    ...
  }

  .... methodX()
  {
    ... new Money("HKD", 0.00);
  }
```

Note: **`methodX`** can mean the constructor of the outer-class, i.e., the one in example 2 (last page).

More details about Inner Class (For interested students only)

- An inner class can be
  - Nonstatic, like our example :
    - Nonstatic inner class object must arise from an outer class object.
    - Has a connection between an outer class object and the inner class object.
    - Nonstatic inner class must not have static members.
  - Static

    [We do not go into details. Interested students may read Core Java Chp06 / Absolute Java Chp13]

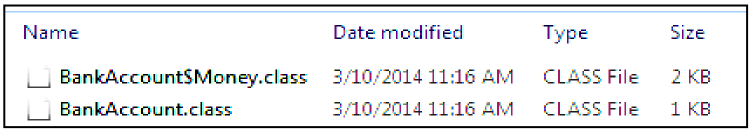
    Static: No connection between outer class object and inner class object. (eg. inner class object created in a static method of the outer class)

```
class BankAccount                          Outer class
{
  private class Money                      Inner class
  {
     private String currency;
     private double value;

     public Money(String c, double b) {..}

     @Override
     public String toString() {return ..;}
  }

  private Money balance;

  public BankAccount(String currency)
  {
     balance = new Money(currency, 0.00);
  }

  public String getBalance()
  {
     return balance.toString();
  }

  public void addMoney(double incr)
  {
     balance.value += incr;
  }
}
```

```
public class OuterClass
{
  private static class InnerClass
  {
     .. Fields and methods of InnerClass
  }
  .. Fields and methods of OuterClass
}
```

Other Facts

- Each inner class gets its own .class file:



Name	Date modified	Type	Size
BankAccount$Money.class	3/10/2014 11:16 AM	CLASS File	2 KB
BankAccount.class	3/10/2014 11:16 AM	CLASS File	1 KB

- Visibility of Inner Class
  - An inner class can be private, like example 2
  - An inner class can be public.
    If so, it can be used outside the outer class.
    [We do not go into details. For interested students only]

```
BankAccount.Money amount;
amount = account.new Money("USD",123);
System.out.println(amount.toString());
```

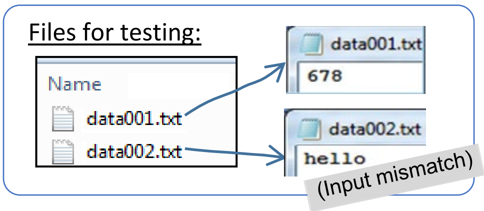
- Interesting variations:
  - Nested inner classes
  - Anonymous class
    (want one object only, lazy to give class name; created using new in a method, as an inner class)
  - When a class inherits an outer class, the inner class is also inherited.
    [We do not go into details. Interested students may read Core Java Chp06 / Absolute Java Chp13]

--- end ---

Contents

Topic 06 -Exception Handling

I Introductory Examples



Requirement: **Read one positive integer from a file**

Example 1 - Problems checked by Java Virtual Machine

```
public static void main(String[] args) throws FileNotFoundException
{
   Scanner in = new Scanner(System.in);
   System.out.print("Input the file pathname: ");
   String fname = in.next();

[Line 13]   Scanner inFile = new Scanner(new File(fname));
[Line 14]   int x = inFile.nextInt();
   System.out.println("Data is: "+x);
   inFile.close();

   in.close();
}
```

```
Rundown 1.1:
Input the file pathname: c:\data001.txt
Data is: 678
```

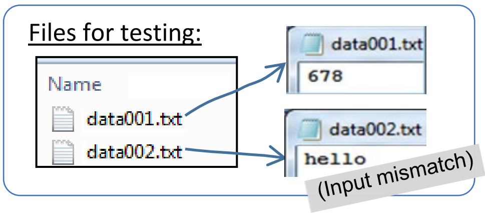
```
Rundown 1.2:
Input the file pathname: c:\data02.txt
Exception in thread "main" java.io.FileNotFoundException
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(Unknown Source)
    at java.util.Scanner.<init>(Unknown Source)
    at Main.main(Main.java:13)
```

```
Rundown 1.3:
Input the file pathname: c:\data002.txt
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at Main.main(Main.java:14)
```

- Problems are checked by **Java Virtual Machine**
- java.io.FileNotFoundException and java.util.InputMismatchException are JAVA classes, each means a type of exception.
- JVM outputs the message by calling:

  `public void printStackTrace()`

  [a method of the `Java.lang.Throwable` class]



Files for testing:

Name
data001.txt
data002.txt

data001.txt
```
678
```

data002.txt
```
hello
```

(Input mismatch)

Example 2 - We check and handle the problems ourselves

Below shows how that can be done, using try-catch blocks

Program:

```
public static void main(String[] args)
{
    try
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Input the file pathname: ");
        String fname = in.next();
        Scanner inFile = new Scanner(new File(fname));
        int x = inFile.nextInt();
        System.out.println("Data is: "+x);

        inFile.close();
        in.close();
    }
    catch (FileNotFoundException e)
    {
        System.out.println("Cannot open the file. Please check or ask CS2312 helpers.“);
    }
    catch (InputMismatchException e)
    {
        System.out.println("Cannot read the required number from the opened file. "+
            " Please download from Helena's website again.");
    }
}
```

- A Try block tells what to do when everything goes smoothly.
- A catch block handles a kind of caught problems (exceptions)

```
catch (InputMismatchException e)
```

Declare the exception type (Data type) and exception object variable (other names are ok, but “e” is often used)

Testing:

```
Rundown 2.1:
Input the file pathname: c:\data001.txt
Data is: 678
```

```
Rundown 2.2:
Input the file pathname: c:\data02.txt
Cannot open the file.  Please check or ask
CS2312 helpers.
```

```
Rundown 2.3:
Input the file pathname: c:\data002.txt
Cannot read the required number from the
opened file.  Please download from Helena's
website again.
```

```
try
{
  ..
}
catch (FileNotFoundException e)
{
  System.out.println("Cannot open ..“);
}
catch (InputMismatchException e)
{
  System.out.println("Cannot read ..”);
}
```

We write code to check and take action (here simply output the situation)

A common case: we handle more processing:

1) Put the actions in one method. The actions may have problems.
2) Place the *try-catch* block in the caller. The caller will handle the problems.

Example 3

```
public static void processFile(String fname) throws FileNotFoundException, InputMismatchException
{
   Scanner inFile = new Scanner(new File(fname));
   int x = inFile.nextInt();
   System.out.println("Data is: "+x);
   inFile.close();
}

public static void main(String[] args)
{
   try
   {
       Scanner in = new Scanner(System.in);
       System.out.print("Input the file pathname: ");
       String fname = in.next();
       processFile(fname);
       in.close();
   }
   catch (FileNotFoundException e)
   {
       System.out.println("Cannot open the file.  Please check or ask CS2312 helpers.");
   }
   catch (InputMismatchException e)
   {
       System.out.println("Cannot read the required number from the opened file.  Please download from Helena's website again.");
   }
}
```

The throws clause – declare what exceptions might occur.

Rundown: [Same as 2.1, 2.2, 2.3 in Example 2]

We can write an Exception Controlled Loops
- Let the user get things right on a subsequent try

Example 4

```
public static void processFile(String fname) throws FileNotFoundException, InputMismatchException
{
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    System.out.println("Data is: "+x);
    inFile.close();
}

public static void main(String[] args)
{
    Scanner in = new Scanner(System.in);
    boolean shouldEnd=false;
    while (!shouldEnd)
    {
       try
       {
          System.out.print("Input the file pathname: ");
          String fname = in.next();
          processFile(fname);
          shouldEnd=true;
       }
       catch (FileNotFoundException e)
       {
          System.out.println("Cannot open the file.");
          System.out.print("Try another file?  Type your choice [y/n]: ");
          shouldEnd=(in.next().charAt(0)=='n');
       }
       catch (InputMismatchException e)
       {
          System.out.println("Cannot read the required
          System.out.print("Try another file?  Type yo
          shouldEnd=(in.next().charAt(0)=='n');
       }
    }
    in.close();
}
```

Rundown 4.1:

```
Input the file pathname: c:\data002.txt
Cannot read the required number from the opened file.
Try another file?  Type your choice [y/n]: y
Input the file pathname: c:\data02.txt
Cannot open the file.
Try another file?  Type your choice [y/n]: y
Input the file pathname: c:\data001.txt
Data is: 678
```

Rundown 4.2:

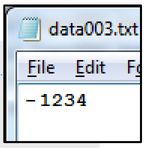
```
Input the file pathname: c:\data002.txt
Cannot read the required number from the opened file.
Try another file?  Type your choice [y/n]: y
Input the file pathname: c:\data02.txt
Cannot open the file.
Try another file?  Type your choice [y/n]: n
```

Create exception class + throw an exception object of the kind

- Suppose we expect a non-negative integer, however the file contains a –ve integer.
  This is what we want:

If the file content is –ve, the output should tell:

```
Rundown 5.1:
Input the file pathname: c:\data003.txt
Unexpected negative value from the file.
Try another file?  Type your choice [y/n]: n
```



```
-1234
```

- This problem is not described by JAVA standard exception classes.
  Therefore we create and use our own exception class.

Example 5

```
public class NegativeIntegerException extends Exception
{
        public NegativeIntegerException()
        {
                super("Negative integer!");
        }

        public NegativeIntegerException(String message)
        {
                super(message);
        }
}
```

It is customary to give both a default constructor and a constructor that contains a detailed message.

processFile():

```
public static void processFile(String fname) throws FileNotFoundException,
                InputMismatchException, NegativeIntegerException
{
        Scanner inFile = new Scanner(new File(fname));

        int x = inFile.nextInt();
        if (x<0) {
                throw new NegativeIntegerException();
        }

        System.out.println("Data is: "+x);
        inFile.close();
}
```

May throw FileNotFoundException

May throw InputMismatchException

throw NegativeIntegerException

Inside **main():**

