MT201 Computing Fundamentals with Java

Revision

Created by Vicmon FAN (Mr)
Optimized by Kenny YUEN (Mr)

Unit 0

Exam Guide

Unit 0 - Important Exam Remarks from CC
- No need to write comments
- No need to write import statements
- appendices of the units are not required, except those appear in the Specimen Paper and TMAs
- text book of King is not required, except the Readings that you are asked to read in units and they are not covered in the units.

Unit 0 - Appendix of specimen paper
- For String,
  - s1.compareTo(s2) will return 0 (if equal), <0 or >0 (depending on result of s1 - s2)
  - trim() will remove leading and trailing white spaces (but not the middle ones)
  - substring(iBegin, iEnd) will return a substring from index iBegin to iEnd - 1, e.g. "smiles".substring(1,5) = "mile"
- class method valueOf() is not mentioned in the appendix, but try to remember it and make good use of it. It can return the string of (nearly) all primitive type, e.g. String.valueOf(3.14)
- replace(c1,c2) is to replace all occurrence of old char (c1) with new char (c2)

Unit 0 - Appendix of specimen paper
- For collection,
  - toArray() is to return an array of all elements in collection, e.g., Collection coll = ...; Object array[] = coll.toArray();
- For File,
  - length() returns file size
- For JFrame,
  - setSize(i, j) where i is width, j is height (note: typo in appendix)
- For JTextArea,
  - JTextArea(i,j) where i is row, j is column
- Please add your own notes to those you are not familiar with

Unit 0 - Appendix of specimen paper
- For String,
  - s1.compareTo(s2) will return 0 (if equal), <0 or >0 (depending on result of s1 - s2)
  - trim() will remove leading and trailing white spaces (but not the middle ones)
  - substring(iBegin, iEnd) will return a substring from index iBegin to iEnd - 1, e.g. "smiles".substring(1,5) = "mile"
- class method valueOf() is not mentioned in the appendix, but try to remember it and make good use of it. It can return the string of (nearly) all primitive type, e.g. String.valueOf(3.14)
- replace(c1,c2) is to replace all occurrence of old char (c1) with new char (c2)

Unit 0 - Appendix of specimen paper
- For collection,
  - toArray() is to return an array of all elements in collection, e.g., Collection coll = ...; Object array[] = coll.toArray();
- For File,
  - length() returns file size
- For JFrame,
  - setSize(i, j) where i is width, j is height (note: typo in appendix)
- For JTextArea,
  - JTextArea(i,j) where i is row, j is column
- Please add your own notes to those you are not familiar with

Unit 0 - Exam Paper
- Time Allowed: 3 hours
- part I (55%), 11 short questions, ALL compulsory
  - Q1-Q10 covers each of the 10 units
  - Q11 is usually a hard question.
- part II (45%), long questions, choose 3 questions out of 4
  - more difficult questions

Unit 0 - Suggestion
- the format of actual exam paper will be somewhat similar to the specimen paper
- therefore try to do the specimen exam paper and understand the solutions
- Suggested Time Allocation for Questions by CC
  - spend about 9 minutes for each question in part I (totally about 100 minutes)
  - spend about 10 minutes in the selection of questions in part II (about 10 minutes)
  - spend about 20-25 minutes for each question in part II (totally about 70 minutes)
  - please note that the above time is the MAX time you can spend on one single question/task. If you find that you have spend more than the suggested time on any question, skip to next question first. You can always have chance go back to it after you finish your first attempt to all the questions.
Unit 0 - Suggestion

Suggested Sequence of Answering Questions

To make yourself feel confident, answer the easier
questions first, and skip the difficult question. (usually the
short questions are easier)

I suggest to try your best to do as fast as you can in the
first attempt

- don’t spend too much time on one single question
- if the question too difficult or too time consuming, skip
it first
- as you can always go back to it later if you have time
left

On the second attempt, try to estimate the time required
for the left-over questions against the time left. Then
answer the questions according to level of difficulties.

Finally, if you have completed the paper and still have
time left, try to read through the whole paper once.

Unit 0 - Suggestion

Difficult Questions

Even if you don’t know how to answer a whole
question, please try to answer the parts you know
since partial marks are likely to be given to the
correct statements.

For example in Q6 of the specimen paper, there
must be a statement printing two asterisks (***)
and writing such a statement is easy
(System.out.print('**');).

Unit 0 - order of revision

1. Skim through the specimen paper
   - get a feel on the style and scope of questions
   - this will give you a direction of revision, e.g.,
   - go straight to study those areas you don’t know how to do
   - study more on the topics where they are asked frequently
2. Revise your TMAs and understand your own strength and weakness
3. Study the course material accordingly
4. Do the revision paper
   - check against the solution and mark it to get your own score
   - make sure to understand why you made the mistake and review
     that topics again

Unit 1 - The history of computers

- Programmable computers - devices that can accept sequences of instructions
  (programs) and perform them
  - Early computers: mechanical binary switches
  - First generation modern computers: vacuum tubes
    - unreliable, hot, huge in size
  - Second generation: transistors (semi-conducting material)
    - smaller, faster, consume less power
  - Third generation: Integrated Circuit (IC)
    - transistors and others in a circuit
    - LSI, VLSI, ULSI
  - Personal Computer (PC)

Unit 1 - Type of Computer

- Special-purpose: TV game machine, digital camera
- General-purpose: desktop, notebook, handheld
- Personal computers (microcomputers): single-user
- Workstations: single-user, more powerful
- Mini-computers: multi-user, multi-processor
- Mainframes: organizational or enterprise-wise
- Supercomputers: extremely high speed, very expensive

Unit 1 - Number system

- Different number systems
  - Decimal
    - 1234₁₀ = 1x10³ + 2x10² + 3x10¹ + 4x10⁰
  - Binary
    - 1011₂ = 1x2³ + 0x2² + 1x2¹ + 1x2⁰
  - Octal
    - 456₈ = 4x8² + 5x8¹ + 6x8⁰
  - Hexadecimal
    - use 16 digits, 0 to F for representing number (where A=10, B=11,
      C=12, D=13, E=14, F=15)
    - 3BD₁₆ = 3x16² + 11x16¹ + 13x16⁰

Unit 1 - Number system

- Converting Hexadecimal to Binary
  3 B D
  0011 1011 1101
- Converting Binary to Hexadecimal
  0011 1011 1101
  3 B D
- Converting Octal to Binary
  4 5 6
  100 101 110
- Converting Binary to Octal
  100 101 110
  4 5 6

Unit 1

Computer Concepts and First Java Program
Unit 1 - The von Neumann architecture

- Arithmetic and Logic Unit (ALU)
- Control Unit (CU)
- Memory

• Proposed architecture of general-purpose computer
• Fetch-Decode-Execute cycle: The control unit fetches one instruction (encoded in binary format) at a time from the memory; then decodes it and executes it.

Unit 1 - A simplified general-purpose computer

- Central Processing Unit (CPU): brain of the computer
- Instruction Pointer (IP) (or Program Counter): a special register used for storing the location of the next instruction to be executed. The CU fetches instruction from memory according to IP.