```
try
{
        System.out.print("Input the file pathname: ");
        String fname = in.next();
        processFile(fname);
        shouldEnd=true;
}
catch (FileNotFoundException e)
{
        ...
}
catch (InputMismatchException e)
{
        ...
}
catch (NegativeIntegerException e)
{
        System.out.println("Unexpected negative value from the file.");
        System.out.print("Try another file?  Type your choice [y/n]: ");
        shouldEnd=(in.next().charAt(0)=='n');
}
```

II Exception Handling – Basic Idea

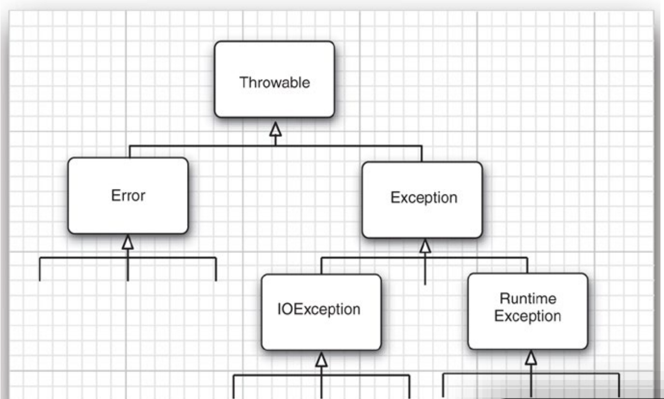
- **Possible errors of a running JAVA program**
  1) User input error (e.g. typing mistake, wrong format of URL)
  2) Device errors (e.g. printer suddenly jammed, webpage temporarily unavailable)
  3) Physical limitation (e.g. disk full, out of memory)
  4) Code error (e.g. invalid array index)

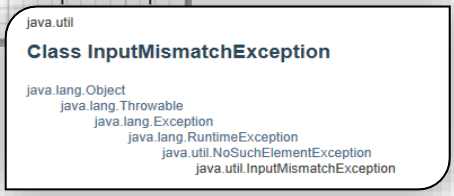
- **Error handling**
  1) [Traditional approach I] Method returns -1, special end-of-file value, etc..
  2) [Traditional approach II] Method returns a null reference.
  3) **JAVA's exception handling:**

     [Mission] **Transfer control from the location where error occurred to an error handler that can deal with the situation**

     - throws an object that encapsulates the error information
     - the code section exits immediately
     - it does not return any value (even the method return type is not void)
     - Search for an exception handler that can deal with this particular error.

III Exception Hierarchy in JAVA





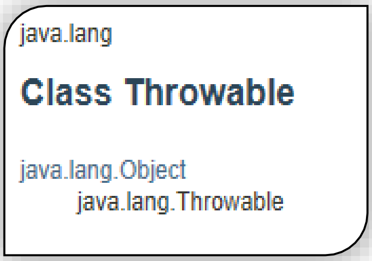
java.util

Class InputMismatchException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.util.NoSuchElementException
          java.util.InputMismatchException

http://docs.oracle.com/javase/7/docs/api/java/util/InputMismatchException.html

The Topmost Class: Throwable



java.lang

Class Throwable

java.lang.Object
  java.lang.Throwable

http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

- **`Throwable`** is the superclass of all errors and exceptions in JAVA,

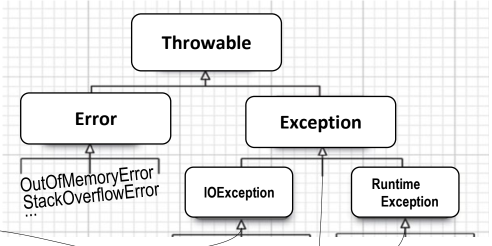
  - Thrown by JVM, e.g.

    `FileNotFoundException`

    `InputMismatchException`

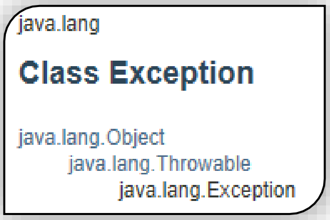
  - Thrown in our code, e.g.

    `NegativeIntegerException` (we defined in Example 5)



The Exception Hierarchy:

[http://docs.oracle.com/javase/7/docs/api/java/lang/Error.html]

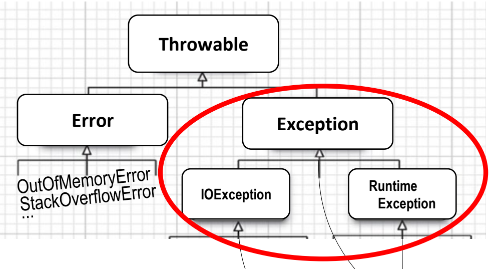


java.lang

Class Exception

java.lang.Object
  java.lang.Throwable
    java.lang.Exception

- The **Exception** hierarchy indicates conditions that a reasonable application might want to catch. (Ref. Examples 1-5)



Example 5 [recalled]

```
public static void processFile(String fname) throws ..
{
   Scanner inFile = new Scanner(new File(fname));
   int x = inFile.nextInt();
   if (x<0)
      throw new NegativeIntegerException();
   System.out.println("Data is: "+x);
   inFile.close();
}
```

JVM may throw FileNotFoundException

JVM may throw InputMismatchException

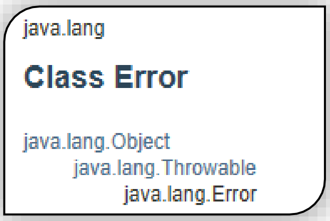
throw NegativeIntegerException

```
public static void main(String[] args)
{
  ..
  try {
         ...
         processFile(fname);
         ...
  }
  catch (FileNotFoundException e) {..}
  catch (InputMismatchException e) {..}
  catch (NegativeIntegerException e) {..}
  ...
}
```

Catch and handle

The Error Hierarchy:

[http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html]



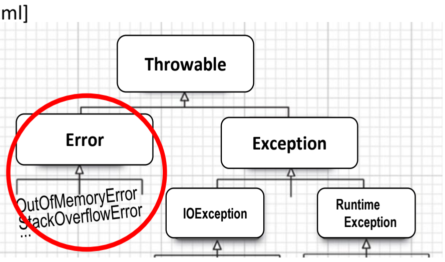
java.lang

Class Error

java.lang.Object
  java.lang.Throwable
    java.lang.Error

- The **Error** hierarchy describes internal errors and resource exhaustion situations which are abnormal.

  Indicates serious problems that a reasonable application should NOT try to catch. *(But if you want, you can!!)*



It should be aborted and the system (not our code) **need to take over the control.**

St

Note: Java programmers often call "Errors" as "Exceptions" as well. We speak in the same way in this topic.

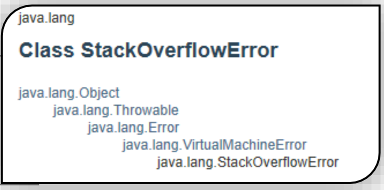
Examples:

java.lang.OutOfMemoryError (Previous lecture exercise: out of heap space)

```
Object[] arr1 = new Object[10000000];
Object[] arr = arr1;

for (int i=0;i<200;i++) {
      arr[0]=new Object[10000000];
      arr=(Object[])arr[0];
}
```

Program aborted by JAVA runtime

```
Exception in thread "main"
    java.lang.OutOfMemoryError:
      Java heap space
    at Main.main( Main.java:12)
```



java.lang

Class StackOverflowError

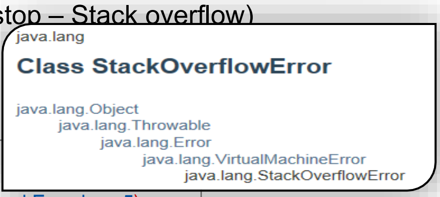
java.lang.Object
  java.lang.Throwable
    java.lang.Error
      java.lang.VirtualMachineError
        java.lang.StackOverflowError

java.lang.StackOverflowError (Previous lecture exercise: recursion cannot stop – Stack overflow)

```
private static int factorial(int n) {
   if (n==1) return 1;
   else return n*factorial(n+1);
}

public static void main(String[] args) {
   System.out.println(factorial(4));
}
```

Program aborted by JAVA runtime

```
Exception in thread "main"
  java.lang.StackOverflowError
  at MainStackError.factorial(MainStackError.java:5)
  at MainStackError.factorial(MainStackError.java:6)
  at MainStackError.factorial(MainStackError.java:6)
  at MainStackError.factorial(MainStackError.java:6)
  ...
```

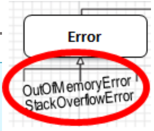


java.lang

Class StackOverflowError

java.lang.Object
  java.lang.Throwable
    java.lang.Error
      java.lang.VirtualMachineError
        java.lang.StackOverflowError

Catch an error and handle it, continue running.
[For illustration purpose only; recall: “a reasonable application should NOT try to catch”]



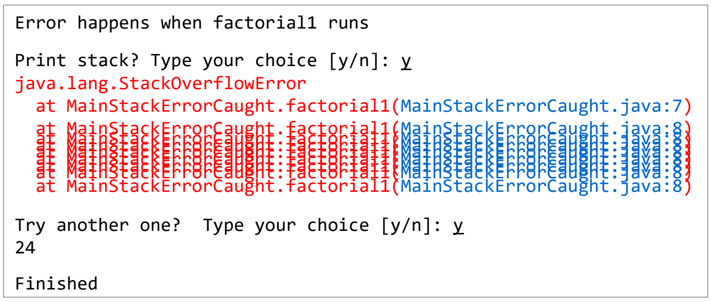
```
private static int factorial1(int n)
{
    if (n==1)       return 1;
    else      return n*factorial1(n+1);
}

private static int factorial2(int n)
{
    if (n==1)  return 1;
    else      return n*factorial2(n-1);
}
```

```
public static void main(String[] args)
{
    Scanner in = new Scanner(System.in);
    try
    {
        System.out.println(factorial1(4)); //failed
    }
    catch (StackOverflowError e)
    {
        System.out.printf("Error happens when factorial1 runs\n\n");
        System.out.print("Print stack? Type your choice [y/n]: ");
        if (in.next().charAt(0)=='y')
            e.printStackTrace();

        System.out.print("\nTry another one?  Type your choice [y/n]: ");
        if (in.next().charAt(0)=='y')
            System.out.println(factorial2(4)); //OK: 24
    }
    System.out.println("\nFinished");
}
```

We catch the error and handle it,

make the program continue to run

However, practically it should be aborted and the system (not our code) need to take over the control.