Unit 1 - Computer Software

- Operating Systems (OS)
  • manage and coordinate computer resources (examples?)
  • provide security features
  • support multi-users to perform different tasks concurrently
  • provide GUI
- Application software
  • allow computer to perform specific tasks (e.g. word-processing)
  • must match operating system to work properly
  • cannot manipulate hardware directly and must access them using the services provided by OS (hence a 3-layer architecture)

Unit 1 - Programming Languages

- Low-level languages
  • 1GL - machine language - native language understood directly by computer (but language different for different computers)
  • 2GL - assembly language
- High-level languages
  • 3GL - use human-language-like statements (easy for us to read and understand)
  • 4GL - usually used for accessing database (SQL)
  • 5GL - support declarative programming (e.g. in AI area)
  • Notice that there is a gap between LL and HL languages

Unit 1 - Comparison of programming language

- Low-level language
  • speed: faster
  • portability: impossible
  • readability: difficult to learn, read or modify
  • selectivity: no choice
- High-level language
  • speed: slower (code not optimal after translation)
  • portability: good (with no or minor modifications)
  • readability: easy to learn, read or modify
  • selectivity: plenty of choices (e.g. C, C++, Java, etc.)

Unit 1 - How programs are executed

- How to fill up the gap between LL and HL languages?
  i.e. How to make the computer understand a HL language?
- Compilation
  • convert the HL language (source code) to machine language (object code or machine code)
  • execute the machine code (in binary format)
- Interpretation
  • use a software interpreter to translate the source code and then execute it
- Hybrid
  • compilation process translates the source code to an intermediate code, and the interpreter executes it (e.g. Java)

Unit 1 - Imperative (procedural) programming paradigm

- Program statements/instructions are executed one-by-one and applied with the data.
- Computer hardware implementations are basically imperative (which is the execution model of von Neumann architecture), and so are their low-level programming languages.
- As a consequence, most early high-level programming languages were also imperative.
- Programming languages like Basic, Pascal, C, etc. are typical imperative languages.

Unit 1 - Object-oriented (OO) programming paradigm

- Researchers found that modelling the real world entities was a better way to develop software.
- From the perspective of writing programs, a problem can be solved by first determining the involved agents (objects) and the messages that are sent among them. Then each agent is concentrated and programs written to perform all the determined operations.
- It is a more natural methodology for solving problems. Software developed using object-oriented programming is more robust, easier to debug and maintain, and it is easier to reuse the components built
Unit 1 - Java

- Java is a hybrid language involving both the object-oriented and imperative features (according to this course)
- It is hence up to the developers to write programs with object-oriented methodology or not.
- Common errors
  - Compile-time error: detect while compiling a Java source program (detected by Java compiler)
  - Run-time error: happen when running a compiled Java program (detected by JRE)

Unit 2

Problem solving through objects

Unit 2 - Phases of S/W Development Cycle

- Six Phases
  1. requirement
  2. analysis
  3. design
  4. implementation
  5. testing
  6. maintenance
- The SDC is cyclical, not linear. The cycle is also called the 'waterfall model'
- A single pass might only complete a portion of the features (or a module) of the system. Hence the entire software development is divided into iterations (first 5 phases).
- The iteration approach enables pipeline development.

Unit 2 - Requirement Phase

- We have to collect the requirements (needs) from the users or clients who asked you to develop the software
  - The requirements include limitations of the software, which are constraints such as budget, hardware and time.
  - The needs refer to the functionalities of the software.
  - Actions to be taken:
    - interviewing the clients and the users
    - find the deficiencies of the existing software
- In this phase, developers usually prepare a document known as a problem statement that summarizes the requirements of the clients, including the limitations of the software.

Unit 2 - Analysis Phase

- In this phase the problem statement is further investigated:
  - roles of the users of the software
  - functionalities (operations) the software supports
    - which operations are performed by which roles
  - operational scenarios of an operation under different conditions
  - business entities involved and their attributes / behaviours
- If not enough information was collected, we have to repeat the requirement phase until thoroughly clear about what the users expect and the details of all functionalities

Unit 2 - Design Phase

- Transform the requirements into technical designs.
- OO approach is adopted and developers try to model real-world scenarios and system operations by objects.
- Objects of the same type, and entities with the same set of behaviours and attributes, are defined with a single class definition. In this phase, these class definitions are designed.
- Operational scenarios are consolidated to determine a well-defined sequence of steps of behaviour.
- A list of classes, each with a list of attributes and behaviours (with well-defined steps) is produced

Unit 2 - Implementation Phase

- Convert the class designs derived in the design phase to actual class definitions in the chosen programming language (e.g. Java)

Unit 2 - Testing Phase

- Verify the programs by feeding test data to them and checking whether the outputs are exactly the same as the expected results
- But usually not all errors can be found and corrected if the system is large and sophisticated
- This is one reason we need maintenance to fix the errors found after software release
Unit 2 - Maintenance Phase

- Users might still uncover problems not detected before system release
- Software may need to be updated or enhanced according to the changes in business operations
- Maintaining a software system usually involves modifications of the source code
- Software fixes and patches are produced
- The maintenance cost throughout the whole period can even be greater than the sum of the first 5 phases

Unit 2 - Class diagram

<table>
<thead>
<tr>
<th>WashingMachine</th>
</tr>
</thead>
<tbody>
<tr>
<td>functionName : String</td>
</tr>
<tr>
<td>modelNumber : String</td>
</tr>
<tr>
<td>capacity : double</td>
</tr>
<tr>
<td>timeRequired : double</td>
</tr>
<tr>
<td>consumptionRate : double</td>
</tr>
<tr>
<td>height : double</td>
</tr>
<tr>
<td>width : double</td>
</tr>
<tr>
<td>depth : double</td>
</tr>
<tr>
<td>start ()</td>
</tr>
<tr>
<td>top ()</td>
</tr>
<tr>
<td>setTimer (hour : int)</td>
</tr>
<tr>
<td>getTimeElapsed () : int</td>
</tr>
</tbody>
</table>

Figure 2.4 A class definition for a washing machine

Unit 2 - Classes and objects

- A class is the blueprint for a group of objects of the same category (having same sets of attributes and behaviours).
- Whenever an object receives a message, it performs the behaviour according to the type of the messages and the supplementary data. The behaviour can be updating its attributes, giving a feedback, or subsequently sending messages to other objects to trigger them to perform other operations.
- It is not the message sender’s concern how the recipient object perform an operation (it only concerns “what”).
- Hence each object performs its own part and all objects cooperate to constitute a complex operation of the system

Unit 2 - Determining (i) objects and (ii) their attributes

- Potential candidates can be obtained by highlighting the nouns and noun phrases in the sentences of the problem statement, which may be the entities or objects that must be modelled in the software.
- Some of the nouns are objects, some object attributes and some supplementary information a behaviour.
- Group the nouns that refer to the same type of entity and give it a name (i.e. a class name).
- You should determine whether an object comprises other identified objects. If so, the object has an attribute for each component object.
- If an object always sends messages to a few fixed objects, the corresponding class definition has attributes for them

Unit 2 - Determining (iii) object behaviours

- By identifying the verbs in the problem statement, the list of potential object behaviours can be obtained.
- Determine the object (and hence the class) that has the behaviours.
- The scenarios presented in the problem statement can be modelled by message-sending among objects.
- Should also verify whether the behaviours need supplementary information to perform the behaviour

Unit 2 - General class and its specific classes

- We can extract the common attributes and behaviours among classes. A new class is defined that has them; all other classes are defined as extensions.
- Each class of extension has all the attributes and behaviours of the base class. This phenomenon is known as inheritance.
- In a class hierarchy, an arrow is drawn from the specific class pointing to the general class. It clearly shows the general class and all its specific class.
- A specific class has all the attributes and behaviours of its general class. If the specific class needs a behaviour to be performed differently, it can override the method

Unit 2 - Class hierarchy

- We can extract the common attributes and behaviours among classes. A new class is defined that has them; all other classes are defined as extensions.
- Each class of extension has all the attributes and behaviours of the base class. This phenomenon is known as inheritance.
- In a class hierarchy, an arrow is drawn from the specific class pointing to the general class. It clearly shows the general class and all its specific class.
- A specific class has all the attributes and behaviours of its general class. If the specific class needs a behaviour to be performed differently, it can override the method.
Unit 3

Classes and objects

Unit 3 - Identifier

Identifiers are used for names of classes, variables and methods

- **Rules:**
  - consist of any letters, digits and the underscore character
  - must start with a letter or the underscore character (but not digit)
  - no limit on length, case-sensitive
  - cannot use Java keywords (e.g. int, double) or reserved words

- **Recommended:**
  - use nouns for names of classes and variables (e.g. scores); use verbs for names of methods (e.g. computeAverage)
  - class identifiers should start with uppercase letter (e.g. MyClass)
  - if the identifier contains more than one word, start each word (except the first one) with an uppercase letter (e.g. noOfStudents)

Unit 3 - Implicit and explicit initialization

- If you declare variables without setting up an initial value to each of them, Java will initialize them for you using these conventions:
  - byte, char, short, int, long: set to 0
  - float, double: set to 0.0
  - boolean: set to false

- reference variables: set to null

- The above is known as implicit initialization. It occurs to instance variables when an object is created. Local variables (e.g. those defined in a method) will not be initialized implicitly.

- Of course you can explicitly initialize variables yourself.

  type identifier = initial-value;
  e.g. int maxSeats = 50;

Unit 3 - Accessing object members

- Objects members, including variables and methods, can be accessed using the dot-notation. For example:

  student.name  
  student.computeTotalScores( )

- It is preferable to send a message to an object to update its variables than to directly change their values, because the object’s methods should contain the correct way to update its variables.

- Hence it is always a good habit to write some setter and getter methods for the variables in a class.

Access control - private and public

- Access control is used to limit the accessibility of object attributes and methods by other classes.

Unit 3 - Primitive data types

- There are 8 primitive data types:
  - byte, char, short, int (default), long
  - float, double (default)
  - boolean.

- **Integral types:**
  - byte - 8 bits (-128 to 127)
  - short - 16 bits (-32768 to 32767)
  - int - 32 bits (-2^31 to 2^31 - 1)
  - long - 64 bits (-2^63 to 2^63 - 1)

Q: Why are the results of the following assignments?

  float f1 = 99999; // OK
  float f2 = 99999.0; // error: possible loss of precision
  float f3 = 99999.0F; // OK

- **Boolean type:** can only store true or false

Unit 3 - Primitive vs. non-primitive types

- **Character type:** represented in 16-bit Unicode

- A character can represent a single-byte English alphabet or a double-byte Chinese character (and also many other languages)

- Examples:

  char gender = ‘M’;
  char unicode = ’uFFFF’;
  char newLine = ‘n’;
  char carriageReturn = ‘v’;
  char tab = ‘t’;
  char whatIsThis = 65; // character A

- **Reference types (classes):** can only refer to objects

- Access control is used to limit the accessibility of reference types and non-primitive types by other classes.

- Non-primitive types (classes)
  - known as reference types (referring to objects)
  - declare a non-primitive type variable only allocates the memory for storing the reference (an address)
  - actual memory for storing the object is allocated when the object is created
Unit 3 - Implicit and explicit type conversions

- Java allows the types on LHS and RHS of an assignment to be different (with some limitations). This is called type conversion (converting RHS to LHS).

- **Implicit type conversion** (automatic conversion by computer)
  - the sole restriction is that the variable on LHS must have sufficient storage (number of bits) to store the value on RHS
  - in Java, an integer value can always be assigned to a real type variable (but not the other way round)
  - hence assignment fails if the assignment loses its precision or variable does not have sufficient storage to store the value

Unit 3 - Assignment compatibility

<table>
<thead>
<tr>
<th>Real type</th>
<th>Integer type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double (64-bit)</td>
<td>long (64-bit)</td>
</tr>
<tr>
<td>float (32-bit)</td>
<td>int (32-bit)</td>
</tr>
<tr>
<td></td>
<td>char (16-bit)</td>
</tr>
<tr>
<td></td>
<td>short (16-bit)</td>
</tr>
<tr>
<td></td>
<td>byte (8-bit)</td>
</tr>
</tbody>
</table>

A value of a type that occurs lower in the above diagram can be converted automatically to a value that occurs higher. But it is not possible to do it the other way round.

Unit 3 - Explicit type conversion

Explicit type conversion

- In order not to be bounded by the compatibility restriction, you can explicitly forced a conversion by casting.
  
  `type variable = (type) value;`

- Assigning a `char` type value to a `short` type variable or vice versa needs casting (but both are 16-bit).

Unit 3 - Class packages

- We can organize the class definitions using class packages.
  ```java
  package package-name;
  class Class-name {
    ...
  }
  ```

- There is at most one package declaration per Java source file.

- If we want to refer to a class in another package (provided that it is accessible), we need to again use the dot-notation (the fully qualified name):
  ```java
  package-name.Class-name
  ```

- To prevent having to repeat the lengthy fully qualified names, the `import` declaration can be used (put at the beginning of file):
  ```java
  import package-name.Class-name;
  import package-name.*;
  ```

Unit 3 - Class members

- Instead of each object having its own copy of instance variables, we also have the need to share some variables between all objects of the same class. These are called class variables.

- Similarly we also have class methods which are for the class.

- To define class members, we use the `static` keyword.

- When to declare class variables?
  - As global variables (all objects of the class can see them)
  - As constants (should also use the `final` keyword)

- To store data for class methods (class method cannot access instance variables)

- To access a class variable, again use the dot-notation:
  ```java
  Class-name.class-variable (c.f. object-name.instance-variable)
  ```

Unit 4 - Comparing primitive data

- Integral division gives an integer result only, which might involve truncation (ignoring the decimal part).