```
Error happens when factorial1 runs

Print stack? Type your choice [y/n]: y
java.lang.StackOverflowError
  at MainStackErrorCaught.factorial1(MainStackErrorCaught.java:7)
  at MainStackErrorCaught.factorial1(MainStackErrorCaught.java:8)
  at MainStackErrorCaught.factorial1(MainStackErrorCaught.java:8)
  at MainStackErrorCaught.factorial1(MainStackErrorCaught.java:8)
  at MainStackErrorCaught.factorial1(MainStackErrorCaught.java:8)

Try another one?  Type your choice [y/n]: y
24

Finished
```

IV The Try-throw-catch Mechanism

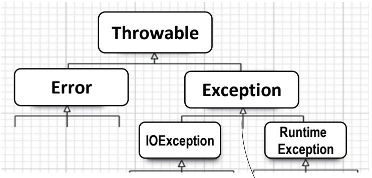
The Try-throw-catch Mechanism:

- The **try** block
  - Tells what to do when everything goes smooth
  - Can have code that throws exceptions for unusual conditions
  - If something goes wrong within the try block, execution in the try block is stoped and an exception is thrown
- The **throw** statement: We can write the throw statement to (create exception object and) throw an exception.
  - The similar is done when JAVA detects problem (e.g., JAVA may throw an object of **StackOverflowError**). **FileNotFoundException, InputMismatchException, etc..**
  - Syntax

    ```
    throw new ExceptionClassName(arguments);
    ```

    exception object is created and thrown
  - After exception is thrown, the flow of control is transferred to a catch block, and the catch block begins to run
- The **catch** block
  - Tells what to do when an exception is caught
  - One parameter, usually named “e” (but other names are also ok)
  - Syntax

    ```
    catch (ExceptionClassName e)
    {
            ..code to handle the exception
    }
    ```



V Define our own exception classes

- Predefined exception class contains:
  - An object field to store a message and a constructor that sets the message
  - An accessor method, `String getMessage()`
- We define our own exception class by inheriting `Exception` (or other throwables)
  - We can add fields and methods in our class
    E.g. `int problemValue`, `int getProblemValue()`
  - Constructors to be implemented:

```
public class NegativeIntegerException extends Exception
{
    private int problemValue;
    public int getProblemValue() { return problemValue; }

    public NegativeIntegerException() { super("Negative integer!"); }
    public NegativeIntegerException(String message) { super(message); }

    public NegativeIntegerException(String message, int v)
    {
        super(message); problemValue=v;
    }
}
```

It is customary to give both a default constructor and a constructor that contains a detailed message.

Add constructor according to our design

  - Which constructor will run is decided by how we create the exception object:

```
throw new NegativeIntegerException();
throw new NegativeIntegerException("-ve number");
throw new NegativeIntegerException("-ve number", x);
```

[Complete Code]

Example 6 Revised NegativeIntegerException: Add a field to store the problem value

`NegativeIntegerException:`

```
public class NegativeIntegerException extends Exception
{
   private int problemValue;
   public int getProblemValue() {return problemValue;}

   public NegativeIntegerException() {super("Negative integer!"); }
   public NegativeIntegerException(String msg) {super(msg); }

   public NegativeIntegerException(String msg, int v)
   {
      super(msg); problemValue=v;
   }
}
```

Inside `main()`:

```
try
{
   Scanner inFile = new Scanner(new File(fname));
   int x = inFile.nextInt();
   if (x<0)
      throw new NegativeIntegerException(“-ve number”, x);

   System.out.println("Data is: "+x);
  inFile.close();
}
catch (FileNotFoundException e) {
   System.out.println("Cannot open ...“);
}
catch (InputMismatchException e) {
   System.out.println("Cannot read ...");
}
catch (NegativeIntegerException e) {
  System.out.println(e.getMessage()+
     " ["+e.getProblemValue()+"]");
}
```

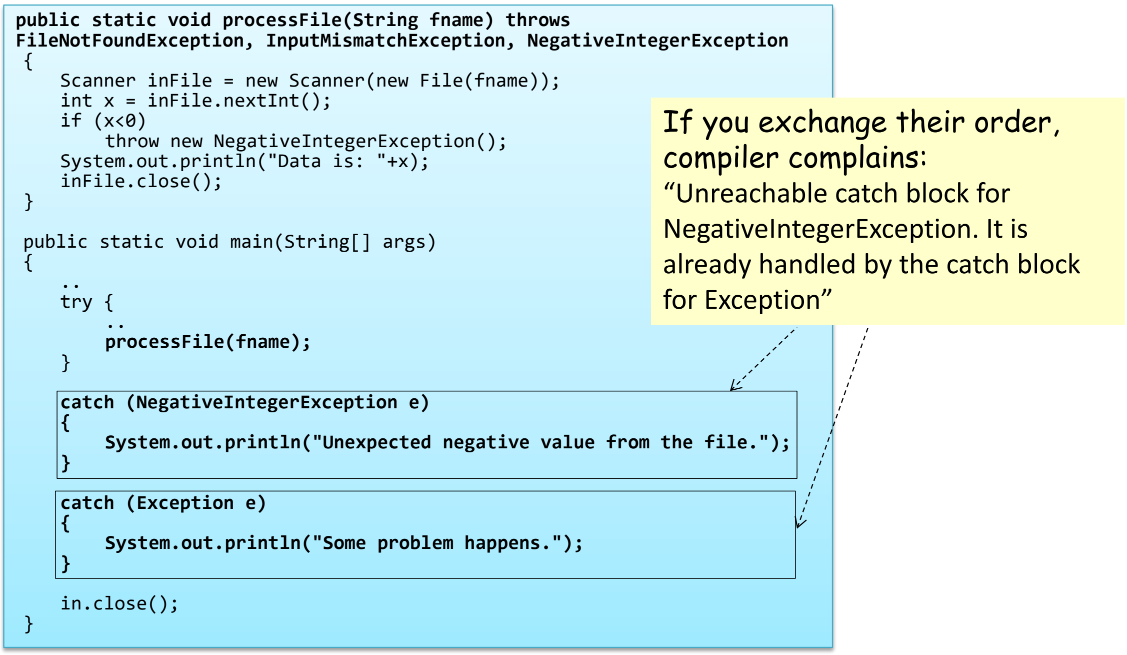
Throw?
Throw?
Throw!

Rundown 6.1:
Input the file pathname: c:\data003.txt
`-ve number [-1234]`

VI Pitfall: Catch the more specific exception first

- When we have 2 or more catch-blocks, catch the more specific exception first

Example 7 [based on Example 5]



```
public static void processFile(String fname) throws
FileNotFoundException, InputMismatchException, NegativeIntegerException
 {
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    if (x<0)
        throw new NegativeIntegerException();
    System.out.println("Data is: "+x);
    inFile.close();
 }

 public static void main(String[] args)
 {
    ..
    try {
        ..
        processFile(fname);
    }

    catch (NegativeIntegerException e)
    {
        System.out.println("Unexpected negative value from the file.");
    }

    catch (Exception e)
    {
        System.out.println("Some problem happens.");
    }

    in.close();
 }
```

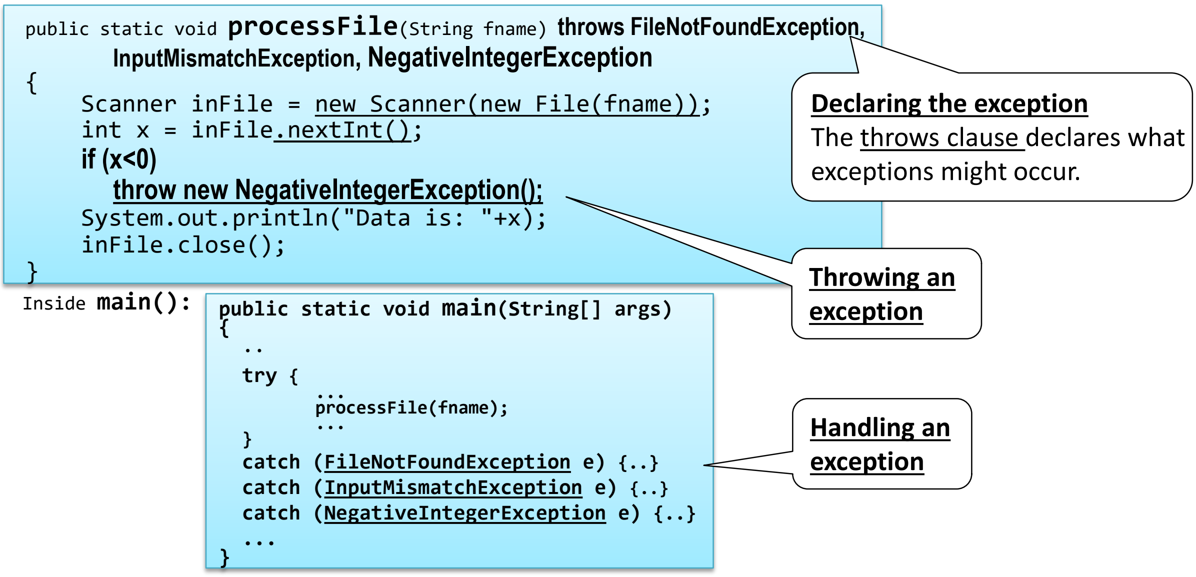
If you exchange their order, compiler complains:
“Unreachable catch block for NegativeIntegerException. It is already handled by the catch block for Exception”

VII The throws clause, throws clause in derived class

The Throws clause

- When a method may cause exception, but no catch is done within the method,
  Then it may inform the user by using <u>the Throws clause</u> to <u>declare</u> the exception
- The user may catch and handle the exception (or the user itself declares it again).

Example 5 [recall]



```
public static void processFile(String fname) throws FileNotFoundException,
        InputMismatchException, NegativeIntegerException
{
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    if (x<0)
        throw new NegativeIntegerException();
    System.out.println("Data is: "+x);
    inFile.close();
}
```

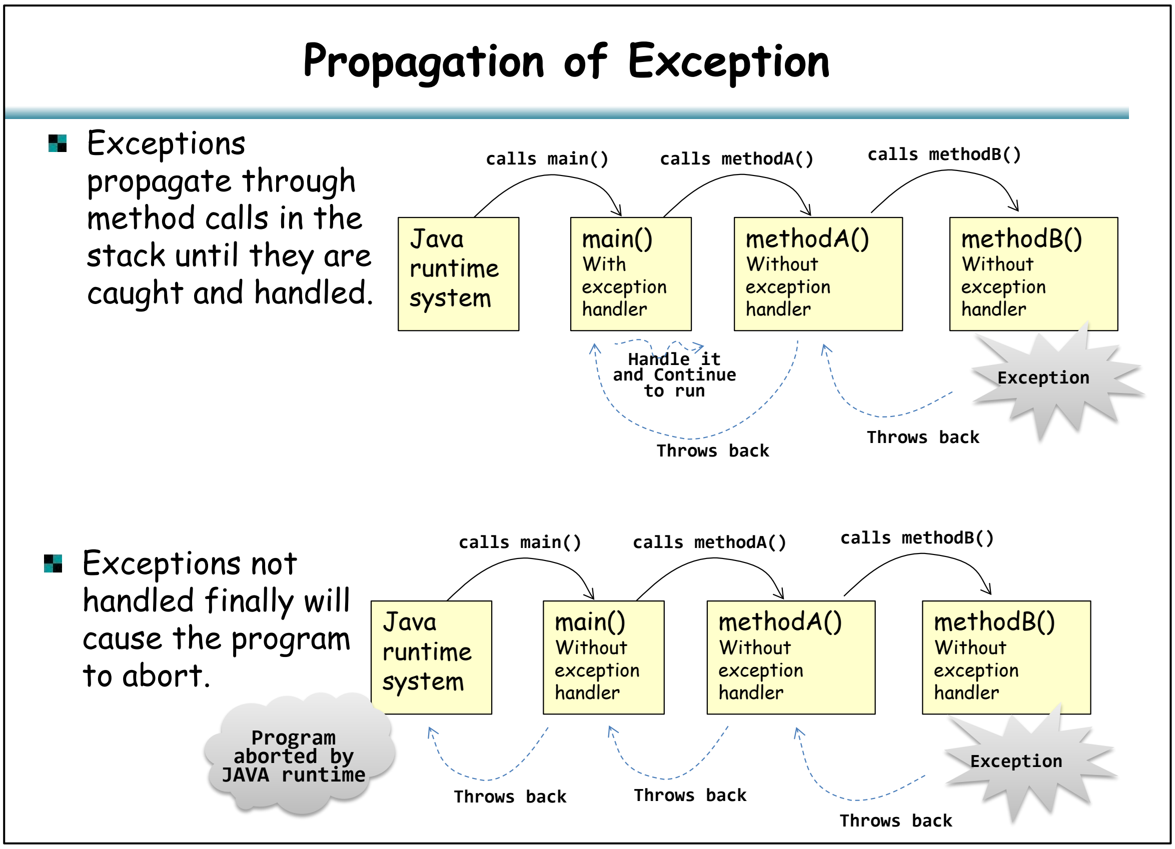
Inside main():

```
public static void main(String[] args)
{
   ..
   try {
          ...
          processFile(fname);
          ...
   }
   catch (FileNotFoundException e) {..}
   catch (InputMismatchException e) {..}
   catch (NegativeIntegerException e) {..}
   ...
}
```

Declaring the exception
The throws clause declares what exceptions might occur.

Throwing an exception

Handling an exception



Propagation of Exception

- Exceptions propagate through method calls in the stack until they are caught and handled.
- Exceptions not handled finally will cause the program to abort.

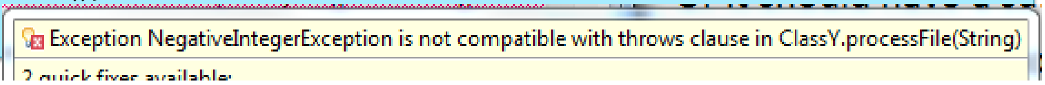
calls main()
calls methodA()
calls methodB()
Java runtime system
main() With exception handler
methodA() Without exception handler
methodB() Without exception handler
Exception
Throws back
Handle it and Continue to run
Throws back
calls main()
calls methodA()
calls methodB()
Java runtime system
main() Without exception handler
methodA() Without exception handler
methodB() Without exception handler
Exception
Program aborted by JAVA runtime
Throws back
Throws back
Throws back

Throws clause in derived classes

- When we redefine a method in a subclass, it should have the same exception classes listed in its throws clause that it had in the superclass
- Or it should have a subset of them
- i.e., a subclass may not add any exceptions to the throws clause,
  but it can delete some

Example 8

```
class ClassY
{
  public void processFile(String fname) throws FileNotFoundException
  {
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    System.out.println("Data is: "+x);
    inFile.close();
  }
}

class ClassZ extends ClassY
{
  public void processFile(String fname) throws FileNotFoundException, NegativeIntegerException
  {
    Scanner inFile = new Scanner(new File(fname));
    int x = inFile.nextInt();
    if (x<0)
        throw new NegativeIntegerException();
    System.out.println("Data is: "+x);
    inFile.close();
  }
}
```



Exception NegativeIntegerException is not compatible with throws clause in ClassY.processFile(String)

2 quick fixes available:

VIII The Catch-or-Declare Rule, Checked Exceptions, Unchecked Exceptions

The Catch-or-Declare Rule

- If an exception may be thrown inside a method,

  the method may deal with it by

  1. placing the concerned code within a try-block, and handle in a catch-block,

  or

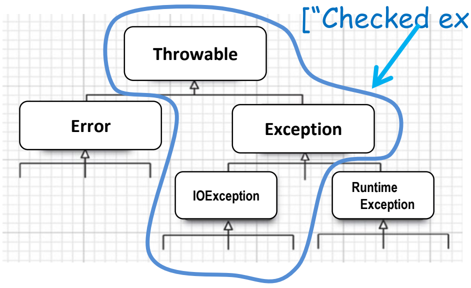
  2. declaring it

```
public void methodX()
{
  ..
  try {
        ... //code which may throw exception
  }
  catch (.. e) {
        ... //catch and handle the exception
  }
}
```

```
public void methodX() throws ..
{
    ..
}
```

- The compiler reinforces the Catch-or-Declare Rule on the following exceptions [Known as "checked exceptions"]:

  For other exceptions/errors,
  it is okay whether we deal
  with them or not.
  [Known as "unchecked exceptions"]



Must Catch-or-Declare
["Checked exceptions"]

IX The Finally Block

The Finally Block

- The finally block contains code to be executed whether or not an exception is thrown in a try block.

Example 6’

```
public static void main(String[] args)
 {
    Scanner in = new Scanner(System.in);
    System.out.print("Input the file pathname: ");
    String fname = in.next();

    Scanner inFile=null;
    try
    {
        inFile = new Scanner(new File(fname));
        ..
    }
    catch (FileNotFoundException e)
    {
        System.out.println("Cannot open the file.");
    }
    catch (InputMismatchException e)
    {
        System.out.println("Cannot read the required number.");
    }
    catch (NegativeIntegerException e)
    {
        System.out.println(e.getMessage()+" ["+e.getProblemValue()+"]");
    }
    finally {
        if (inFile != null) {
            System.out.println("Closing inFile");
            inFile.close();
        } else {
            System.out.println("inFile not open");
        }
        ..
        in.close();
    }
 }
```

--- end ---

Contents

Topic 07 - Generic Programming

I Generic Programming - Introduction

Generic Programming

- **Generics**
  – method and class definitions which involve type parameters.

- **Generic Programming**
  – writing code that can be reused for objects of many different types.

- User-defined generic classes and methods
  [See examples 1 and 2]

- There are also generic classes and methods provided in the standard Java libraries:
  e.g. the **ArrayList** generic class,
  the **Collections.sort** generic method
  [See example 3]

Example 1 Simple Generic Method

x is called the *value parameter*.
T is called the *type parameter*.
<T> means that:
In the following, T is the type parameter which stands for the actual type which is known when printTwice is called.

Declare a Generic Method

Use the Generic Method

```
public class Main
{
   public static <T> void printTwice(T x)
   {
      System.out.println(x);
      System.out.println(x);
   }

   public static void main(String[] args)
   {
      printTwice("hello");    //This time T is a string
      printTwice(1234);       //This time T is an integer
      printTwice(4.0/3);      //This time T is a double
   }
}
```

Output:

```
hello
hello
1234
1234
1.3333333333333333
1.3333333333333333
```

Example 2 Simple Generic Class

<T> is called the type parameter.
<T> means that:
In the following, T is the type parameter which stands for the actual type which is known when an object of this Generic is created.

Declare a Generic Class

```
class Pair<T>
{
      private T first;
      private T second;

      public Pair() { } //first and second automatically initialized as null
      public Pair(T x1, T x2) { first = x1; second = x2; }

      @Override
      public String toString() {return "(1)"+first+" (2)"+second;}
}
```

Each time when we use the Generic Class Pair, we need to tell the type which T stands for.

Use the Generic Class

```
public class Main
{
      public static void main(String[] args)
      {
            Pair<String> p0 = new Pair<String>();
            Pair<String> p1 = new Pair<String>("hello","cheers");
            Pair<Boolean> p2 = new Pair<Boolean>(true,false);
            Pair<Integer> p3 = new Pair<Integer>(123,456);
            Pair<Number> p4 = new Pair<Number>(123,456);
            Pair<Object> p5 = new Pair<Object>(123,"cheers");
            System.out.println(p0);
            System.out.println(p1);
            System.out.println(p2);
            System.out.println(p3);
            System.out.println(p4);
            System.out.println(p5);
      }
}
```

Output:

```
(1)null (2)null
(1)hello (2)cheers
(1)true (2)false
(1)123 (2)456
(1)123 (2)456
(1)123 (2)cheers
```

Example 3 The **ArrayList** generic class, the **Collections.sort** generic method [provided in standard Java libraries]