- `a / b * b == a`

- is true in mathematics, is not necessarily true in computers

- Floating point calculation results are also just approximations because of rounding-off

- Instead of comparing `a == 0.85`

- Use `Math.abs(a - 0.85) < 0.0000001` where `0.0000001` is an arbitrary value of the tolerable round-off error

Unit 4 - Control structures

Relational Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td><code>==</code></td>
<td>Equal to</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Not equal</td>
</tr>
</tbody>
</table>

- The above six operators can be used for all numeric primitive:
  - byte, short, char, int, long, float and double
  - Results either true or false
  - `!=` means not equal to
  - Exclamation mark (!) in Java means ‘negation’ or ‘not’
## Unit 4 - Method equals()

- To compare the states of objects
  - Use method `equals()` instead of `==` operator
  - Class must implement its own `equals()`
  - Inherit from `java.lang.Object`
  - The default behaviour is just like `==`
  - For example, class `String` overrides this method to compare the content of string objects

## Unit 4 - Precedence and Associativity

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Associativity</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+ -</code></td>
<td>Right</td>
<td>Unary operators. eg. <code>A * (-2)</code></td>
</tr>
<tr>
<td><code>/ %</code></td>
<td>Left</td>
<td></td>
</tr>
<tr>
<td><code>&gt;= &lt; &lt;=</code></td>
<td>Left</td>
<td></td>
</tr>
<tr>
<td><code>== !=</code></td>
<td>Left</td>
<td></td>
</tr>
<tr>
<td><code>+= -= *= /= %=</code></td>
<td>Right</td>
<td>Assignment operators</td>
</tr>
</tbody>
</table>

- `a + b * c > 10`
- `((a + (b * c)) > 10)`
- `a * b = c`
- `(a * (b + c))` or `a = a * (b + c)`

## Unit 4 - Java Branching

- A control structure that allows a program to conditionally perform a group of instructions based on a Boolean expression result or the result of a condition so that there is a different path or flow of control

- Java supports two branching structures,
  - `if/else` statement
  - `Two-way branching`
  - `switch/case` statement
  - `Multiple branching`

## Unit 4 - Looping

- Looping structures enable a statement or a block of statements to be executed repeatedly.

- A looping structure consists of:
  1. `loop body to be executed`
  2. `loop condition to determine whether the loop has to continue`
  3. `initialization part that sets up the variables`
  4. `update part that might affect the loop condition`

- Java provides 3 looping structures that support different combinations and arrangements of the above four parts
  1. `while`
  2. `do/while`
  3. `for`

## Unit 5 - Creating and using arrays

- An array is implemented as an `object` in the Java programming language.
- The data stored in an array are known as the elements of the array
- All elements of an array must be of the same type (either primitive or object)

### Array

- The content is the reference of the array object
- The content is an array object
- The elements are of the same type as the array object

### Figure 5.2 A typical scenario of using an array (p-6)
Unit 5 - Declaring arrays

To declare an array variable

- **Type[] Variable-name;** // preferable
- **Type Variable-name[];**

```java
double[] tmaScores;
double tmaScores[];
Student[] g8Students;
String[] args; // as in main()
int[] a, b;   // both a & b of type int[]
int a[], b;   // only a of type int[]
```

Remember that array is itself an object - when array is declared, no array object is created. We need to use keyword `new` to create it explicitly.

Unit 5 - Creating arrays

To create array object

- new **Type[n]**  // n is number of elements (non-negative)
- `days = new int[12];`

Array cannot be resized once created

To declare and create array in one statement

- `int[] days = new int[12];`

![Figure 5.6](image)
The interpretation of the expression `days[0]`

Unit 5 - Creating arrays

The elements of all newly created array objects are automatically initialized.

Table on right shows the initial values of each element for different array element types:

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Dimensional initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int[]</code></td>
<td>0</td>
</tr>
<tr>
<td><code>double[]</code></td>
<td>0.0</td>
</tr>
<tr>
<td><code>String[]</code></td>
<td>&quot;null&quot;</td>
</tr>
<tr>
<td><code>Student[]</code></td>
<td>Student()</td>
</tr>
</tbody>
</table>

Unit 5 - Accessing elements in arrays

It is necessary to provide a subscript (or index) to access a particular array element:

- **ArrayVariable[substring]** // subscript start with 0

If an array has n elements, its index is ranged from 0 to n-1.

```java
double[] tmaScores = new double[4];
```

| int size = tmaScores.length; // size of array, not method name |

Unit 5 - Initializing Arrays

- Combined declaration, construction and initialization
  - `int[] fontSize = { 9, 11, 13, 15, 17 };`
  - `String[][] fontDesc = {
    { "TimesRoman", "bold" },
    { "Courier", "italic" },
    { "Tahoma", "normal" };
  }
  - `String[] stuNames = { "Peter", "Paul", "Mary" };`
  - `String[] stuNames = { new String("Peter"), new String("Paul"), new String("Mary") };`
  - `Student[] students = { new Student("Peter"), new Student("Paul"), new Student("Mary") };`

Unit 5 - Increment & Decrement Operator

- If the operator is placed before a variable, it is known as the pre-increment operator or pre-decrement operator
- If the operator is placed after a variable, it is known as post-increment operator or post-decrement operator.

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Dimensional initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int[]</code></td>
<td>0</td>
</tr>
<tr>
<td><code>double[]</code></td>
<td>0.0</td>
</tr>
<tr>
<td><code>String[]</code></td>
<td>&quot;null&quot;</td>
</tr>
<tr>
<td><code>Student[]</code></td>
<td>Student()</td>
</tr>
</tbody>
</table>

Table 5.3
Comparing different increment & decrement operators based on an initial value of i = 5

<table>
<thead>
<tr>
<th>Statement</th>
<th>Equivalent statements</th>
<th>Values of the variables after executing the statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = ++i;</td>
<td>1 = i+1; j = i;</td>
<td>6</td>
</tr>
<tr>
<td>j = i++;</td>
<td>j = i; 1 = i+1;</td>
<td>6</td>
</tr>
<tr>
<td>j = --i;</td>
<td>1 = i-1; j = i;</td>
<td>4</td>
</tr>
<tr>
<td>j = i--;</td>
<td>j = i; 1 = i-1;</td>
<td>4</td>
</tr>
</tbody>
</table>

Unit 5 - Searching

In programming, we often need to check
- if a data item exists in a collection
- if a data item fulfill certain conditions
- two common ways to search for data in an array
  - Linear search / sequential search
  - Binary search
Unit 5 - Linear/Sequential Searching

- Easier to implement
- Access each array element sequentially (one by one)
- Not effective to search for a number in huge of data
- Data is not need to be sorted beforehand

```java
public class LinearSearcher {
    private int numbers[] = new int[1000];
    private int total;
    public void storeNumber (int number) {
        if (total < numbers.length) {
            numbers[total++] = number;
        } else {
            System.out.println("Too many numbers");
        }
    }
    public boolean contains(int target) {
        boolean found = false;
        for (int i = 0; i < total; i++) {
            if (numbers[i] == target) {
                found = true;
                break;
            }
        }
        return found;
    }
}
```

Unit 5 - Binary searching

- Much faster, esp. if amount of data is huge
- Data in the array must be sorted
- For a list of sorted data:
  1. separate the searching scope into two halves
  2. verify whether the target data exist in the first half or in the second half
  3. set the half that contains the target data as new searching scope
  4. repeat steps 1 to 3 until there is only one element left in the searching scope

---

Unit 6 - Short-circuit evaluation of logical expressions

- The evaluations of the && and || operators in Java (like in some other languages) are short-circuit.
- If the result of the left sub-condition can determine the result of the entire condition, the right sub-condition is not evaluated.
- The reason is that no matter what the result of the right sub-condition is, it won’t affect the overall result of the entire condition.

| A  | B  | A && B | A  | B  | A || B |
|----|----|--------|----|----|--------|
| true | true | true  | true | don’t care | true |
| true | false | false | false | true | true |
| false | don’t care | false | false | false | false |

- The && and || operators implicitly eliminate unnecessary boolean sub-expressions for performance consideration.