```
import java.util.ArrayList;
import java.util.Collections;

public class Main
{
    public static void main(String[] args)
    {
        ArrayList<Integer> arrlist = new ArrayList<>();
        arrlist.add(1234);
        arrlist.add(8899);
        arrlist.add(36);
        Collections.sort(arrlist);
        System.out.println(arrlist); //Output: [36, 1234, 8899]
    }
}
```

[For interested students only]

The ArrayList class is defined as: **class ArrayList<E>**

The sort method is defined as: **public static <T extends Comparable<? super T>> void sort(List<T> list)**

Where **List** is a Java interface implemented by **ArrayList** and a number of other Java classes.

[http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html]
[http://stackoverflow.com/questions/4343202/difference-between-super-t-and-extends-t-in-java]

II Type Inference and Diamond Syntax

Type Inference

- *Type inference* is a Java compiler's ability to determine the type argument(s) that make the invocation applicable.
- Compiler looks at each method invocation① and corresponding declaration②, in order to decide the type.

Ref: http://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html

Example 1

value parameter

```
public static <T> void printTwice(T x)
{
    System.out.println(x);
    System.out.println(x);
}

public static void main(String[] args)
{
    printTwice("hello"); //This time T is a string
    printTwice(1234);    //This time T is an integer
    printTwice(4.0/3);   //This time T is a double ①
}
```

type parameter

type parameter

type argument

Example 2

```
class Pair<T>
{
    private T first;
    private T second;
    ..
}
public static void main(String[] args)
{
    Pair<String> p0 = new Pair<String>();
    Pair<String> p1 = new Pair<String>("..", "..");
    Pair<Boolean> p2 = new Pair<Boolean>(true,false);
}
```

②

This time T is a boolean

The Diamond Syntax: <>

- We can omit the types in <> when **new** is used. (Since Java 7)
- i.e., simply write

```
Pair<String> p0 = new Pair<> ();
Pair<String> p1 = new Pair<> ("hello","cheers");
Pair<Boolean> p2 = new Pair<>(true,false);
```

Compiler checks the type of the object variable (here p0, p1, p2) to guess and fill in the type parameter.

III Pitfalls and Type Erasure

- Static fields belong to the generic class, not the instantiated classes.

```
class Smartphone {}
class Pager {}
class TabletPC {}
class MobileDevice<T> {
    private static int count=0;
    MobileDevice() {count++;System.out.println(count);}
    // ...
}
public class Main
{
    public static void main(String[] args)
    {
            MobileDevice<Smartphone> phone = new MobileDevice<>();
            MobileDevice<Pager> pager = new MobileDevice<>();
            MobileDevice<TabletPC> pc = new MobileDevice<>();
    }
}
```

<u>Output:</u>

```
1
2
3
```

- Cannot declare static fields of a type parameter

```
public class MobileDevice<T> {
    private static T os; // compile-time error

    // ...
}
```

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

What should be the type of <u>os</u>?

[http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createObjects]

- Cannot create an object instance of a type parameter

Compile error:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

- A class cannot have two overloaded methods that will have the same signature after *type erasure*.

```
public class X{
    public void print(Pair<String> strSet) { } //compile-time error
    public void print(Pair<Integer> intSet) { } //compile-time error
}
```

Compile-time error:
Method print(Pair<String>) has the same erasure print(Pair<T>) as another method in type X

For *Type Erasure* – We may think of it in this way:

When the compiler generates the bytecode, type parameters in generic types are <u>"replaced with the raw type* : *Object*"</u>.

Advantage: *Type Erasure* ensures that no new classes are created for parameterized types; consequently generics incur no runtime overhead. (c.f. Different approach in C++ *templates*)

* A type parameter could be *bounded. If so, it is replaced by the bound instead of Object.*

Type Erasure [For interested students only]

http://docs.oracle.com/javase/tutorial/java/generics/erasure.html
http://docs.oracle.com/javase/tutorial/java/generics/genTypes.html

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies **type erasure** to:

- Replace all type parameters in generic types with ***raw types***, which are their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur **no runtime overhead**.

The raw type for `Pair<T>` in Example 2 looks like:

```
class Pair
{
   private Object first;
   private Object second;
   public Pair() { }

   public Pair(Object first, Object second)
   {
      this.first = first;
      this.second = second;
   }

   @Override
   public String toString() {
        return "(1)"+first+" (2)"+second;
   }
}
```

```
public static void main(String[] args)
{
      Pair p0 = new Pair();
      Pair p1 = new Pair("hello","cheers");
      Pair p5 = new Pair(123,"cheers");
      System.out.println(p0); //(1)null (2)null
      System.out.println(p1); //(1)hello (2)cheers
      System.out.println(p5); //(1)123 (2)cheers
}
```

Pitfalls - More [For interested students only]

```
class Pair<T>
{
      private T first;
      private T second;
      public Pair() { }
      public Pair(T x1, T x2) { first = x1; second = x2; }
```

- Constructor headings do not include the type parameter

  We do not write

```
☒ public <T> Pair() { }
☒ public <T> Pair(T x1, T x2) { first = x1; second = x2; }
```

- Cannot use a generic class as the base type of an array

```
String [] x1=new String[3]; //OK
Pair<String>[] x2 = new Pair<String>[3]; //Error: Cannot create a generic array
                                                  of Pair<String>
```

- Cannot create, catch, or throw objects of parameterized types

```
class MathException<T> extends Exception { /* ... */ }    // compile-time error

class QueueFullException<T> extends Throwable { /* ... */} // compile-time error
```

- Cannot use casts or instanceof with parameterized types

```
if (list instanceof ArrayList<Integer>) // compile-time error
```

```
Pair<Integer> li = new Pair<>();
Pair<Number>  ln = (Pair<Number>) li; // compile-time error
```

But we can use *bounded wildcard* *:
Pair<? **extends Number> ln = li;**

IV Generic Class with more than 1 parameter [For interested students only]

Generic Class can have more than 1 type parameters

Example 2a

```
class TwoTypePair<T1, T2>
{
    private T1 first;
    private T2 second;

    public TwoTypePair(T1 firstItem, T2 secondItem)
    {
        first = firstItem;
        second = secondItem;
    }
    public void setFirst(T1 newFirst) { first = newFirst; }
    public void setSecond(T2 newSecond) { second = newSecond; }
    public T1 getFirst() { return first; }
    public T2 getSecond() { return second; }
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass( ) != otherObject.getClass( ))
            return false;
        else
        {
            TwoTypePair<T1, T2> otherPair =
                (TwoTypePair<T1, T2>)otherObject;
            return (first.equals(otherPair.first)
                && second.equals(otherPair.second));
        }
    }
}
```

```
public static void main(String[] args)
{
   TwoTypePair<String, Integer> x1,x2,x3;
   x1 = new TwoTypePair<>("USD",123);
   x2 = new TwoTypePair<>("USD",456);
   x3 = new TwoTypePair<>("USD",123);

   System.out.println(x1.equals(x2));//false
   System.out.println(x1.equals(x3));//true
}
```

V Other Notes - Bounds for type parameter, Generic Interface, Inheritance with Generic Class

Bounds for Type Parameters

- To restrict the possible types that can be plugged in for a type parameter T
- Example:

```
public class TwoBTypePair<T1 extends Class1, T2 extends Class2 & Comparable>
```

Generic Interfaces (Similar to generic classes)

- Example:

```
interface PairInterface<T1, T2>
{
      void processThePair(T1 t1, T2 t2);
}
```

Inheritance with Generic Classes

- A generic class can be defined as a derived class of an ordinary class or of another generic class
- Example:

```
class C0 {}
class C1<T> extends C0  {}
class C2<T> extends C1<T>  {}
class C3<T> extends C1<String> {}
```

Contents

Topic 08 - Collections

[Ref: Core Java Chp 13 , Intro to Java Programming [Liang] Chp 22, Absolute Java Chp 16, docs.oracle.com/javase/tutorial/collections/TOC.html]

I Introduction - Java Collection Hierarchy

Java Collection Framework

- A collection is a container object that holds a group of objects

“Elements”

- A framework is a set of classes, which form the basis for building advanced functionality
- The Java Collections Framework supports different types of collections:

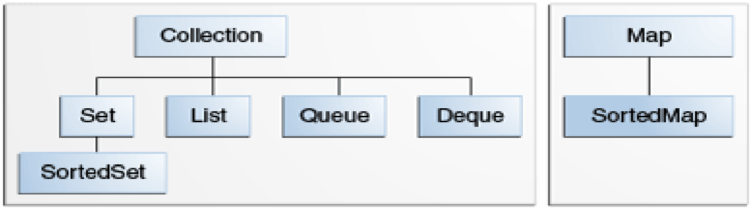
  Containers for storing a collection of elements:

  1. Sets – store a group of non-duplicate elements
  2. Lists – store an ordered collection of elements
  3. Queues– store objects that are processed in first-in, first-out fashion

  Containers for storing key/value pairs:

  4. Maps – store key/value pairs

- Interfaces in the hierarchies:

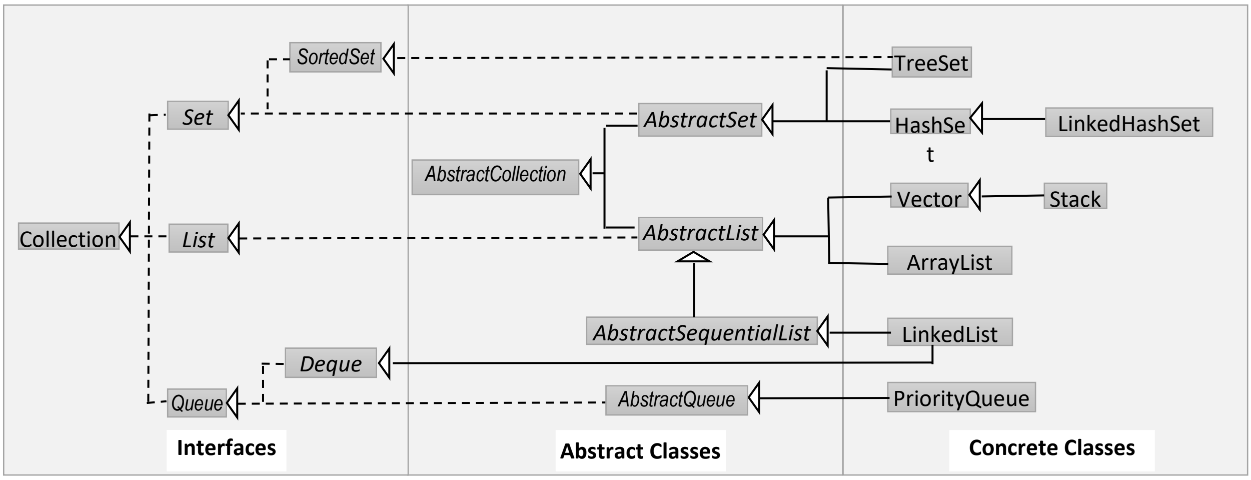


Two distinct trees: Collection and Map

[http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html]

- The Java Collection Framework is an excellent example of using interfaces, abstract classes, and concrete classes.
  - Interfaces – define the framework
  - Abstract classes – provide partial implementation
  - Concrete classes – implement the interfaces with concrete data structures

  Some of the interfaces and classes in Java Collection [Liang Chp.22]:



  Providing an abstract class (partial implements an interface) makes it convenient for the user to write the code.

  The user can simply define a concrete class that extends the abstract class (rather than implementing all methods in the interface)

II Choosing/using collections

- How to choose a data structure from the Java Collection Framework?
  - *Need quick search?*
  - *Data should be kept sorted?*
  - *Rapid insertion/removal in the middle?*
  - *Need association between keys and values?*
  - *etc..*

- The way to use a data structure:

```
import java.util.*;

public class Main
{
   public static void main(String[] args) {
      List<Integer> dataList;
      dataList = new Vector<>();
      dataList.add(100);
      dataList.add(200);
      System.out.println(dataList);
   }
}
```

When we use a list, we do not need to know which implementation is actually chosen once it has been constructed.

Therefore we use the interface type for the variable (to hold the reference)

If we change our mind, we can easily use a different one.
e.g., change to:

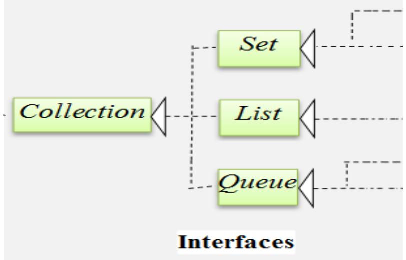
```
        dataList = new ArrayList<>();
              or:
        dataList = new LinkedList<>();
```

For interested students: http://beginnersbook.com/2013/12/**difference-between-arraylist-and-vector-in-java**/

III Collection and Iterator *(For interested students only)*

- The `Collection` Interface:

```
public interface Collection<E>
{
      //2 fundamental methods:
      boolean add(E element);
      Iterator<E> iterator();
      ..
}
```



- The `iterator` method returns an `Iterator` object.
- The `Iterator` object is for visiting the elements in the collection one by one (See the `Iterator` interface).

```
public static void main(String[] args)
{
      List<Integer> dataList;
      dataList = new ArrayList<>();
      for(int i=0;i<10;i++)
            dataList.add(i*i);

      //(1) Get the iterator and use it to visit elements
      Iterator<Integer> itr = dataList.iterator();
      while (itr.hasNext()){
            Integer e = itr.next();
            System.out.print(e+" ");
      }                         Output: 0 1 4 9 16 25 36 49 64 81

      //(2) for-each loop (Actually a shortcut for (1))
      for (Integer e: dataList)
            System.out.print(e+" ");
}                               Output: 0 1 4 9 16 25 36 49 64 81
```

The Iterator Interface:

```
public interface Iterator<E>
{
      E next();
      boolean hasNext();
      ..
}
```

IV Methods of Collection

- The Collection interface is generic
- Usage examples:
  `ArrayList<Employee> emList;`
  `ArrayList<Student> sList;`
- ie., when we create a collection, there is a type parameter for us to provide the class type of the collection elements.

```
public interface Collection<E>
{
      ..
      int size();
      boolean isEmpty();
      boolean contains(Object obj);
      boolean containsAll(Collection<?> c);
      boolean equals(Object other);
      boolean addAll(Collection<? extends E> from);
      boolean remove(Object obj)
      boolean removeAll(Collection<?> c)
      void clear()
      boolean retainAll(Collection<?> c)
      Object[] toArray()
      <T> T[] toArray(T[] arrayToFill)
}
```

```
public abstract class AbstractCollection<E> implements Collection<E>
{
      public boolean contains(Object obj)
      {
            for (E element : this) // calls iterator()
                if (element.equals(obj))
                      return true;
            return false;
      }
      ..
}
```

V Concrete classes

- Some concrete classes:

ArrayList	An indexed sequence that grows and shrinks dynamically
LinkedList	An ordered sequence that allows efficient insertion and removal at any location
ArrayDeque	A double-ended queue that is implemented as a circular array
HashSet	An unordered set collection (set: rejects duplicates)
TreeSet	A sorted set collection (set: rejects duplicates)
PriorityQueue	A collection that allows efficient removal of the smallest element
HashMap	A map data structure that stores key/value associations
TreeMap	A map data structure that stores key/value associations (sorted by keys)

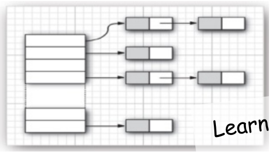
VI Implementation (Hash table, resizable array, tree, linked-list)

- Commonly used implementations (concrete classes) for collection interfaces

General-purpose Implementations				
Interfaces	Resizable array Implementations	Linked list Implementations	Hash table Implementations	Tree Implementations
Set			HashSet	TreeSet
List	ArrayList	LinkedList		
Queue, Deque	ArrayDeque	LinkedList		
Map			HashMap	TreeMap

[http://docs.oracle.com/javase/tutorial/collections/implementations/index.html]

- Hash table implementation: fast lookup, data unsorted, require *hash code*
  In Java, hash tables are implemented as an array of buckets (linked-lists)



Learn in CS3334

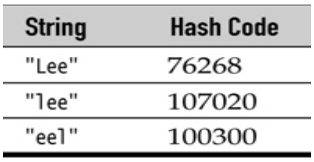
- Tree implementation: fast lookup, data sorted, implemented as Red-black tree

Learn in CS3334

VII HashSet, TreeSet, Comparator

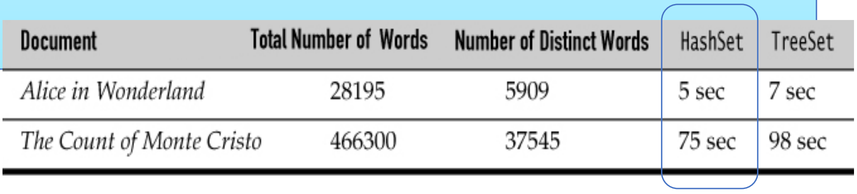
HashSet

- Example: using `HashSet<String>` to store the words in "Alice in Wonderland"
- Hash code - an integer for each object to be hashed
  - computed quickly based on the state (field values) of the object
  - determine where to insert the object in the hash table.
- Hash codes for Strings in Java:



String	Hash Code
"Lee"	76268
"lee"	107020
"eel"	100300

```
public static void main(String[] args) throws FileNotFoundException
{
      Set<String> words = new HashSet<>(); // HashSet implements Set
      Scanner in = new Scanner(new File("alice.txt"));
      while (in.hasNext())
      {
            String word = in.next();
            words.add(word);
      }

      System.out.println(words); //Output: [..,..,..] <== all distinct words in the file
      in.close();
}
```



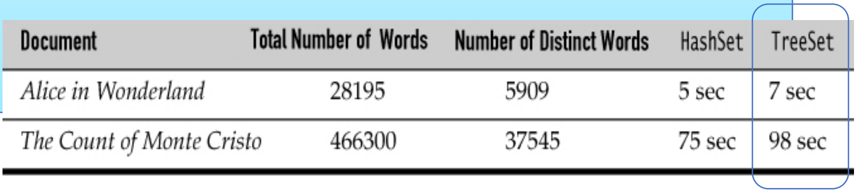
Document	Total Number of Words	Number of Distinct Words	HashSet	TreeSet
Alice in Wonderland	28195	5909	5 sec	7 sec
The Count of Monte Cristo	466300	37545	75 sec	98 sec

TreeSet

- Example: using `TreeSet<String>` to store the words in "Alice in Wonderland"
- `TreeSet` is: - similar to `Hashset`
  - plus improvement: as sorted collection
    ie. when iterated, values are presented in sorted order
  - insertion is slower than HashSet **but** much faster than array/linked-list

```
public static void main(String[] args) throws FileNotFoundException
{
      Set<String> words = new TreeSet<>(); // TreeSet implements Set

      Scanner in = new Scanner(new File("alice.txt"));
      while (in.hasNext())
      {
            String word = in.next();
            words.add(word);
      }

      System.out.println(words); //Output: [..,..,..] <== all distinct words in the file
      in.close();
}
```

In sorted (alphabetical, case sensitive) order



Document	Total Number of Words	Number of Distinct Words	HashSet	TreeSet
Alice in Wonderland	28195	5909	5 sec	7 sec
The Count of Monte Cristo	466300	37545	75 sec	98 sec

Comparator

- We've learnt the `Comparable` interface for comparison of objects (also used in sorting)
- But, how to sort items by field `f1` in one collection, then field `f2` in another collection?

```
class Product implements Comparable<Product> {
      private int part_number;          //sometimes we want to compare part_number
      private String product_name;      //sometimes we want compare product name
      public int compareTo(Product other) {return Integer.compare(part_number, other.part_number);
      ..
}
```

- Solution in Java: pass a `Comparator` object into the `TreeSet` constructor
  - `Comparator` is an interface:

```
public interface Comparator<T>
{
      int compare(T a, T b);
}
```

- We often implement it as an inner class (with no class name).

This is an anonymous inner class, which implements the Comparator<Product> interface

- Further learning of `Comparator`:
  See lecture exercise:
  Using `Comparator` in `.sort(..)`.

```
public static void main(String[] args)
{
   SortedSet<Product> parts = new TreeSet<>();
   parts.add(new Product("Widget", 4562));
   parts.add(new Product("Toaster", 1234));
   parts.add(new Product("Modem", 9912));
   System.out.println(parts);
```

```
[Toaster(1234), Widget(4562), Modem(9912)]
```

Ordered by part_number (See compareTo)

```
   SortedSet<Product> sortByName = new TreeSet<>(
     new Comparator<Product>()
     {
        public int compare(Product a, Product b)
        {
           String descrA = a.getName();
           String descrB = b.getName();
           return descrA.compareTo(descrB);
        }
     }
   );
   sortByName.addAll(parts);
   System.out.println(sortByName);
}
```

```
[Modem(9912), Toaster(1234), Widget(4562)]
```

Ordered by product_name (See compare)

VIII Priority Queue *(For interested students only)*

- **PriorityQueue** (Underlying implementation: priority heap)

  Example: job scheduling

```
class Assignment implements Comparable<Assignment>
{
      private int priority; //1 means highest priority
      private String name; //e.g. "CS2312 Assignment", "CS3342 Project"  "CS3334 Survey"
      public Assignment(String n,int p) { priority=p; name=n; }
      public int compareTo(Assignment other) {return Integer.compare(priority, other.priority);}
      public String toString() {return name+"(Priority:"+priority+")";}
}
```

```
public static void main(String[] args)
{
      PriorityQueue<Assignment> qToDo = new PriorityQueue<>();
      qToDo.add(new Assignment("CS3342 Project", 2));
      qToDo.add(new Assignment("CS3334 Survey", 1));
      qToDo.add(new Assignment("CS2312 Assignment", 1));
      System.out.println(qToDo); //order not guaranteed
```

```
[CS3334 Survey(Priority:1), CS3342 Project(Priority:2), CS2312 Assignment(Priority:1)]
```

```
      System.out.println(qToDo.remove()); //removed based on priority
      System.out.println(qToDo.remove());
      System.out.println(qToDo.remove());
}
```

```
CS3334 Survey(Priority:1)
CS2312 Assignment(Priority:1)
CS3342 Project(Priority:2)
```

IX HashMap, TreeMap

- **The `Map` Interface (implementing classes: `HashMap`, `TreeMap`)**

  - A map stores key/value pairs. Both key and value must be objects.

    Example: we have some key info, we want to look up the associated element.

- **Implementing classes:**

  <u>`HashMap`</u> unordered implementation of `Map`; hashing the key (Learn in CS3334)

  <u>`TreeMap`</u> ordered implementation of `Map`; ordering on the key$^{\text{which implements Comparable}}$ (Red-black tree; Learn in CS3334)

  - Both `HashMap` and `TreeMap` hash/compare on keys.
  - For `Hashmap`, the class of the keys needs to provide `equals()` and `hashCode()`. `hashCode()` should return the hash code such that 2 objects which are considered as equal should have the same *hash code*.

    How JAVA locates an object: Find the location with `hashcode()`; then use `equals()` to identify it.

  - Useful "view" methods to get the set of keys, collection of values, or set of key-value pairs.

- **Example:** `.get, .put`

```
class Product {
     private String product_name;
     public Product(String name) {product_name=name;}
     public String toString() {return product_name;}
}
```

```
public static void main(String[] args)
{
     Map<Integer,Product> parts = new HashMap<>();
     parts.put(4562,new Product("Widget"));
     parts.put(1234,new Product("Toaster"));
     parts.put(9912,new Product("Modem"));

     System.out.println(parts.get(9912));       //output: Modem
     parts.put(9912,new Product("Router"));
     System.out.println(parts.get(9912));       //output: Router

     //Get the view: a set of the keys, for iteration;
     //output: Toaster(1234) Router(9912) Widget(4562)
     Set<Integer> kSet=parts.keySet();
     for(Integer k:kSet)
          System.out.printf("%s(%d) ",parts.get(k),k);
}
```

X Conversion to/from array *(For interested students only)*

From array to collection wrapper

- **Lightweight Collection Wrappers:** created using `asList(..)` of the `Arrays` class

```
public static void main(String[] args)
{
    String [] nArr = {"Helena", "Kit", "Jason"};    //an array

    List<String> nList = Arrays.asList(nArr);      //returns a List wrapper

    nList.set(0, "Marian");  //cannot apply .add or .remove which changes array size

    System.out.println(Arrays.toString(nArr));     //output: [Marian, Kit, Jason]
    System.out.println(nList);                     //output: [Marian, Kit, Jason]
}
```

From collection to an array copy

- **`Collection.toArray(..)`**

```
public static void main(String[] args)
{
    Collection<String> c = new ArrayList<>();
    c.add("Helena");c.add("Kit");c.add("Jason");
    String[] arr = c.toArray(new String[1]); //create a new array copy; initial size=1; expand as needed

    arr[0]="Tom";

    System.out.println(c);                         //output: [Helena, Kit, Jason]
    System.out.println(Arrays.toString(arr));      //output: [Tom, Kit, Jason]
}
```

XI Simple algorithms: shuffling, sorting, binarySearch, reverse, disjoint..

- **`Collections.xxx(..) – Useful methods for collections`**

```
public static void main(String[] args)
{
    List<String> c = new ArrayList<>(); c.add("Helena");c.add("Kit");c.add("Jason");
    System.out.println(c); //Output: [Helena, Kit, Jason]

    //Sorting
    Collections.sort(c); System.out.println(c);             //Output: [Helena, Jason, Kit]

    //Binary search
    System.out.println(Collections.binarySearch(c,"Helena")); //returns the index: 0

    //Others:
    System.out.println(Collections.disjoint(c,c.subList(0, 1)));  //Output: false
}

//Other methods: max, min, frequency, reverse, rotate, shuffle
```

true if no elements in common

Contents

Topic 09 - OOP Review – Features, Techniques, Practices and Principles

I OOP Basics, Classes and Objects, Refactoring

Programming - Procedural Approach vs OO Approach

Procedural Approach	Object-Oriented Approach
Specify what tasks to do in each step.	Specify who performs what tasks in each step.

World objects vs OO Programs

The world : Objects **communicate** to complete some tasks

OO Programs : Objects **communicate** to provide some functions to users

Why OOP?

- State-of-the-art, popular approach in the industry
- To handle complexity
- To ease change, e.g., Encapsulation

Relationship between classes (Ref Topic03: A class is the template / blueprint from which objects are made)

- Inheritance ("is-a") [See topics 04]
- Interface ("Implements-a") [See topics 05]
- Dependency ("uses-a") [Next page]

Relationship between objects

- Association ("Knows-a") [Next page]
- Aggregation and Composition ("Has-a") [Next page]

How to maintain good quality of an OOP program - Refactoring

Refactoring

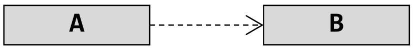
At the beginning, nobody wanted to write poor code;
Gradually, our code starts to look ugly (code smell).
One day we find that it is hard (then towards impossible) to maintain or change.

Refactoring: Improve code structure, without changing its external behavior

- Remove unhealthy dependencies between classes or packages
- Solve for bad class / method responsibilities; Reduce duplicate code and confusion
- Programmers ***refactor continuously*** to keep code as clean, simple, and expressive as possible.

II. The Dependency, Association, Aggregation, Composition relationships

Dependency ("Uses a ")



- We say "Class A the client depends on Class B the supplier " if:
  Any change to B requires A to be changed and/or ***recompiled****.*
- Eg.
  1. B is the superclass of A; or B is the interface implemented by A
  2. B is the type of
     a field of class A,
     a parameter of a class A method, or
     a local variable of a class A method
  3. a class B method or field is used / accessed from the code of class A, or
     a B object is created in class A's code

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

```
class A implements B,C,D {
      ..
}
```

```
class A {
      private B b;
      ..
}
```

```
class A {
      private static B b;
      ..
}
```

```
class A {
      public void handleIt(B b) {
            xxx(b);
      }
}
```

```
class A {
      public void handleIt() {
            B b = Company.getB();
            b.xxx();
      }
}
```

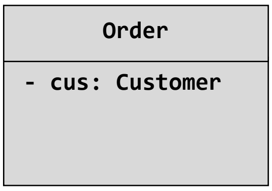
```
class A {
  public void handleIt(C c) {
    c.getB().xxx();  //xxx is B's method
  }
}
```