---

Unit 6 - Non-short-circuit operators for logical AND/OR

- There are equivalent but non-short-circuit version of logical AND and OR operators: & and | respectively.
- These two operators have exactly the same truth table as their respective short-circuit version.
- Normally we should choose to use the short-circuit version (for performance consideration) unless the sub-expressions contain side-effects which we want to realize them.
- For example, compare these two program fragments.
  ```java
  int i = 0, j = 0;
  boolean x = (i++ > 0 && ++j > 0);
  boolean x = (i++ > 0 & ++j > 0);
  ```
  - We should try to avoid boolean sub-expressions with side-effects because they are difficult to understand and maintain.

---

Unit 6 - Bitwise operators

- The &, | and ^ operators are not only boolean relational operators. They are also bitwise operators to be used with two integral values.
- We illustrate these bitwise operators using 3 examples. Assume we have 2 integers of byte-type (i.e. 8 bits). We have:
  ```java
  25 & 101 = 1
  25 | 101 = 125
  25 ^ 101 = 124
  ```
  - There is also a bitwise negation (or complement) operator ~ which can be applied on a single integral value. Hence we have
  ```java
  ~25 = -26
  ```

---

Unit 6 - Advanced constructs

- Ternary condition statement in Java:
  ```java
  condition? value-for-true: value-for-false
  ```
  - For example:
    ```java
    remarks = (marks >= 80)? "Good": ((marks >= 40)? "Pass": "Fail");
    ```
  - Executing a break statement in a loop will terminate the looping structure at once.
  - Hence subsequent statements after the break statement in a loop are skipped, and the loop is terminated unconditionally.
  - On the other hand, executing a continue statement in a loop will transfer the flow of control to the point just before the end of the loop.
  - Hence subsequent statements after the continue statement in a loop are skipped. But the loop is not terminated and the loop structure behaves as usual.
Unit 6 - Multidimensional arrays

- Elements in a one-dimensional array is considered arranged as a row. Hence elements in a two-dimensional array is considered arranged as a table (with rows and columns).
- Java supports multidimensional arrays with no logical limit of dimensions. And the content of each array element is initialized in the same way as in a one-dimensional array.
- Example: declaring, creating and accessing a 2-D array

```java
int[][] twoDimArray = new int[10][20];
twoDimArray[0][0] = 999; twoDimArray[9][19] = 123;
String[][] sittingPlan = {{“Paul”, “Mary”}, {“Peter”, “John”}};
```

Unit 6 - Recursion

- Recursion is a problem-solving technique that:
  - For a given problem it is resolved to another identical problem but with a smaller size.
  - The approach is repeated until a well-defined problem with answer is obtained. This well-defined problem is known as the base case.
  - The solution to the base case is used to synthesize the solution of a larger problem that is closer to the original problem.
  - Hence the solutions to the smaller problems can be used to synthesize the solutions to the larger problems and ultimately the original problem.
- There are many problems that can be solved naturally using recursive approach. Unit 6 shows you two such examples: calculating the factorial (P.92) and finding GCD using the Euclidean algorithm (P.98-99).

Unit 6 - Recursion

```java
public int remainder(int m, int n) {  // to find remainder of m where n is divisor
  int result;
  if (m < n) {
    result = m;  // base case where we can no longer subtract m by n
  } else {
    result = remainder(m - n, n);  // otherwise, subtract m recursively
  }
  return result;
}
```

Unit 7 - Setter and Getter methods

- For an object attribute, a class definition can:
  - provide both setter and getter methods
  - attribute being made readable and writeable
  - Provide either one of them
  - attribute being made either readable only, or writeable only.

```java
private int hour;
// getter
public int getHour() {
  return hour;
}
// setter
public void setHour(int theHour) {
  hour = theHour;
}
```

Unit 7 - Process of Creating an Object with keyword new

1. Memory allocation.
   - A software object occupies memory space in the Java Virtual Machine (JVM), and the JVM allocates the necessary memory for it first.
2. Implicit initialization.
   - All instance (non-class or non-static) variables of the object are implicitly initialized. (0 for number; false for boolean; null for non-primitive)
3. Explicit initialization.
   - If an instance variable is declared with initialization, it is initialized according to the initialization expression.
4. Execution of the constructor.

Unit 7 - Constructor

- constructor of a class can be treated as a special method that is executed during the process of creating an object
- a constructor has no return type and the name must match the class name, e.g., for class TicketCounter:

```java
public TicketCounter() { // without parameter
  reading = 0;
}
public TicketCounter(int theReading) { // with parameter
  reading = theReading;
}
```

Unit 7 - More on Classes

More on Classes
if a class defines no constructor, a default constructor or a 'no-arg' constructor with an empty parameter list and an empty method body is implicitly added to it.

```java
public TicketCounter() {
    // No statement
}
```

That is why we can create objects without defining any constructors.

But if a class defines a constructor, no default constructor will be added implicitly.

---

### Unit 7 - Constructor

- If a class defines no constructor, a default constructor or a 'no-arg' constructor with an empty parameter list and an empty method body is implicitly added to it.
- `public TicketCounter() { // No statement }`

That is why we can create objects without defining any constructors.

But if a class defines a constructor, no default constructor will be added implicitly.

---

### Unit 7 - Keyword final

- The keyword `final` is applicable to member variables (both class/static and instance) and local variables.
- Besides final variables, it is possible to mark a method or a class final (see appendix A).

```java
public void changeParam(final int param) {
    System.out.println("The value of param = " + param);
    param = 1; // compile-time error
    System.out.println("The value of param is set to " + param);
}
```

---

### Unit 7 - Inheritance

- The `has-a` relationship concerns the possession of an object by another one.
- A washing machine has an incoming valve, a drum motor and an outgoing valve.
- If a ClassA object has a ClassB object, the definition of the class ClassA defines an instance variable of type ClassB to refer to a ClassB object.

---

### Unit 7 - Polymorphism

- Ability for objects that are subclasses of the same parent class to behave differently on receiving the same message that is defined by the common parent class.
- In Java, it is implemented as a variable of a superclass type, say class T, that can refer to an object of any subclass of class T.
- If a message is sent to the object via the variable of type class T, the behaviour (or the method to be executed) is determined by the actual class of the object (that is, a subclass of class T or class T itself), not by the type of the variable.
Like the usual methods, it is possible to write overloaded constructors. That is, a class can define more than one constructor and there is more than one way to create an object of the class.

The rules governing the method overloading are applicable to the overloading of constructors.

It is possible to have more than one constructor, provided that the parameter lists are different in number of parameters and/or parameter types.