```
class A {
  public void handleIt() {
    (new B()).xxx();
  }
}
```

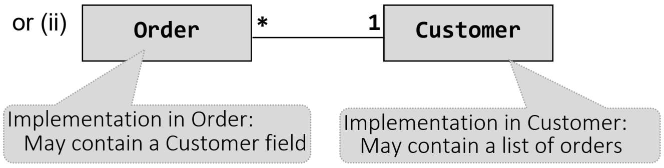
Association ("Knows a")

- An association is a type of object links.
- Often implemented as object fields, like other attributes
- Often expressed in two notations (depends on your intention):

(i)



or (ii)

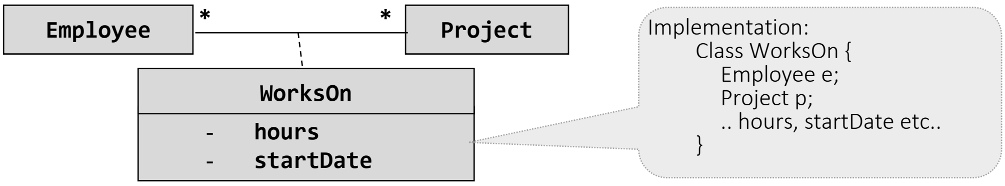


Implementation in Order:
May contain a Customer field

Implementation in Customer:
May contain a list of orders

↑ the written code is
the same as (i)

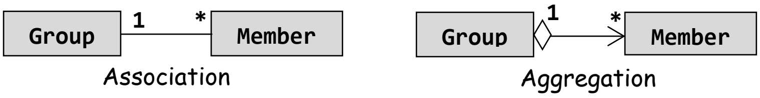
- **Association class**
  If the association has attributes, it should be implemented as a separate class:



Implementation:
    Class WorksOn {
        Employee e;
        Project p;
        .. hours, startDate etc..
    }

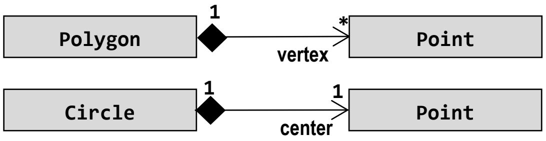
Aggregation and Composition ("Has a")

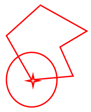
- **Aggregation** has vague semantics (Aggregations are also associations)



- **Composition** relates to creating and controlling the lifetime of another object (also think about: Garbage Collection).

  Single-owner: When B is composed by A, A owns B.





Suppose a vertex of a polygon p1 is (123, 45) , and the center of a circle c1 happens to be (123, 45) as well; Even so, p1 and c1 should link to two different Point objects

FAQ:

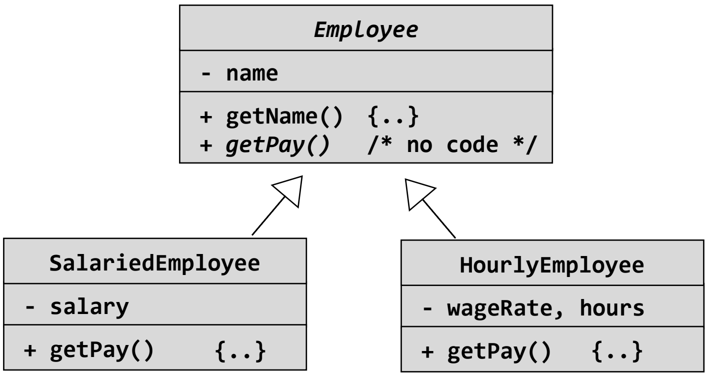
[Q] It is often vague to tell two classes' relationship (*Association? Dependency? Aggregation?..*) Then, use which one?
[A] Fact is: **More than one could be correct!** Therefore, see which can best match your intention (what to tell).

III. OO Techniques: Encapsulation, Abstraction, Generalization, Realization, Delegation

- **Encapsulation** [Topic03] : Combine data and behavior in one package Hide details from the users

- **Abstraction**: specify the framework and hide implementation details.
  - Give the developer a blueprint to follow.
  - Tell the user *what* instead of *how*
  - Eg. in Java: interfaces and abstract methods

- **Generalization**: provide common structure and behavior at an upper level of the hierarchy (available to lower level along inheritance)
  - Eg. We first design SalariedEmployee and HourlyEmployee as 2 separate classes,
    Then we observe common structure and behavior and rewrite as:

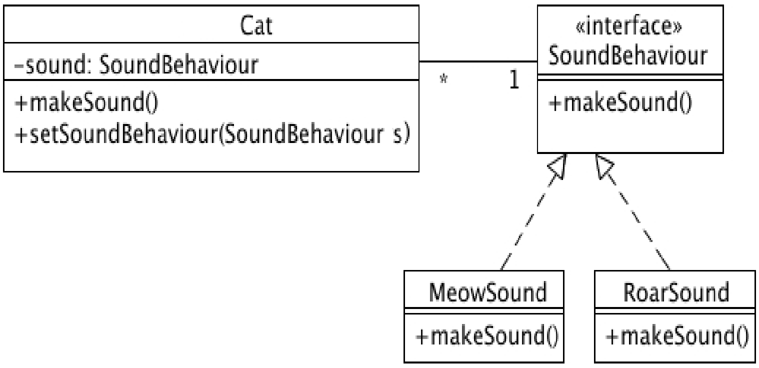


- **Realization**: provide implementation details to realize the abstract blueprint

Examples : .... implements Cloneable
.... implements Comparable <..>
class CmdAddSalary extends RecordedCommand
↑ Concrete class ↑ Abstract class

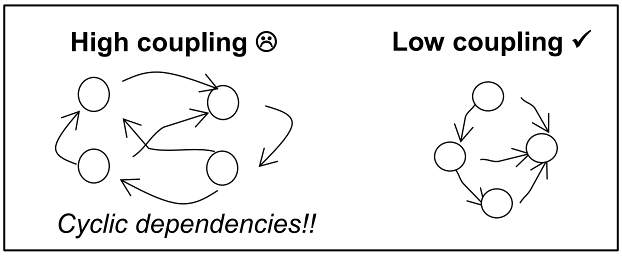
- **Delegation:** An object forwards or delegates a request to another object. The delegate carries out the request on behalf of the original object.

  Advantage:
  Easily compose behaviors at run-time.
  Often used as an alternative to inheritance.



```
interface SoundBehavior {
   void makeSound();
}
---------------------------------------------------------------------
public class MeowSound implements SoundBehavior {
   public void makeSound() { System.out.println("Meow"); }
}
---------------------------------------------------------------------
public class RoarSound implements SoundBehavior {
   public void makeSound() { System.out.println(“Roar!"); }
}
---------------------------------------------------------------------
public class Cat {
   private SoundBehavior sound = new MeowSound();
   public void makeSound() { sound.makeSound(); }
   public void setSoundBehavior(SoundBehavior newsound) { sound = newsound; }
}
```

IV. Measurements: Cohesion and coupling



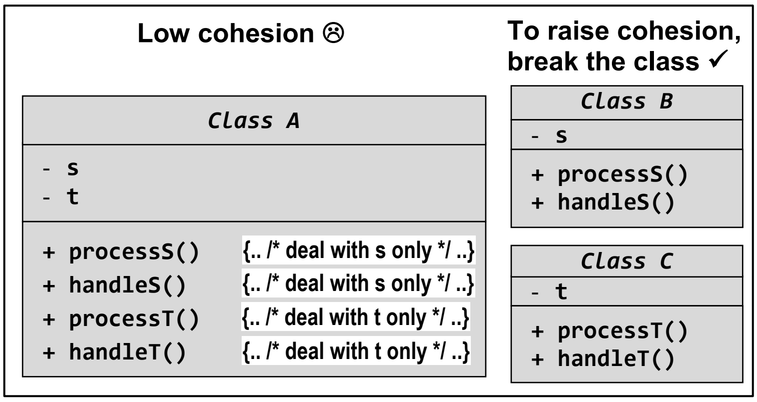
Coupling

- The degree to which software components depend on each other
- Two classes (methods) are coupled if changing one of them leads to a change of the other.

Cohesion

- The degree to which a class/method has ONE and ONLY ONE purpose.
- A class (method) has a low cohesion if it does many unrelated things or too much work.
- One common mistake of OO design: Too few classes/methods! Measurement: *Cohesion!*

A good design should have **high cohesion**, **low coupling**



Low cohesion ☹

Class A
- s
- t
+ processS() {.. /* deal with s only */ ..}
+ handleS() {.. /* deal with s only */ ..}
+ processT() {.. /* deal with t only */ ..}
+ handleT() {.. /* deal with t only */ ..}

To raise cohesion, break the class ✓

Class B
- s
+ processS()
+ handleS()

Class C
- t
+ processT()
+ handleT()

V. Principles

Liskov Substituion Principle (LSP)

- LSP: Subclasses must be substitutable for their superclasses.
- -ve case: A subclass's object is not 100% a superclass's object.
  (Partial code reuse only, eg. BatteryDuck inheriting a Duck which has the eat() method)

Dependency Inversion Principle (DIP)

- DIP: High level modules should not depend upon low level modules. Both should depend upon abstractions.
  Abstractions should not depend upon details. Details should depend upon abstractions.
- -ve case: Classes depend too much on each other, changing one will lead to the change of another.
- Example 1: Topic04 P.10

  Employee, SalariedEmployee, HourlyEmployee; main, Company etc. depends on Employee

  Abstract Details Details High level Abstract

- Example 2: Lab12Q2 – Person, Playables (Football, Piano, + Chess!)

Open-Closed Principle (OCP)

- OCP: Modules should be open for extension, but closed for modification
- Robert Martin: "it should be easy to change the behavior of a module without changing the source code of that module."
- If OCP is applied well, then further changes can be done by adding new code, not by changing old code that already works. (Ref: Lab06 page 1 State-Pattern; Lab06-Q3: add "Disappeared Member")

VI. Design Patterns

- Design patterns are referred to as *best practices* to approach *common object-oriented design problems*
- Examples of design patterns
  - Singleton - Ensure a class has only one instance, and provide a global point of access to it.
  - State - Encapsulate a state as an object
    To allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
  - Command - Encapsulate a command as an object
    To let you support undoable operations.

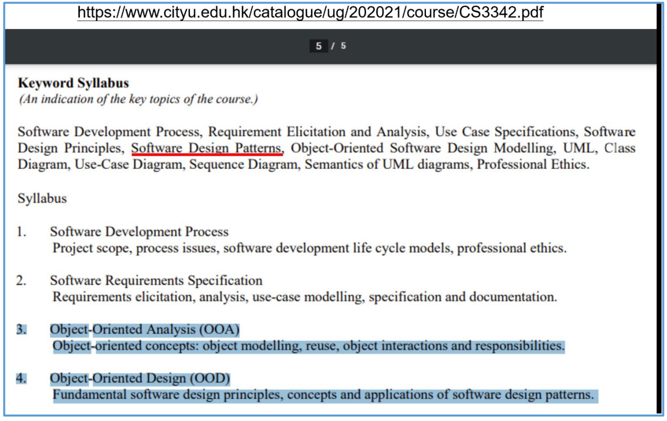
[A famous book]

Design Patterns - Elements of Reusable Object-Oriented Software

by *Gang of Four*
Erich Gamma / Richard Helm /
Ralph Johnson / John Vlissides

First published: 1994;
Latest print: 2016 (44th printing)

Looking forward to learn more in your next course:
CS3342 Software Design



https://www.cityu.edu.hk/catalogue/ug/202021/course/CS3342.pdf

5 / 5

Keyword Syllabus
(An indication of the key topics of the course.)

Software Development Process, Requirement Elicitation and Analysis, Use Case Specifications, Software Design Principles, Software Design Patterns, Object-Oriented Software Design Modelling, UML, Class Diagram, Use-Case Diagram, Sequence Diagram, Semantics of UML diagrams, Professional Ethics.

Syllabus

1. Software Development Process
   Project scope, process issues, software development life cycle models, professional ethics.

2. Software Requirements Specification
   Requirements elicitation, analysis, use-case modelling, specification and documentation.

3. Object-Oriented Analysis (OOA)
   Object-oriented concepts: object modelling, reuse, object interactions and responsibilities.

4. Object-Oriented Design (OOD)
   Fundamental software design principles, concepts and applications of software design patterns.