Java enables a constructor to call another constructor as part of the creation process, by using the keyword this.

the statement must be the first statement in the method or compilation fails.

the keyword this can also be considered an object variable that refers to the object itself
it is commonly used in the following two situations:
1. explicitly referring to the object behaviours and attributes
   public void setReading(int reading) {
       this.reading = reading;
   }
2. calling a method with keyword this as parameter values ( e.g. button.addActionListener(this);

constructors are handled differently with respect to inheritance. A subclass does not inherit constructors from its parent class.
If Date1 class defines no constructor, a default constructor is added implicitly to the class definition.
For a constructor without the this(parameter-value-list) statement as the first statement, an implicit super() is added before all statements in the constructor.

abstract method is a method definition marked abstract
public abstract return-type method-name(parameter-list);
If a class defines one or more abstract methods, the class has to be marked abstract as well.
Because abstract classes model a conceptual idea or a general type, it is not possible to create an object of an abstract class.
if a subclass does not define all abstract methods defined in the abstract superclass, the subclass is still abstract because it still has abstract methods, and it has to be marked abstract or compile-time errors will occur.
Unit 7 - Abstract class with concrete subclasses

// Definition of class Pet
public abstract class Pet {
    // No Attributes
    // Abstract method to be defined in concrete subclasses
    public abstract void sayAWord();
    // The concrete way a pet to say a few words.
    public void sayAFewWords() {
        // Say arbitrary number of words
        int count = (int) (Math.random() * 10) + 1;
        // Use a for loop to say a word for a few times
        for (int i = 0; i < count; i++) {
            sayAWord();
        }
        // Skip to next new lines
        System.out.println();
    }
}

Unit 8 - File

- some methods defined in File class:
  public boolean isAbsolute()
  public boolean isDirectory()
  public boolean isFile()
  public boolean isHidden()
  public long lastModified()
  public long length() // the size of the file

- E.g.:
  File start = new File("c:\\java");
  File[] files = start.listFiles();
  String[] names = start.list();

Unit 8 – I/O stream

- Reading a file can be viewed as a flow of data stream from the outside world into the JVM - input stream / source stream.
- Similarly writing a file can be viewed as a flow of data stream from the JVM to the outside world - output stream / destination stream.
- In Java we can read/write data in units of 8-bit bytes or 16-bit Unicode characters. They are known as byte stream and character stream respectively.
- Byte stream is used to handle (byte-oriented) binary files, e.g. image files, audio files
- Character stream is used to handle (character-based) textual data, e.g. text files

Unit 8 – Byte Stream

- 2 most common ways to create a FileOutputStream object:
  1. FileOutputStream outFile = new FileOutputStream(outFile);
  2. FileOutputStream outFile = new FileOutputStream(pathString);
  - Note that if the specified file already exists, creating an output stream onto that file will overwrite it unless we use:
    FileOutputStream(File file, boolean append)
  - As mentioned, the write() method has been overloaded in the OutputStream class:
    public void write(int b) // int but not byte type… how it works?
    public void write(byte[] b) // no. of bytes read is returned
    public void write(byte[] b, int offset, int length)

- Note that we could check whether we have read up to the end of a file by the condition: in.read() == -1 (note that EOF is user-defined)
**Unit 8 – Character Stream**

- 2 most common ways to create a FileWriter object:
  1. `File outFile = new File(pathString);`
     `Writer writer = new FileWriter(outFile);`
  2. `Writer writer = new FileWriter(pathString);`

- The overloaded `write()` method in the `Writer` class:
  - `public void write(int c) // note int but not char type`
  - `public void write(char[] c)`
  - `public void write(char[] c, int offset, int length)`
  - `public void write(String str)`
  - `public void write(String str, int offset, int length)`

**Unit 8 – Character Stream**

- 2 most common ways to create a FileReader object:
  1. `File inFile = new File(pathString);`
     `Reader reader = new FileReader(inFile);`
  2. `Reader reader = new FileReader(pathString);`

- The overloaded `read()` method in the `Reader` class:
  - `public int read() // the char data read is returned as int`
  - `public int read(char[] buf) // no. of chars read is returned`
  - `public int read(char[] buf, int offset, int length)`

- Note that we could check whether we have read up to the end of a file by the condition: `reader.read() == -1`

**Unit 8 – Stream filter**

- Filter connects to InputStream or OutputStream
- Performs some transformation on data or provides some additional functionality
- E.g. `BufferedXXXStream`, `DataXXXStream` where XXX is Input or Output

```java
try {
    FileInputStream fis = new FileInputStream("abc.dat");
    DataInputStream dis = new DataInputStream(fis);
    double d = dis.readDouble();
} catch (IOException e) {}  
```

**Unit 8 – Stream reader/writer**

- `InputStreamReader` and `OutputStreamWriter` bridge between byte stream & character stream and take into account of the locale of OS

```java
FileInputStream fis = new FileInputStream("chinese.txt");
InputStreamReader isr = new InputStreamReader(fis, "BIG5");
// if default system is not BIG5
char c = (char) isr.read();
```

- `FileReader` is a shorthand of above statements (no need to prepare a byte stream, but less flexible)

```java
Reader reader = new FileReader("chinese.txt");
// default coding system only
```

**Unit 8 – Exception**

- An exception happens when a Java program performs an illegal operation (e.g. division by 0 at runtime), or when an exception condition is experienced (e.g. creating a `FileInputStream` on a non-existing file).
- The exception is generated to tell the program that a problem occurs, and allow the program to perform suitable remedial action.
- When an exception is raised, an exception object is created containing the information about the exception conditions or problems. Java has predefined a set of common exception types (as `Exception` subclasses).
- Examples include `NumberFormatException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`, `ArithmeticException`, `NullPointerException`, `IOException` (subclass `FileNotFoundException`)

**Exception hierarchy**

```
Exception hierarchy

Throwable
  Error
  Exception
    ...(Exception)
    RuntimeException
      ...(unchecked exceptions)
      IOException
        ...(checked exceptions)
```

**Unit 8 – Exception**

- There are 2 ways to deal with exceptions in a Java program: handle the exception in place or pass the responsibility to the caller.
- To handle an exception, we can use a `try/catch/finally` block:

```java
try {
    // codes that cause a potential checked exception
    // catch (Exception-class1 variable1) // more specific exception
    // codes to execute in case of exception1
    // catch (Exception-class2 variable2) // more general exception
    // codes to execute in case of exception2
    // catch (...) {
    //   ...  
    // finally {
    //   // codes to execute no matter whether there is or isn’t exception
    
```

**Unit 8 – Exception**

- If you prefer not to handle the exception in the method codes, you can pass the responsibility back to the caller to handle it when the exception is raised.
- This can be achieved by declaring a method to throw the potential checked exception types (but not recommended to use in our course)

```java
return-type method-name(parameter-list) throws exception-list
```

- When an exception is raised in a method that throws it, execution in the method is terminated immediately and control is returned to the caller.
- The caller (which is also inside a method) can again choose to handle the exception or throw the exception to its own caller.
Unit 8 - Example

```java
public void textFilter(String binaryName, String textName) {
    final int EOF = -1; // note: EOF is undefined
    int byteRead;
    try {
        InputStream in = new FileInputStream(binaryName);
        OutputStream out = new FileOutputStream(textName);
        while ((byteRead = in.read()) != EOF) {
            if (byteRead >= 32 && byteRead <= 127) {
                out.write(byteRead);
            }
        }
        in.close();
        out.close();
    }
    catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Unit 9 - JFrame

- Constructors:
  - public JFrame()
  - public JFrame(String title)
- Methods:
  - void show()  void hide()  void setVisible(boolean visible)
  - void setSize(int width, int height)
  - void setDefaultCloseOperation(int action)  e.g. EXIT_ON_CLOSE

```java
MyFrame aFrame = new MyFrame();
aFrame.show(); // When a JFrame object is created, it is invisible
```

Unit 9 - JComponent

** JButton **

- Constructors:
  - public JButton()
  - public JButton(String label)
- Method:
  - void setText(String label)

** JLabel **

- Constructors:
  - public JLabel()
  - public JLabel(String text)
- Methods:
  - void setText(String label)
  - String getText()  void setEditable(boolean editable)

Unit 9 - Swing

- JComponent is the base class for all Swing components except top-level containers.
- To use a component that inherits from JComponent, you must place the component in a containment hierarchy whose root is a top-level Swing container.
- Top-level Swing containers -- such as JFrame, JDialog, and JApplet -- are specialized components that provide a place for other Swing components to paint themselves.
- A component is added to the JFrame's content pane

Unit 9 - Example

```java
// step 1 - write event handler class that implements the listener interface
public class MyFrame extends JFrame implements ActionListener {
    private JButton button = new JButton("Click Me");
    public MyFrame() {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);
        pack(); // set size of frame and its component to optimal
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // step 3 - register the event handler object with the GUI component
        button.addActionListener(this);
    }
    // step 2 - implement event handler
    public void actionPerformed(ActionEvent ae) {
        button.setText("Clicked");
    }
}
```
Unit 9 - JMenuBar, JMenu and JMenuItem

For many applications, a (pull-down) menu is provided for accessing the functionalities of the system.

The steps in adding a menu bar to a JFrame object:
1. Create a JMenuBar object and add it to the JFrame, using its setJMenuBar() method
2. For each menu on the menu bar, create a JMenu object and add it to the JMenuBar
3. For each menu item in the menu, create a JMenuItem object and add it to the JMenu

The order of adding the JMenu objects and JMenuItem objects are important as it determines the appearance of the whole menu.

To add a line as separator into a menu, use the addSeparator() method of the JMenu object.

Unit 9 - Canvas

A canvas is a GUI component that models a drawing area for drawing anything (text, lines, figures, etc.).

Note that the Swing packages do not provide a Swing version of canvas. We make use of the one in the AWT packages.

Constructors: public Canvas()

Note that the preferable size of a Canvas object is 0 x 0. You can use the setSize(width, height) method to set its size.

The usual practice of creating a canvas is to write a subclass of the Canvas class and override the paint() method. The paint() method is called to construct the appearance of the canvas whenever it is uncovered.

public void paint(java.awt.Graphics g) { … }

Unit 9 - BorderLayout

The NORTH and SOUTH components may be stretched horizontally;
the EAST and WEST components may be stretched vertically;
the CENTER component may stretch both horizontally and vertically to fill any space left over

The default layout manager of the content pane of a JFrame object
arrange and resize its components to fit in five regions: north, south, east, west, and center (default).
Each region may contain no more than one component

cp.setLayout(new BorderLayout());
cp.add(new Button("OK"), BorderLayout.SOUTH);

Unit 9 - FlowLayout

The default layout manager of JPanel object.
give component its preferred sizes
does not resize component
add components from left to right & start a new row when first one is filled
components that doesn't fit are simply not shown
Unit 9 - GridLayout

- available space is divided into a grid of equal-sized cells & components are forced to fit
- with rows being filled from left to right, starting at top left

Unit 9 - JPanel

- In Swing, the JPanel object is a GUI component which itself is a container and can be added to another container.
- By embedding JPanel objects within other containers, we can design complex GUI layouts by playing with the layout manager of each of these containers.
- The default layout manager of a JPanel is a FlowLayout object.
- JPanel is actually a subclass of Container. So methods that can be applied to a Container object can also be applied to a JPanel object, e.g. add(), setLayout(), etc.
- e.g. to create a button panel
  ```java
  JPanel buttonPanel = new JPanel();
  buttonPanel.add(okButton);
  buttonPanel.add(cancelButton);
  ```

Unit 9 - Dialog

- A dialog is a special type of frame for displaying messages (which needs user’s immediate attention) or getting user input.
- In Swing, some standard dialog boxes can be created from the class JOptionPane.
- Dialog for showing a message:
  ```java
  public static void showMessageDialog(…)
  ```
- Dialog for getting confirmation:
  ```java
  public static int showConfirmDialog(…)
  ```
- Dialog for getting user input (as String):
  ```java
  public static String showInputDialog(…)
  ```
- User-defined dialog: creating a JDialog object

Unit 9 - JOptionPane

```java
int iAns;
String sReason;
iAns = JOptionPane.showConfirmDialog(null, "Do you enjoy studying MT2017?");
if (iAns == JOptionPane.YES_OPTION) {
  JOptionPane.showMessageDialog(null, "Thank you for taking the course!");
} else if (iAns == JOptionPane.NO_OPTION) {
  do {
    // return null if click "cancel"
    // return "" if click "OK" but input nothing
    sReason = JOptionPane.showInputDialog("Why?");
  } while (sReason != null && sReason.equals(""));
}
```

Unit 9 - Event handling

- The actions taken by a user are called events and we need to write codes for event-handling.
- In order to handle a particular event, we need to register an event handler/listener object with the respective GUI component.
- The scenario of event-handling is hence like this. When user operates on a GUI component (e.g. clicking a button), an event object is created. A particular method of the registered event handler object is called with reference to the event object. Then the method can determine the information about the event and perform operations accordingly.

Unit 9 - Event Handling

- Events of GUI are classified into different categories (e.g. action, focus, mouse, key, window, etc.), and listener classes (actually interfaces) are defined for each category.
- Note that not all event categories are applicable to each type of GUI component. And if a component class supports an event category, it would have defined the method
```
public void addCategoryListener(CategoryListener listener)
```
for registering the event handler object.
- The parameter of the above method is an object whose class implements the CategoryListener interface. Hence that class must have implemented all methods specified by the said interface.
- Similarly the following method would also have been defined:
```
public void removeCategoryListener(CategoryListener listener)
```

Unit 9 - Example

```java
// step 1 - write event handler class that implements the listener interface
public class MyFrame extends JFrame implements ActionListener {
  private JButton button = new JButton("Click Me");
  public MyFrame() { …
    getContentPane().setLayout(new FlowLayout());
    button.addActionListener(this);
  }
  // step 2 - implement event handler
  public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == button)
      button.setText("Clicked");
  }
}
```

Unit 10

Data Structures
A data structure is a collection of data that can be handled altogether using a single name. Java's generic data structures can be divided into two categories: collections and maps.

A collection is more or less what it sounds like: a collection of objects.

A map associates objects in one set with objects in another set in the way that a dictionary associates definitions with words or a phone book associates phone numbers with names. A map is similar to what I called an "association list".

There are six collection interfaces in the Java.util package.

The most basic interface is `Collection`.

Three interfaces extend `Collection`: `Set`, `List`, and `SortedSet`.

The other two collection interfaces, `Map` and `SortedMap`, do not extend `Collection`, as they represent mappings rather than true collections.

However, these interfaces contain collection-view operations, which allow them to be manipulated as collections.

Classes implementing `Collection` may contain duplicates and null values.

Commonly used (abstract) methods

- `public boolean add(Object obj)`
- `public void clear()` // remove all elements
- `public boolean contains(Object obj)`
- `public boolean isEmpty()`
- `public boolean remove(Object obj)`
- `public int size()`
- `public Object[] toArray()`

There are several implementing classes: `ArrayList`, `LinkedList`, `Vector`, `AbstractList`.

For example, `List`:

```
List aList = new ArrayList();
aList.add(0, new Double(3.14));
```

Additional commonly used (abstract) methods:

- `public void add(int index, Object element) // elements shift to right`
- `public Object remove(int index)`
- `public Object get(int index)`
- `public void set(int index, Object element) // replace the existing element`
- `public int indexOf(Object element)`
- `public int lastIndexOf(Object element)`

There are six collection interfaces in the Java.util package.

The most basic interface is `Collection`.

Three interfaces extend `Collection`: `Set`, `List`, and `SortedSet`.

The other two collection interfaces, `Map` and `SortedMap`, do not extend `Collection`, as they represent mappings rather than true collections.

However, these interfaces contain collection-view operations, which allow them to be manipulated as collections.

Classes implementing `Collection` may contain duplicates and null values.

Commonly used (abstract) methods

- `public boolean add(Object obj)`
- `public void clear()` // remove all elements
- `public boolean contains(Object obj)`
- `public boolean isEmpty()`
- `public boolean remove(Object obj)`
- `public int size()`
- `public Object[] toArray()`

There are several implementing classes: `ArrayList`, `LinkedList`, `Vector`, `AbstractList`.

For example, `List`:

```
List aList = new ArrayList();
aList.add(0, new Double(3.14));
```

Additional commonly used (abstract) methods:

- `public void add(int index, Object element) // elements shift to right`
- `public Object remove(int index)`
- `public Object get(int index)`
- `public void set(int index, Object element) // replace the existing element`
- `public int indexOf(Object element)`
- `public int lastIndexOf(Object element)`

There are six collection interfaces in the Java.util package.

The most basic interface is `Collection`.

Three interfaces extend `Collection`: `Set`, `List`, and `SortedSet`.

The other two collection interfaces, `Map` and `SortedMap`, do not extend `Collection`, as they represent mappings rather than true collections.

However, these interfaces contain collection-view operations, which allow them to be manipulated as collections.

Classes implementing `Collection` may contain duplicates and null values.

Commonly used (abstract) methods

- `public boolean add(Object obj)`
- `public void clear()` // remove all elements
- `public boolean contains(Object obj)`
- `public boolean isEmpty()`
- `public boolean remove(Object obj)`
- `public int size()`
- `public Object[] toArray()`

There are several implementing classes: `ArrayList`, `LinkedList`, `Vector`, `AbstractList`.

For example, `List`:

```
List aList = new ArrayList();
aList.add(0, new Double(3.14));
```

Additional commonly used (abstract) methods:

- `public void add(int index, Object element) // elements shift to right`
- `public Object remove(int index)`
- `public Object get(int index)`
- `public void set(int index, Object element) // replace the existing element`
- `public int indexOf(Object element)`
- `public int lastIndexOf(Object element)`

There are six collection interfaces in the Java.util package.

The most basic interface is `Collection`.

Three interfaces extend `Collection`: `Set`, `List`, and `SortedSet`.

The other two collection interfaces, `Map` and `SortedMap`, do not extend `Collection`, as they represent mappings rather than true collections.

However, these interfaces contain collection-view operations, which allow them to be manipulated as collections.

Classes implementing `Collection` may contain duplicates and null values.

Commonly used (abstract) methods

- `public boolean add(Object obj)`
- `public void clear()` // remove all elements
- `public boolean contains(Object obj)`
- `public boolean isEmpty()`
- `public boolean remove(Object obj)`
- `public int size()`
- `public Object[] toArray()`

There are several implementing classes: `ArrayList`, `LinkedList`, `Vector`, `AbstractList`.

For example, `List`:

```
List aList = new ArrayList();
aList.add(0, new Double(3.14));
```

Additional commonly used (abstract) methods:

- `public void add(int index, Object element) // elements shift to right`
- `public Object remove(int index)`
- `public Object get(int index)`
- `public void set(int index, Object element) // replace the existing element`
- `public int indexOf(Object element)`
- `public int lastIndexOf(Object element)`

There are six collection interfaces in the Java.util package.

The most basic interface is `Collection`.

Three interfaces extend `Collection`: `Set`, `List`, and `SortedSet`.

The other two collection interfaces, `Map` and `SortedMap`, do not extend `Collection`, as they represent mappings rather than true collections.

However, these interfaces contain collection-view operations, which allow them to be manipulated as collections.

Classes implementing `Collection` may contain duplicates and null values.

Commonly used (abstract) methods

- `public boolean add(Object obj)`
- `public void clear()` // remove all elements
- `public boolean contains(Object obj)`
- `public boolean isEmpty()`
- `public boolean remove(Object obj)`
- `public int size()`
- `public Object[] toArray()`

There are several implementing classes: `ArrayList`, `LinkedList`, `Vector`, `AbstractList`.

For example, `List`:

```
List aList = new ArrayList();
aList.add(0, new Double(3.14));
```

Additional commonly used (abstract) methods:

- `public void add(int index, Object element) // elements shift to right`
- `public Object remove(int index)`
- `public Object get(int index)`
- `public void set(int index, Object element) // replace the existing element`
- `public int indexOf(Object element)`
- `public int lastIndexOf(Object element)`

There are six collection interfaces in the Java.util package.

The most basic interface is `Collection`.

Three interfaces extend `Collection`: `Set`, `List`, and `SortedSet`.

The other two collection interfaces, `Map` and `SortedMap`, do not extend `Collection`, as they represent mappings rather than true collections.

However, these interfaces contain collection-view operations, which allow them to be manipulated as collections.

Classes implementing `Collection` may contain duplicates and null values.

Commonly used (abstract) methods

- `public boolean add(Object obj)`
- `public void clear()` // remove all elements
- `public boolean contains(Object obj)`
- `public boolean isEmpty()`
- `public boolean remove(Object obj)`
- `public int size()`
- `public Object[] toArray()`

There are several implementing classes: `ArrayList`, `LinkedList`, `Vector`, `AbstractList`.

For example, `List`:

```
List aList = new ArrayList();
aList.add(0, new Double(3.14));
```

Additional commonly used (abstract) methods:

- `public void add(int index, Object element) // elements shift to right`
- `public Object remove(int index)`
- `public Object get(int index)`
- `public void set(int index, Object element) // replace the existing element`
- `public int indexOf(Object element)`
- `public int lastIndexOf(Object element)"
**Unit 10 - Iterator**

- can write code for printing all the items in any collection. Suppose that coll is of type Collection:
  ```java
  Collection coll = new ...; // collection of staff
  Iterator iter = coll.iterator();
  while ( iter.hasNext() ) {
    Staff item = (Staff) iter.next();
    System.out.println(item);
  }
  ```

---

**Unit 10 – Wrapper class**

- collection classes can only store elements of object, but not primitive value. To resolve this problem, Java provides 8 wrapper classes for the 8 primitive types to make these primitive values appear as objects.
  ```java
  Double doubleObj = new Double("3.14"); // accept a string
  Double doubleObj = new Double(3.14); // accept a double
  double d = doubleObj.doubleValue(); // primitiveValue()
  ```

- Class hierarchy of the wrapper classes:
  ```java
  Object
  |   Boolean, Character (for char)
  |   Number
  |     - Byte, Double, Float, Integer (for int), Long, Short
  ```

---

**Unit 10 – String vs. StringBuffer**

- String objects are immutable and there is large overhead to perform string concatenation
- StringBuffer objects are mutable and is preferable to concatenate strings repeatedly.

```java
String str = str1 + str2 + str3;
StringBuffer strBuf = new StringBuffer();
strBuf.append(str1).append(str2).append(str3);
String str = strBuf.toString();
```

---

**last but not least...**

**The best of luck in your exams!**